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 λ_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 λ_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

PLAS'12 June 15, Beijing, China.

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

```
// Create a hashtable of handlers:
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 processCommand(cmd) {
 // cmd.action contains the command type:
 handlers["before_" + cmd.action](cmd);
 cmd.process() ;
 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.

$pr \in \mathsf{Prog} = \mathsf{Exp}$		[programs]
$v \in \mathsf{Var}$ is a set of identifiers		[variables]
$c \in \mathsf{Const} = \mathit{String} + \mathbb{Z}$		[literals]
$lam \in \mathsf{Lam}$	$::= (\lambda\ (v_1 \ldots v_n)\ e)$	[lambda terms]
$f, æ \in \mathsf{AExp}$	$::= lam \mid v \mid c$	[atomic expressions]
	$\mid\ (op\ æ_1 \ldots æ_n)$	[primitive operations]
$op \in \mathsf{Op} \supseteq \{\texttt{+}, \texttt{len}, \texttt{get}\}$		[primitives]
$e \in \mathsf{Exp}$	$::= (\texttt{let}\ ((v\ ce))\ e)$	[expressions]
	$\mid\ æ$	[return]
	$\mid\ ce$	[tail]
$ce \in \mathsf{CExp}$	$::= (f\ æ_1 \ldots æ_n)$	[complex expressions]
	$\mid\ (\texttt{if}\ æ\ e_1\ e_2)$	[branching]
	$\mid\ \{\texttt{map}\ \mathit{field}_1 \ldots \mathit{field}_n\}$	[object creation]
	$\mid\ (\texttt{set!}\ v\ æ)$	[variable mutation]
	$\mid\ (\texttt{set!}\ [æ_{\text{object}}\ æ_{\text{field}}]\ æ_{\text{value}})$	[field mutation]
	$\mid\ (\texttt{foreach}\ v_{\text{key}}\ v_{\text{value}}\ æ_{\text{object}}\ e)$	[field introspection]
	$\mid\ (\texttt{call/cc}\ æ)$	[first-class control]
$\mathit{field} \in \mathit{Field}$	$::= [æ_{\text{name}}\ æ_{\text{value}}]$	

Figure 2. Syntax for λ_H

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 first design λ_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 λ_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 produces 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:

- λ_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: λ_H

Figure 2 presents the syntax of λ_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.

λ_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:

$$s \in \mathit{String} = \mathsf{C}^*$$
$$b \in \mathsf{C} = \{\mathsf{a}, \mathsf{b}, \ldots\}\,.$$

To simplify analysis, the syntax for λ_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 λ_H by adding an additional taint store, as present 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.

Continuations encode the program stack, and λ_H needs two kinds of frames: one to denote return to a let-bound expression (**letk**), and another to denote return to the interior of a introspective loop (**fork**). The **halt** continuation lurks at the bottom of the stack to terminate the program.

Time-stamps are required to strictly increase each time the machine steps forward, so they can serve as a source of freshness for al-

$$
\begin{aligned}
\varsigma \in \Sigma &= \mathsf{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{Taint} \times \mathit{Kont} \times \mathit{Time} \\
\rho \in \mathit{Env} &= \mathsf{Var} \rightharpoonup \mathit{Addr} \\
\sigma \in \mathit{Store} &= \mathit{Addr} \rightharpoonup D \\
\sigma^T \in \mathit{Taint} &= \mathit{Addr} \rightharpoonup T \\
d \in D &= \mathit{Clo} + \mathit{Kont} + \mathit{OLoc} + \mathit{String} + \mathbb{Z} \\
d^T \in T &= \{\mathtt{tainted}, \mathtt{untainted}\} \\
\mathit{clo} \in \mathit{Clo} &= \mathsf{Lam} \times \mathit{Env} \\
\kappa \in \mathit{Kont} &::= \mathbf{letk}(v, e, \rho, \kappa) \\
&\mid \ \mathbf{fork}(v_{\text{key}}, v_{\text{value}}, e, \rho, o, \kappa) \\
&\mid \ \mathbf{halt} \\
o \in \mathit{Obj} &= \mathit{String} \rightharpoonup D \\
a \in \mathit{Addr} &::= \mathbf{bindaddr}(v, t) \\
&\mid \ \mathbf{fieldaddr}(\ell, s)
\end{aligned}
$$

$\ell \in \mathit{OLoc}$ is an infinite set of object locations

$t \in \mathit{Time}$ is an infinite, ordered set of times

Figure 3. Taint analysis as an extension to the standard concrete state-space for a CESK-style machine.

locating entities like addresses. (During analysis, engineering the structure of time-stamps to contain execution context allows them to tune the degree of context-sensitivity.)

We will skip defining the transition relation for this machine. The definition is not only straightforward, but also unnecessary for our purposes, since we will abandon this machine for a reformulation in the next section.

In particular, to model objects (and hash maps) in this semantics, objects are mapped to a base object location (some ℓ in the set OLoc). The address of an individual field comes from pairing this location with the field name, s: $\mathbf{fieldaddr}(\ell, s)$.

For example, suppose we want to evaluate the expression or the taint property of `(get req "data")` (which would be written `req["data"]` in many dynamic languages). First, we look up the variable `req` in the current environment (ρ) to find it lives at binding $\mathbf{bindaddr}(\mathtt{req}, 10)$. We then look up the address $\mathbf{bindaddr}(\mathtt{req}, 10)$ in the store to find it yields object location ℓ. To look up the key `"data"`, we would then read the address $\mathbf{fieldaddr}(\ell, \mathtt{data})$ from the store (or the taint store in the case of taint property evaluation).

For a language like Java, where the fields in a class are determined statically, this is a reasonable structure. And, to be sure, one *can* model a dynamic language with flexible objects in this fashion. However, this representation is awkward even in some corners of the concrete semantics, and it becomes especially burdensome under a straightforward structural abstraction.

3.1 Attempting structural abstraction

Let us consider a structural abstraction of the concrete state-space to highlight the awkwardness that results in the abstract state-space. A structural abstraction α is a *family* of abstraction functions that lifts abstraction element-wise across tuples:

$$\alpha(x_1, \ldots, x_n) = (\alpha(x_1), \ldots, \alpha(x_n)),$$

monotonically and point-wise across functions:

$$\alpha(f) = \lambda \hat{x}. \bigsqcup_{\alpha(x) \sqsubseteq \hat{x}} \alpha(f(x)),$$

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 λ_H, these leaves are strings (String), integers ($\mathbb{Z}$), times (Time) and object/hash-map locations (OLoc), . We use Galois connections to abstract them as follows:

Strings (String) For now, we leave the abstract domain for strings opaque as the lattice String. That is, we can assume that there exists a Galois connection between sets of concrete strings and abstract strings:

$$(\mathcal{P}(\mathit{String}), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\widehat{\mathit{String}}, \sqsubseteq).$$

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:

$$(\mathcal{P}(\mathbb{Z}), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\hat{\mathbb{Z}}, \sqsubseteq).$$

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

Time (Time) We can assume that the analysis designer is supplying an abstraction on times:

$$(\mathcal{P}(\mathit{Time}), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\widehat{\mathit{Time}}_\perp^\top, \sqsubseteq).$$

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

Object/Hash-map Locations (OLoc) We assume a similar Galois connection for object locations:

$$(\mathcal{P}(\mathit{OLoc}), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\widehat{\mathit{OLoc}}_\perp^\top, \sqsubseteq).$$

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:

$$
\begin{aligned}
\alpha(\mathbf{bindaddr}(v, t)) &= \mathbf{bindaddr}(v, \alpha\{t\}) \\
\alpha(\mathbf{fieldaddr}(\ell, s)) &= \mathbf{fieldaddr}(\alpha\{\ell\}, \alpha\{s\}).
\end{aligned}
$$

which yields the following abstract address space:

$$
\begin{aligned}
\hat{a} \in \widehat{\mathit{Addr}} &::= \mathbf{bindaddr}(v, \hat{t}) \\
&\mid \ \mathbf{fieldaddr}(\hat{\ell}, \hat{s}).
\end{aligned}
$$

Because $\widehat{String}$ is not a flat lattice, addresses are not flat either, which means that both abstract store and abstract taint store must be *monotonic* maps:

$$\widehat{Store} = \widehat{Addr} \xrightarrow{\text{mon}} \hat{D}$$
$$\widehat{Taint} = \widehat{Addr} \xrightarrow{\text{mon}} \hat{T},$$

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

$$x \sqsubseteq y \text{ implies } f(x) \sqsubseteq f(y).$$

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:

```
; 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 $\{\texttt{foo}, \texttt{bar}\}$. 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:

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

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 $\{\texttt{foo}, \texttt{bar}, \texttt{qux}\}$. It is clearly the case that $\{\texttt{foo}, \texttt{bar}\} \subseteq \{\texttt{foo}, \texttt{bar}, \texttt{qux}\}$, 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:

$$\lceil x \mapsto y \rceil = \lambda x'. \begin{cases} y & x \sqsubseteq x' \\ \bot & \text{otherwise.} \end{cases}$$

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

$$\hat{\sigma}' = \hat{\sigma} \sqcup \lceil \mathbf{fieldaddr}(\hat{\ell}, \{\texttt{foo}, \texttt{bar}\}) \mapsto \alpha(3) \rceil,$$

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 $\{\texttt{foo}\}$. 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:

$$\hat{\sigma}' = \hat{\sigma} \sqcup \bigsqcup \left\{ \lceil \hat{a} \mapsto \alpha(3) \rceil : \bot \sqsubseteq \hat{a} \sqsubseteq \mathbf{fieldaddr}(\hat{\ell}, \{\texttt{foo}, \texttt{bar}\}) \right\},$$

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 this sound 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"—$\sqcup\!\!\!\sqcup$:

$$f \sqcup\!\!\!\sqcup [x \mapsto y] = f \sqcup \bigsqcup_{\bot \sqsubset x' \sqsubseteq x} \lceil x' \mapsto y \rceil.$$

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.

$$
\begin{aligned}
\varsigma \in \Sigma &= \mathsf{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{OStore} \times \mathit{Taint} \times \mathit{OTaint} \\
&\quad \times \mathit{Kont} \times \mathit{Time} \\
\rho \in \mathit{Env} &= \mathsf{Var} \rightharpoonup \mathit{Addr} \\
\sigma \in \mathit{Store} &= \mathit{Addr} \rightharpoonup D \\
\omega \in \mathit{OStore} &= \mathit{OLoc} \rightharpoonup \mathit{Obj} \\
\sigma^T \in \mathit{Taint} &= \mathit{Addr} \rightharpoonup T \\
\omega^T \in \mathit{OTaint} &= \mathit{OLoc} \rightharpoonup \mathit{Obj}^T \\
d \in D &= \mathit{Clo} + \mathit{Kont} + \mathit{OLoc} + \mathit{String} + \mathbb{Z} \\
d^T \in T &= \{\mathtt{tainted}, \mathtt{untainted}\} \\
\mathit{clo} \in \mathit{Clo} &= \mathsf{Lam} \times \mathit{Env} \\
o \in \mathit{Obj} &= \mathit{String} \rightharpoonup D \\
o^T \in \mathit{Obj}^T &= \mathit{String} \rightharpoonup T \\
\kappa \in \mathit{Kont} &::= \mathbf{letk}(v, e, \rho, \kappa) \\
&\quad \mid \ \mathbf{fork}(v_{\text{key}}, v_{\text{value}}, e, \rho, o, o^T, \kappa) \\
&\quad \mid \ \mathbf{halt} \\
a \in \mathit{Addr} &::= \mathbf{bindaddr}(v, t) \\
\ell \in \mathit{OLoc} & \text{ is an infinite set of object locations} \\
t \in \mathit{Time} & \text{ is an infinite, ordered set of times}
\end{aligned}
$$

Figure 4. A concrete state-space designed for abstraction.

In the following subsections, we will transform the concrete state-space to use a separate object store, and then we will *Curry* the hash maps within this store to isolate the monotonicity in Section 4.1. Then we develop abstract semantics in Section 4.2, following that we will provide a proof sketch of the soundness of HFTA in Section 4.2.4. 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 λ_H.

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*:

$$\mathit{OStore} = \mathit{OLoc} \rightharpoonup \mathit{String} \rightharpoonup D.$$

- 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:

$$\mathit{Taint} = \mathit{Addr} \rightharpoonup T,$$

  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:

$$\mathit{OTaint} = \mathit{OLoc} \rightharpoonup \mathit{String} \rightharpoonup T.$$

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:

$$
\begin{aligned}
\widehat{\mathit{OStore}} &= \widehat{\mathit{OLoc}} \to \widehat{\mathit{String}} \xrightarrow{\text{mon}} \hat{D} \\
\widehat{\mathit{OTaint}} &= \widehat{\mathit{OLoc}} \to \widehat{\mathit{String}} \xrightarrow{\text{mon}} \hat{T}.
\end{aligned}
$$

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, $\mathcal{I} : \mathsf{Prog} \to \mathit{State}$:

$$\mathcal{I}(pr) = (pr, [], [], [], [], [], \mathbf{halt}, t_0),$$

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, $(\llbracket(f\ æ_1 \ldots æ_n)\rrbracket, \rho, \sigma, \omega, \sigma^T, \omega^T, \kappa, t)$

$$
\begin{aligned}
&\Rightarrow (e, \rho'', \sigma', \omega, \sigma^{T\prime}, \omega^T, \kappa, t'), \text{ where} \\
t' &= t + 1 \\
(\llbracket(\lambda\ (v_1 \ldots v_n)\ e)\rrbracket, \rho') &= \mathcal{A}(f, \rho, \sigma, \omega) \\
\rho'' &= \rho'[v_i \mapsto a_i] \\
a_i &= \mathbf{bindaddr}(v_i, t') \\
d_i &= \mathcal{A}(æ_i, \rho, \sigma, \omega) \\
d_i^T &= \mathcal{A}^T(æ_i, \rho, \sigma^T, \omega^T) \\
\sigma' &= \sigma[a_i \mapsto d_i] \\
\sigma^{T\prime} &= \sigma^T[a_i \mapsto d_i^T],
\end{aligned}
$$

where $\mathcal{A} : \mathsf{AExp} \times \mathit{Env} \times \mathit{Store} \times \mathit{OStore} \rightharpoonup D$ is the atomic argument evaluator:

$$
\begin{aligned}
\mathcal{A}(c, \rho, \sigma, \omega) &= c \\
\mathcal{A}(v, \rho, \sigma, \omega) &= \sigma(\rho(v)) \\
\mathcal{A}(\mathit{lam}, \rho, \sigma, \omega) &= (\mathit{lam}, \rho) \\
\mathcal{A}(\llbracket(op\ æ_1 \ldots æ_n)\rrbracket, \rho, \sigma, \omega) &= \mathcal{O}(op)(\sigma, \omega)\langle d_1, \ldots, d_n\rangle \\
&\quad \text{where } d_i = \mathcal{A}(æ_i, \rho, \sigma, \omega),
\end{aligned}
$$

where $\mathcal{O} : \mathsf{Op} \to \mathit{Store} \times \mathit{OStore} \to D^* \rightharpoonup D$ is the primitive operation evaluator, and where $\mathcal{A}^T : \mathsf{AExp} \times \mathit{Env} \times \mathit{Taint} \times \mathit{OTaint} \to T$ is like $\mathcal{A}$, but is evaluating taintedness:

$$
\begin{aligned}
\mathcal{A}^T(c, \rho, \sigma^T, \omega^T) &= \mathtt{untainted} \\
\mathcal{A}^T(v, \rho, \sigma^T, \omega^T) &= \sigma^T(\rho(v)) \\
\mathcal{A}^T(\mathit{lam}, \rho, \sigma^T, \omega^T) &= \mathtt{untainted} \\
\mathcal{A}^T(\llbracket(op\ æ_1 \ldots æ_n)\rrbracket, \hat{\rho}, \sigma^T, \omega^T) &= \mathcal{O}^T(op)(\sigma^T, \omega^T)\langle d_1^T, \ldots, d_n^T\rangle \\
&\quad \text{where } d_i^T = \mathcal{A}^T(æ_i, \rho, \sigma^T, \omega^T),
\end{aligned}
$$

where $\mathcal{O}^T : \mathsf{Op} \to \mathit{Taint} \times \mathit{OTaint} \to T^* \to T$ is the taint evaluator on primitive operators.

`get` handles accessing a key in a hash map as:

$$
\begin{aligned}
\mathcal{O}(\mathtt{get})(\sigma, \omega)\langle \ell, s\rangle &= \omega(\ell)(s) \\
\mathcal{O}^T(\mathtt{get})(\sigma^T, \omega^T)\langle \ell, s\rangle &= \omega^T(\ell)(s).
\end{aligned}
$$

The rule for hash map update is also straightforward:

$$
\begin{aligned}
&([\![\texttt{(set! [}æ_{\text{object}}\ æ_{\text{key}}\texttt{]}\ æ_{\text{value}}\texttt{)}]\!], \rho, \sigma, \omega, \sigma^T, \omega^T, \kappa, t) \\
&\qquad \Rightarrow \mathit{apply}(\kappa, d_{\text{value}}, \sigma, \omega', \sigma^T, \omega^{T'}, t'), \text{ where} \\
&\qquad t' = t + 1 \\
&\qquad \ell = \mathcal{A}(æ_{\text{object}}, \rho, \sigma, \omega) \\
&\qquad o = \omega(\ell) \\
&\qquad o^T = \omega^T(\ell) \\
&\qquad s_{\text{key}} = \mathcal{A}(æ_{\text{key}}, \rho, \sigma, \omega) \\
&\qquad d_{\text{value}} = \mathcal{A}(æ_{\text{value}}, \rho, \sigma, \omega) \\
&\qquad d^T_{\text{value}} = \mathcal{A}^T(æ_{\text{value}}, \rho, \sigma^T, \omega^T) \\
&\qquad o' = o[s_{\text{key}} \mapsto d_{\text{value}}] \\
&\qquad \omega' = \omega[\ell \mapsto o'] \\
&\qquad o^{T'} = o^T[s_{\text{key}} \mapsto d^T_{\text{value}}] \\
&\qquad \omega^{T'} = \omega^T[\ell \mapsto o^{T'}],
\end{aligned}
$$

The continuation-application helper $\mathit{apply} : \mathit{Kont} \times D \times T \times \mathit{Store} \times \mathit{OStore} \times \mathit{Taint} \times \mathit{OTaint} \times \mathit{Time} \rightharpoonup \Sigma$ completes the transition by dispatching on the type of continuation.

For let-based continuations:

$$
\begin{aligned}
&\mathit{apply}(\mathbf{letk}(v, e, \rho, \kappa), d, d^T, \sigma, \omega, \sigma^T, \omega^T, t) \\
&\qquad = (e, \rho', \sigma', \omega, \sigma^{T'}, \omega^T, \kappa), \text{ where} \\
&\qquad a = \mathbf{bindaddr}(v, t) \\
&\qquad \rho' = \rho[v \mapsto a] \\
&\qquad \sigma' = \sigma[a \mapsto d] \\
&\qquad \sigma^{T'} = \sigma^T[a \mapsto d^T].
\end{aligned}
$$

and, *for* introspective iteration:

$$
\begin{aligned}
&\mathit{apply}(\mathbf{fork}(v, v', e, \rho, [s_1 \mapsto d_1, \ldots], \\
&\qquad [s^T_1 \mapsto d^T_1, \ldots], \kappa), d, d^T, \sigma, \omega, \sigma^T, \omega^T, t) \\
&\qquad = (e, \rho', \sigma', \omega, \kappa'), \text{ where} \\
&\qquad a = \mathbf{bindaddr}(v, t) \\
&\qquad a' = \mathbf{bindaddr}(v', t) \\
&\qquad \rho' = \rho[v \mapsto a, v' \mapsto a'] \\
&\qquad \sigma' = \sigma[a \mapsto s_1, a' \mapsto d_1] \\
&\qquad \sigma^{T'} = \sigma^T[a \mapsto s^T_1, a' \mapsto d^T_1] \\
&\qquad \kappa' = \mathbf{fork}(v, v', e, \rho, [s_2 \mapsto d_2, \ldots], [s^T_2 \mapsto d^T_2, \ldots], \kappa).
\end{aligned}
$$

At this point, the definition of the remaining rules is straightforward, and follow the style of work done by Van Horn and Might [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 $\hat{\mathcal{I}} : \mathsf{Prog} \to \hat{\Sigma}$ provides injection into the abstract state-space:

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

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

$$
\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &= \mathsf{Exp} \times \widehat{\mathit{Env}} \times \widehat{\mathit{Store}} \times \widehat{\mathit{OStore}} \times \widehat{\mathit{Taint}} \times \widehat{\mathit{OTaint}} \\
&\quad \times \widehat{\mathit{Kont}} \times \widehat{\mathit{Time}} \\
\hat{\rho} \in \widehat{\mathit{Env}} &= \mathsf{Var} \rightharpoonup \widehat{\mathit{Addr}} \\
\hat{\sigma} \in \widehat{\mathit{Store}} &= \widehat{\mathit{Addr}} \to \hat{D} \\
\hat{\omega} \in \widehat{\mathit{OStore}} &= \widehat{\mathit{OLoc}} \to \widehat{\mathit{Obj}} \\
\hat{\sigma}^T \in \widehat{\mathit{Taint}} &= \widehat{\mathit{Addr}} \to \hat{T} \\
\hat{\omega}^T \in \widehat{\mathit{OTaint}} &= \widehat{\mathit{OLoc}} \to \widehat{\mathit{Obj}}^T \\
\hat{d} \in \hat{D} &= \mathcal{P}\left(\widehat{\mathit{Clo}} + \widehat{\mathit{Kont}} + \widehat{\mathit{OLoc}} + \widehat{\mathit{String}} + \hat{\mathbb{Z}}\right) \\
\hat{d}^T \in \hat{T} &= \{\texttt{tainted}, \texttt{untainted}\} \\
\widehat{clo} \in \widehat{\mathit{Clo}} &= \mathsf{Lam} \times \widehat{\mathit{Env}} \\
\hat{o} \in \widehat{\mathit{Obj}} &= \widehat{\mathit{String}} \xrightarrow{\text{mon}} \hat{D} \\
\hat{o}^T \in \widehat{\mathit{Obj}}^T &= \widehat{\mathit{String}} \xrightarrow{\text{mon}} \hat{T} \\
\hat{\kappa} \in \widehat{\mathit{Kont}} &::= \mathbf{letk}(v, e, \hat{\rho}, \hat{\kappa}) \\
&\quad\ \mid\ \mathbf{fork}(v_{\text{key}}, v_{\text{value}}, e, \hat{\rho}, \hat{o}, \hat{o}^T, \hat{\kappa}) \\
&\quad\ \mid\ \mathbf{halt} \\
\hat{a} \in \mathit{Addr} &::= \mathbf{bindaddr}(v, \hat{t}) \\
\hat{\ell} \in \widehat{\mathit{OLoc}} &\text{ is a finite set of abstract object locations} \\
\hat{t} \in \widehat{\mathit{Time}} &\text{ is a finite set of abstract times}
\end{aligned}
$$

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:

$$(\mathcal{P}(\mathit{String}), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\widehat{\mathit{String}}, \sqsubseteq).$$

The Galois connection over integers determines reasoning over arithmetic:

$$(\mathcal{P}(\mathbb{Z}), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\hat{\mathbb{Z}}, \sqsubseteq).$$

The Galois connection over times determines the context-sensitivity:

$$(\mathcal{P}(\mathit{Time}), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\widehat{\mathit{Time}}^{\top}_{\bot}, \sqsubseteq).$$

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:

$$
\begin{aligned}
\mathit{Time} &= \mathsf{Exp}^* \\
\widehat{\mathit{Time}} &= \mathsf{Exp}^k,
\end{aligned}
$$

$$(e, \hat{\rho}, \hat{\sigma}, \hat{\omega}, \hat{\sigma}^T, \hat{\omega}^T, \hat{\kappa}, \hat{t}) \sqsubseteq (e', \hat{\rho}', \hat{\sigma}', \hat{\omega}', \hat{\sigma}^{T'}, \hat{\omega}^{T'}, \hat{\kappa}', \hat{t}')$$
$$\text{iff } e = e' \text{ and } \hat{\rho} \sqsubseteq \hat{\rho}' \text{ and } \hat{\sigma} \sqsubseteq \hat{\sigma}' \text{ and } \hat{\omega} \sqsubseteq \hat{\omega}'$$
$$\text{and } \hat{\sigma}^T \sqsubseteq \hat{\sigma}^{T'} \text{ and } \hat{\omega}^T \sqsubseteq \hat{\omega}^{T'} \text{ and } \hat{\kappa} \sqsubseteq \hat{\kappa}' \text{ and } \hat{t} \sqsubseteq \hat{t}'$$

$$\hat{\rho} \sqsubseteq \hat{\rho}' \text{ iff } \hat{\rho}(v) \sqsubseteq \hat{\rho}'(v) \text{ for all } v \in dom(\hat{\rho})$$
$$\hat{\sigma} \sqsubseteq \hat{\sigma}' \text{ iff } \hat{\sigma}(\hat{a}) \sqsubseteq \hat{\sigma}'(\hat{a}) \text{ for all } \hat{a} \in dom(\hat{\sigma})$$
$$\hat{\omega} \sqsubseteq \hat{\omega}' \text{ iff } \hat{\omega}(\hat{\ell}) \sqsubseteq \hat{\omega}'(\hat{\ell}) \text{ for all } \hat{\ell} \in dom(\hat{\omega})$$
$$\hat{\sigma}^T \sqsubseteq \hat{\sigma}^{T'} \text{ iff } \hat{\sigma}^T(\hat{a}) \sqsubseteq \hat{\sigma}^{T'}(\hat{a}) \text{ for all } \hat{a} \in dom(\hat{\sigma}^T)$$
$$\hat{\omega}^T \sqsubseteq \hat{\omega}^{T'} \text{ iff } \hat{\omega}^T(\hat{\ell}) \sqsubseteq \hat{\omega}^{T'}(\hat{\ell}) \text{ for all } \hat{\ell} \in dom(\hat{\omega}^T)$$

$$\mathbf{letk}(v, e, \hat{\rho}, \hat{\kappa}) \sqsubseteq \mathbf{letk}(v', e', \hat{\rho}', \hat{\kappa}') \text{ iff } v = v' \text{ and } e = e'$$
$$\text{and } \hat{\rho} \sqsubseteq \hat{\rho}' \text{ and } \hat{\kappa} \sqsubseteq \hat{\kappa}'.$$

$$\mathbf{fork}(v_1, v_2, e, \hat{\rho}, \hat{o}, \hat{o}^T, \hat{\kappa}) \sqsubseteq \mathbf{fork}(v_1', v_2', e', \hat{\rho}', \hat{o}', , \hat{o}^{T'}, \hat{\kappa}')$$
$$\text{iff } v_1 = v_1' \text{ and } v_2 = v_2' \text{ and } e = e'$$
$$\text{and } \hat{\rho} \sqsubseteq \hat{\rho}' \text{ and } \hat{o} \sqsubseteq \hat{o}' \text{ and } \hat{o}^T \sqsubseteq \hat{o}^{T'} \text{ and } \hat{\kappa} \sqsubseteq \hat{\kappa}'$$

$$\mathbf{halt} \sqsubseteq \mathbf{halt}$$

$$\hat{o} \sqsubseteq \hat{o}' \text{ iff } \hat{o}(\hat{s}) \sqsubseteq \hat{o}'(\hat{s}) \text{ for all } \hat{s} \in dom(\hat{o})$$
$$\hat{o}^T \sqsubseteq \hat{o}^T \text{ iff } \hat{o}^T(\hat{s}) \sqsubseteq \hat{o}^{T'}(\hat{s}) \text{ for all } \hat{s} \in dom(\hat{o}^T)$$

$$\hat{d} \sqsubseteq \hat{d}' \text{ iff for all } \hat{x} \in \hat{d} \text{ there exists } \hat{x}' \in \hat{d}' \text{ such that } \hat{x} \sqsubseteq \hat{x}'$$
$$\hat{d}^T \sqsubseteq \hat{d}^{T'} \text{ iff for all } \hat{x} \in \hat{d}^T \text{ there exists } \hat{x}' \in \hat{d}^{T'} \text{ such that } \hat{x} \sqsubseteq \hat{x}'$$

$$(lam, \hat{\rho}) \sqsubseteq (lam', \hat{\rho}') \text{ iff } lam = lam' \text{ and } \hat{\rho} \sqsubseteq \hat{\rho}'$$

$$\mathbf{bindaddr}(v, \hat{t}) \sqsubseteq \mathbf{bindaddr}(v', \hat{t}') \text{ iff } v = v' \text{ and } \hat{t} = \hat{t}'$$

Figure 6. A partial order on abstract states.

so that abstraction will select the first k expressions:

$$\alpha \{\langle e_1, \ldots, e_n \rangle\} = \langle e_1, \ldots, e_k \rangle,$$

while concretization will append every possible tail:

$$\gamma(\top) = \mathsf{Exp}^*$$
$$\gamma(\vec{e}) = \{\vec{e}\} \times \mathsf{Exp}^*$$
$$\gamma(\bot) = \emptyset.$$

The Galois connection over object locations determines object-polyvariance:

$$(\mathcal{P}(OLoc), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\widehat{OLoc}_{\bot}^{\top}, \sqsubseteq).$$

For a monovariant (0CFA-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 current time, so that:

$$OLoc = \mathsf{Exp} \times Time$$
$$\widehat{OLoc} = \mathsf{Exp}$$
$$\alpha \{(e, t)\} = e$$
$$\gamma(\top) = OLoc$$
$$\gamma(e) = \{expr\} \times Time$$
$$\gamma(\bot) = \emptyset.$$

$$\alpha(e, \rho, \sigma, \omega, \sigma^T, \omega^T, \kappa, t) = (e, \alpha(\rho), \alpha(\sigma), \alpha(\omega), \alpha(\sigma^T), \alpha(\omega^T), \alpha(\kappa), \alpha\{t\})$$

$$\alpha(\rho) = \lambda v.\alpha(\rho(v))$$
$$\alpha(\sigma) = \lambda \hat{a}. \bigsqcup_{\alpha(a) = \hat{a}} \alpha(\sigma(a))$$
$$\alpha(\omega) = \lambda \hat{\ell}. \bigsqcup_{\alpha(\ell) = \hat{\ell}} \alpha(\omega(\ell))$$
$$\alpha(\sigma^T) = \lambda \hat{a}. \bigsqcup_{\alpha(a) = \hat{a}} \alpha(\sigma^T(a))$$
$$\alpha(\omega^T) = \lambda \hat{\ell}. \bigsqcup_{\alpha(\ell) = \hat{\ell}} \alpha(\omega^T(\ell))$$

$$\alpha(\mathbf{letk}(v, e, \rho, \kappa)) = \mathbf{letk}(v, e, \alpha(\rho), \alpha(\kappa))$$
$$\alpha(\mathbf{fork}(v, v', e, \rho, o, o^T, \kappa)) = \mathbf{fork}(v, v', e, \alpha(\rho), \alpha(o), \alpha(o^T), \alpha(\kappa))$$
$$\alpha(\mathbf{halt}) = \mathbf{halt}$$

$$\alpha(o) = \lambda \hat{s}. \bigsqcup_{\alpha(s) \sqsubseteq \hat{s}} \alpha(o(s))$$
$$\alpha(o^T) = \lambda \hat{s}. \bigsqcup_{\alpha(s) \sqsubseteq \hat{s}} \alpha(o^T(s))$$

$$\alpha(lam, \rho) = (lam, \alpha(\rho))$$

$$\alpha(\mathbf{bindaddr}(v, t)) = \mathbf{bindaddr}(v, \alpha\{t\}).$$

Figure 7. A structural abstraction.

4.2.3 Abstract transition rules

Our next step is to synthesize the abstract transition relation, $(\leadsto) \subseteq \hat{\Sigma} \times \hat{\Sigma}$.

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 $\hat{t} + 1$ expands into the recording of execution context appropriate for the desired context-sensitivity, and the abstract object allocator $\widehat{alloc} : \hat{\Sigma} \to \widehat{OLoc}$ returns the abstract location for this object.

To match up with the earlier 0CFA-like object-polyvariance example, the correct allocator in this case would return the current ex-

$$\overbrace{(\llbracket \texttt{\{map [}\mathit{æ}_1\ \mathit{æ}'_1\texttt{]} \ldots \texttt{[}\mathit{æ}_n\ \mathit{æ}'_n\texttt{]\}} \rrbracket, \hat{\rho}, \hat{\sigma}, \hat{\omega}, \hat{\sigma}^T, \hat{\omega}^T, \hat{\kappa}, \hat{t})}^{\hat{\varsigma}}$$

$$\begin{aligned}
&\leadsto \widehat{apply}(\hat{\kappa}, \{\hat{\ell}\}, \hat{\sigma}, \hat{\omega}', \hat{\sigma}^T, \hat{\omega}^{T\prime}, \hat{t}'), \text{ where} \\
\hat{t}' &= \hat{t} + 1 \\
\hat{\ell} &= \widehat{alloc}(\hat{\varsigma}) \\
\hat{d}_i &= \hat{\mathcal{A}}(\mathit{æ}_i, \hat{\rho}, \hat{\sigma}, \hat{\omega}) \\
\hat{d}'_i &= \hat{\mathcal{A}}(\mathit{æ}'_i, \hat{\rho}, \hat{\sigma}, \hat{\omega}) \\
\hat{d}_i^{T\prime} &= \hat{\mathcal{A}}^T(\mathit{æ}'_i, \hat{\rho}, \hat{\sigma}^T, \hat{\omega}^T) \\
\hat{s}_i &\in \hat{d}_i \\
\hat{o} &= \hat{\omega}(\hat{\ell}), \hat{o}^T = \hat{\omega}^T(\hat{\ell}) \\
\hat{o}' &= \hat{o} \uplus [\hat{s}_i \mapsto \hat{d}'_i] \\
\hat{\omega}' &= \hat{\omega} \sqcup [\hat{\ell} \mapsto \hat{o}'] \\
\hat{o}^{T\prime} &= \hat{o}^T \uplus [\hat{s}_i \mapsto \hat{d}_i^{T\prime}] \\
\hat{\omega}^{T\prime} &= \hat{\omega}^T \sqcup [\hat{\ell} \mapsto \hat{o}^{T\prime}],
\end{aligned}$$

Figure 8. Allocating a hash map (object)

pression:

$$\widehat{alloc}(e, \hat{\rho}, \hat{\sigma}, \hat{\omega}, \hat{\sigma}^T, \omega^T, \hat{\kappa}, \hat{t}) = e.$$

The abstract argument evaluator $\hat{\mathcal{A}} : \mathsf{AExp} \times \widehat{Env} \times \widehat{Store} \times \widehat{OStore} \to \hat{D}$ mimics its concrete counterpart:

$$\begin{aligned}
\hat{\mathcal{A}}(c, \hat{\rho}, \hat{\sigma}, \hat{\omega}) &= \alpha\{c\} \\
\hat{\mathcal{A}}(v, \hat{\rho}, \hat{\sigma}, \hat{\omega}) &= \hat{\sigma}(\hat{\rho}(v)) \\
\hat{\mathcal{A}}(lam, \hat{\rho}, \hat{\sigma}, \hat{\omega}) &= \{(lam, \hat{\rho})\} \\
\hat{\mathcal{A}}(\llbracket \texttt{(}op\ \mathit{æ}_1 \ldots \mathit{æ}_n\texttt{)} \rrbracket, \hat{\rho}, \hat{\sigma}, \hat{\omega}) &= \hat{\mathcal{O}}(op)(\hat{\sigma}, \hat{\omega})\langle \hat{d}_1, \ldots, \hat{d}_n \rangle \\
&\quad \text{where } \hat{d}_i = \hat{\mathcal{A}}(\mathit{æ}_i, \hat{\rho}, \hat{\sigma}, \hat{\omega}),
\end{aligned}$$

where $\hat{\mathcal{O}} : \mathsf{Op} \to \widehat{Store} \times \widehat{OStore} \to \hat{D}^* \to \hat{D}$ is the abstract primitive operation evaluator.

The abstract taint evaluator $\hat{\mathcal{A}}^T : \mathsf{AExp} \times \widehat{Env} \times \widehat{Taint} \times \widehat{OTaint} \to \hat{T}$ is like $\hat{\mathcal{A}}$, but evaluates taintedness:

$$\begin{aligned}
\hat{\mathcal{A}}^T(c, \hat{\rho}, \hat{\sigma}^T, \hat{\omega}^T) &= \texttt{untainted} \\
\hat{\mathcal{A}}^T(v, \hat{\rho}, \hat{\sigma}^T, \hat{\omega}^T) &= \hat{\sigma}^T(\hat{\rho}(v)) \\
\hat{\mathcal{A}}^T(lam, \hat{\rho}, \hat{\sigma}^T, \hat{\omega}^T) &= \texttt{untainted} \\
\hat{\mathcal{A}}^T(\llbracket \texttt{(}op\ \mathit{æ}_1 \ldots \mathit{æ}_n\texttt{)} \rrbracket, \hat{\rho}, \hat{\sigma}^T, \hat{\omega}^T) &= \hat{\mathcal{O}}^T(op)(\hat{\sigma}^T, \hat{\omega}^T)\langle \hat{d}_1^T, \ldots, \hat{d}_n^T \rangle \\
&\quad \text{where } \hat{d}_i^T = \hat{\mathcal{A}}^T(\mathit{æ}_i, \hat{\rho}, \hat{\sigma}^T, \hat{\omega}^T),
\end{aligned}$$

where $\hat{\mathcal{O}}^T : \mathsf{Op} \to \widehat{Taint} \times \widehat{OTaint} \to \hat{T}^* \to \hat{T}$ is the abstract primitive operation evaluator on taint property.

`get` handles accessing a key in a hash map:

$$\hat{\mathcal{O}}(\texttt{get})(\hat{\sigma}, \hat{\omega})\langle \hat{d}, \hat{d}' \rangle = \bigsqcup_{\hat{\ell} \in \hat{d}} \bigsqcup_{\hat{s} \in \hat{d}'} \hat{\omega}(\hat{\ell})(\hat{s}),$$

$$\hat{\mathcal{O}}^T(\texttt{get})(\hat{\sigma}^T, \hat{\omega}^T)\langle \hat{d}, \hat{d}' \rangle = \bigsqcup_{\hat{\ell} \in \hat{d}} \bigsqcup_{\hat{s} \in \hat{d}'} \hat{\omega}^T(\hat{\ell})(\hat{s}).$$

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

$$\begin{aligned}
&\widehat{apply}(\mathbf{letk}(v, e, \hat{\rho}, \hat{\kappa}), \hat{d}, \hat{d}^T, \hat{\sigma}, \hat{\omega}, \hat{\sigma}^T, \hat{\omega}^T, \hat{t}) \\
&\qquad = (e, \hat{\rho}', \hat{\sigma}', \hat{\omega}, \hat{\sigma}^{T\prime}, \hat{\omega}^T, \hat{\kappa}), \text{ where} \\
\hat{a} &= \mathbf{bindaddr}(v, \hat{t}) \\
\hat{\rho}' &= \hat{\rho}[v \mapsto \hat{a}] \\
\hat{\sigma}' &= \hat{\sigma}[\hat{a} \mapsto \hat{d}] \\
\hat{\sigma}^{T\prime} &= \hat{\sigma}^T[\hat{a} \mapsto \hat{d}^T],
\end{aligned}$$

and in the case of for introspective iteration:[1]

$$\begin{aligned}
&\widehat{apply}(\mathbf{fork}(v, v', e, \hat{\rho}, [\hat{s}_1 \mapsto \hat{d}_1, \ldots], \\
&\quad [\hat{s}_1^T \mapsto \hat{d}_1^T, \ldots], \hat{\kappa}), \hat{d}, \hat{d}^T, \hat{\sigma}, \hat{\omega}, \hat{\sigma}^T, \hat{\omega}^T, \hat{t}) \\
&\quad = \left\{(e, \hat{\rho}', \hat{\sigma}', \hat{\omega}, \hat{\sigma}^{T\prime}, \hat{\omega}^T, \hat{\kappa}') : \hat{\kappa}' \in \hat{K}'\right\} \\
&\quad \cup \widehat{apply}(\hat{\kappa}, \hat{d}, \hat{d}^T, \hat{\sigma}, \hat{\omega}, \hat{\sigma}^T, \hat{\omega}^T, \hat{t}), \text{ where} \\
\hat{a} &= \mathbf{bindaddr}(v, \hat{t}) \\
\hat{a}' &= \mathbf{bindaddr}(v', \hat{t}) \\
\hat{\rho}' &= \hat{\rho}[v \mapsto \hat{a}, v' \mapsto \hat{a}'] \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{s}_1, \hat{a}' \mapsto \hat{d}_1] \\
\hat{\sigma}^{T\prime} &= \hat{\sigma}^T \sqcup [\hat{a} \mapsto \hat{s}_1^T, \hat{a}' \mapsto \hat{d}_1^T] \\
\hat{K}' &= \{\mathbf{fork}(v, v', e, \hat{\rho}, [\hat{s}_1 \mapsto \hat{d}_1, \ldots], [\hat{s}_1^T \mapsto \hat{d}_1^T, \ldots], \hat{\kappa}), \\
&\qquad \mathbf{fork}(v, v', e, \hat{\rho}, [\hat{s}_2 \mapsto \hat{d}_2, \ldots], [\hat{s}_2^T \mapsto \hat{d}_2^T, \ldots], \hat{\kappa})\}.
\end{aligned}$$

The jarring dissonance of this rule for the function $\widehat{apply}$ with respect to its counterpart in the concrete *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:

```
(foreach k v o
  (set! i (+ 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 loop 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:*

$$\alpha(\varsigma) \sqsubseteq \hat{\varsigma} \textit{ and } \varsigma \Rightarrow \varsigma',$$

[1] Should $\widehat{apply}$ recur on itself, we assume the least-fixed point solution to this equation.

then there must exist an abstract state $\hat{\varsigma}'$ such that:

$$\alpha(\varsigma') \sqsubseteq \hat{\varsigma}' \textit{ and } \hat{\varsigma} \rightsquigarrow \hat{\varsigma}'.$$

Proof. Assume $\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$ and $\varsigma \Rightarrow \varsigma'$. The high-level structure of the proof splits on cases with respect to the expression within ς. 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 Might [31], reformulate continuations to contain pointers to continuations rather than continuations themselves:

$$\begin{aligned}\hat{\kappa} \in \widehat{Kont} ::= &\ \mathbf{letk}(v, e, \hat{\rho}, \hat{a})\\ \mid &\ \mathbf{fork}(v_{\text{key}}, v_{\text{value}}, e, \hat{\rho}, \hat{o}, \hat{o}^T, \hat{a})\\ \mid &\ \mathbf{halt}\end{aligned}$$

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

$$\hat{\varsigma}_0 = \hat{\mathcal{I}}(pr),$$

and then finding the states reachable from this state:

$$\{\hat{\varsigma} : \hat{\varsigma}_0 \rightsquigarrow^* \hat{\varsigma}\}.$$

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, $\alpha^\circ : \Sigma \to \widehat{OStore}^\circ$, that measures the cardinality of abstract locations:

$$\omega^\circ \in \widehat{OStore}^\circ = \widehat{OLoc} \to \{0, 1, \infty\}$$
$$\alpha^\circ(e, \rho, \sigma, \omega, \sigma^T, \omega^T, \kappa, t) = \lambda\hat{\ell}.\widehat{size}(\gamma(\hat{\ell}) \cap dom(\omega)),$$

where:

$$\widehat{size}\,\{\} = 0$$
$$\widehat{size}\,\{x\} = 1$$
$$\widehat{size}\,\{x_1, x_2, \ldots, x_n\} = \infty.$$

We then formulate the reduced product transition of the abstractions $\alpha \times \alpha^\circ$, which effectively adds a ω° 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:

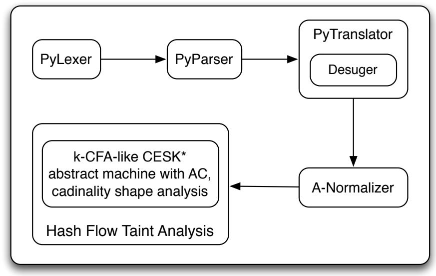

Fig. 9. Implementation of Hash Flow Taint Analysis for Python

$$([\![\texttt{(set! [}æ_{\text{object}}\ æ_{\text{key}}\texttt{]}\ æ_{\text{value}}\texttt{)}]\!], \hat{\rho}, \hat{\sigma}, \hat{\omega}, \hat{\sigma}^T, \hat{\omega}^T, \hat{\kappa}, \hat{t}, \hat{\omega}^\circ)$$

$$\begin{aligned}
&\rightsquigarrow apply(\hat{\kappa}, \hat{d}_{\text{value}}, \hat{d}^T, \hat{\sigma}, \hat{\omega}', \hat{\sigma}^T, \hat{\omega}^{T'}, \hat{t}', \hat{\omega}^\circ), \text{ where}\\
\hat{t}' &= \hat{t} + 1\\
\hat{\ell} &\in \hat{\mathcal{A}}(æ_{\text{object}}, \hat{\rho}, \hat{\sigma}, \hat{\omega})\\
\hat{o} &= \hat{\omega}(\hat{\ell})\\
\hat{s}_{\text{key}} &\in \hat{\mathcal{A}}(æ_{\text{key}}, \hat{\rho}, \hat{\sigma}, \hat{\omega})\\
\hat{d}_{\text{value}} &= \hat{\mathcal{A}}(æ_{\text{value}}, \hat{\rho}, \hat{\sigma}, \hat{\omega})\\
\hat{o}' &= \begin{cases} \hat{o}\lceil\hat{s}_{\text{key}} \mapsto \hat{d}_{\text{value}}\rceil & \widehat{size}(\gamma(\hat{s}_{\text{key}})) = 1\\ \hat{o} \sqcup\!\!\sqcup [\hat{s}_{\text{key}} \mapsto \hat{d}_{\text{value}}] & \text{otherwise}\end{cases}\\
\hat{\omega}' &= \begin{cases} \hat{\omega}[\hat{\ell} \mapsto \hat{o}'] & \hat{\omega}^\circ(\hat{\ell}) = 1\\ \hat{\omega} \sqcup [\hat{\ell} \mapsto \hat{o}'] & \text{otherwise.}\end{cases}\\
\hat{o}^T &= \hat{\omega}^T(\hat{\ell})\\
\hat{d}^T &= \hat{\mathcal{A}}^T(æ_{\text{value}}, \hat{\rho}, \hat{\sigma}^T, \hat{\omega}^T)\\
\hat{o}^{T'} &= \begin{cases} \hat{o}^T\lceil\hat{s}_{\text{key}} \mapsto \hat{d}^T\rceil & \widehat{size}(\gamma(\hat{s}_{\text{key}})) = 1\\ \hat{o}^T \sqcup\!\!\sqcup [\hat{s}_{\text{key}} \mapsto \hat{d}^T] & \text{otherwise}\end{cases}\\
\hat{\omega}^{T'} &= \begin{cases} \hat{\omega}^T[\hat{\ell} \mapsto \hat{o}^{T'}] & \hat{\omega}^\circ(\hat{\ell}) = 1\\ \hat{\omega}^T \sqcup [\hat{\ell} \mapsto \hat{o}^{T'}] & \text{otherwise.}\end{cases}
\end{aligned}$$

Strong deletion The same information that powers strong update also guides "strong deletion." When removing a field $\hat{f}$ from an abstract object at location $\hat{\ell}$, *if* the count of $\hat{\ell}$ is one, and the cardinality of $\alpha^{-1}(\hat{f})$ is also 1, then it is sound to delete the field from the object. If either of these conditions is violated, then this object or this field represents more than one concrete field, and removing it could be unsound. Thus, abstract deletion, by default, simply leaves the object and its field unmodified.

5. Implementation and evaluation

We have implemented the analytical framework HFTA for λ_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 *PyLexer* 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`.

Prog	wfact	wfib	cc	conv	srch
$\texttt{LOC}_{py}$	47	50	73	61	260
$\texttt{LOC}_{\lambda_H}$	596	631	943	574	1746
Time(sec)($k = 1$)	0.322	0.661	0.293	0.078	14.2
$\texttt{metric}\|^{\|1\|}_{\|\texttt{terms}\|}$	69%	56%	68%	57%	55%
# States	364	573	348	160	3018
XSS	3	1	3	1	1

Table 1. Preliminary Results of HFTA with $k = 1$ and cardinality analysis

In addition, we also implemented a set of built-in classes that are preloaded by the translator, including `object`, `list`, `tuple`, `set`, `dict`; meta operations, such as `setattr`, `getattr` and meta attributes such as `__setattr__`, `__getattr__`.

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 [14].

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 (`wfact`), Fibonacci (`wfib`), color chooser (`cc`) and a converter (`conv`). `srch` is an open source web application in Python [9], which is used to search a specific web site for the page users are looking for. We have deliberately enriched the Python search program for better abstraction and mimicked the architecture and coding pattern of other popular web applications in Python, like Pybloxom [1] and MoinMoin [22].

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

The $\texttt{metric}|^{|i|}_{|\texttt{objects}|}$ 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 $|\texttt{objects}|$, where $i = 1, 2, ...$ is the number of abstract location (objects) with a cardinality of i, and $|\texttt{objects}|$ 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 $\texttt{metric}|^{|i|}_{|\texttt{objects}|}$ indicates reasonable precision even for five preliminary analysis.

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. [2].

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.

```
...

class Search():
    def __init__(self, input):
        self.resp = Response(Request(input))
    def do_get(self):
        d = self.resp.req.getData()
        self.resp.result = search(d)
        self.resp.gen_body()
    def render(self):
        write(self.resp.body)

class Request():
    def __init__(self, data):
        self.data = data
        self.handlers = {"santinize_callback": santinize_proc,
                         "...": "..."}
    def getData(self):
        return self.data

class Response():
    def __init__(self, req):
        self.result = []
        self.body = ""
        self.req = req
    def gen_body(self):
        len_res = len(self.result)
        tmp = "<div id= \"search_results\">\n<ol>)\n"
        if len_res == 0:
            self.body = tmp + "<h3> No results.</h3>"
                            + self.req.getData()
        indx = 0
        while indx < len_res:
            self.body = self.body + "<li <a href=\"test/"
                      + self.result[indx][1]+"?search=true&term="
                      + str(self.req.getData()) +"\">\n"
                      + self.result[indx][0] + "</a>\n"
            indx = indx + 1
        self.body = self.body + "</ol>\n</div>\n"

me1 = Search(s1)
me1.do_get()
me1.render()

me2 = Search(s2)
me2.resp.req.handlers[action + "_callback"](me2.resp.req.data)
me2.do_get()
me2.render()
```

Figure 10. Simple search engine vulnerabilities (simplified)

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

[2] The analyzer is implemented in functional scripting language – Racket, of which the analysis time can be slower than implementation in some other languages,such as C/C++ or Scala.

Prog	$k = 0$	$k = 1$
wfib	0.529	0.661
wfact	0.268	0.322
cc	0.244	0.293
conv	0.072	0.078
srch	12	14.2

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

Prog	With-Cardi	No-Cardi
webfib	56%	44%
webfact	69%	53%
cc	68%	55%
conv	57%	57%
srch	55%	–

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 into λ_H. For example, in our experimental prototype for Python, all of Python's metaprogramming features (reflection) are compiled into hash maps in λ_H.

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 analyzer 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 carve 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

[1] Pyblosxom. `http://pyblosxom.bluesock.org/`.

[2] ANDREWS, M. Guest editor's introduction: The state of web security. *IEEE Security and Privacy 4* (July 2006), 14–15.

[3] CHANG, W., STREIFF, B., AND LIN, C. Efficient and extensible security enforcement using dynamic data flow analysis. In *Conference on Computer and Communications Security* (2008).

[4] CHASE, D. R., WEGMAN, M., AND ZADECK, F. K. Analysis of pointers and structures. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1990), PLDI '90, ACM, pp. 296–310.

[5] DENNING, D. E. A lattice model of secure information flow. *Commun. ACM 19*, 5 (May 1976), 236–243.

[6] FELLEISEN, M. *The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Indiana University, 1987.

[7] FELLEISEN, M., AND FRIEDMAN, D. P. Control operators, the SECD-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts* (Aug. 1986).

[8] FLANAGAN, C., SABRY, A., DUBA, B. F., AND FELLEISEN, M. The essence of compiling with continuations. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (New York, NY, USA, June 1993), ACM, pp. 237–247.

[9] GROSSBART, Z. Search engine in python, 2007. `http://www.zackgrossbart.com/hackito/search-engine-python/`.

[10] HUANG, Y. W., YU, F., HANG, C., TSAI, C. H., LEE, D. T., AND KUO, S. Y. Securing web application code by static analysis and runtime protection. In *WWW '04: Proceedings of the 13th international conference on World Wide Web* (New York, NY, USA, 2004), ACM, pp. 40–52.

[11] JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IN 2006 IEEE SYMPOSIUM ON SECURITY AND PRIVACY* (2006), pp. 258–263.

[12] KOZLOV, D., AND PETUKHOV. Implementation of tainted mode approach to finding security vulnerabilities for python technology.

[13] MEYEROVICH, L. A., AND LIVSHITS, B. ConScript: Specifying and enforcing Fine-Grained security policies for JavaScript in the browser. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2010), SP '10, IEEE Computer Society, pp. 481–496.

[14] MIGHT, M. A-normalization: Why and how. `http://matt.might.net/articles/a-normalization/`.

[15] MIGHT, M. You don't understand exceptions, but you should. `http://matt.might.net/articles/implementing-exceptions/`.

[16] MIGHT, M. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, June 2007.

[17] MIGHT, M. Shape analysis in the absence of pointers and structure. In *VMCAI 2010: International Conference on Verification, Model-Checking and Abstract Interpretation* (Madrid, Spain, Jan. 2010), pp. 263–278.

[18] MIGHT, M., AND SHIVERS, O. Improving flow analyses via Gamma-CFA: Abstract garbage collection and counting. In *ICFP '06: Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2006), ACM, pp. 13–25.

[19] MIGHT, M., AND SHIVERS, O. Exploiting reachability and cardinality in higher-order flow analysis. *Journal of Functional Programming 18*, Special Double Issue 5-6 (2008), 821–864.

[20] MIGHT, M., SMARAGDAKIS, Y., AND VAN HORN, D. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In *PLDI '10: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2010), PLDI '10, ACM Press, pp. 305–315.

[21] MINAMIDE, Y. Static approximation of dynamically generated web pages. In *WWW*. 2005, pp. 432–441.

[22] Moin moin. `http://moinmo.in/`.

[23] NGUYEN-TUONG, A., GUARNIERI, S., GREENE, D., SHIRLEY, J., AND EVANS, D. Automatically hardening web applications using precise tainting. In *In 20th IFIP International Information Security Conference* (2005), pp. 372–382.

[24] OWASP. OWASP top 10 for 2010. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`.

[25] RICHARDS, G., LEBRESNE, S., BURG, B., AND VITEK, J. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2010), PLDI '10, ACM, pp. 1–12.

[26] Ruby. `http://ruby-doc.org/docs/ProgrammingRuby/`.

[27] SABELFELD, A., AND MYERS, A. Language-Based Information-Flow security, 2003.

[28] SALIB, M. Static Type Inference with Starkiller. In *PyCon DC* (2004).

[29] SEO, J., AND MONICA. InvisiType: Object-Oriented security policies. In *NDSS* (2010).

[30] TRIPP, O., PISTOIA, M., FINK, S. J., SRIDHARAN, M., AND WEISMAN, O. TAJ: effective taint analysis of web applications. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2009), PLDI '09, ACM, pp. 87–97.

[31] VAN HORN, D., AND MIGHT, M. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 2010), ICFP '10, ACM, pp. 51–62.

[32] VOGT, P., NENTWICH, F., JOVANOVIC, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Cross site scripting prevention with dynamic data tainting and static analysis.

[33] WALL, L., CHRISTIANSEN, T., SCHWARTZ, R. L., AND POTTER, S. *Programming Perl (2nd Edition)*.

[34] WASSERMANN, G., AND SU, Z. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2007), ACM, pp. 32–41.

[35] WASSERMANN, G., AND SU, Z. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th international conference on Software engineering* (Leipzig, Germany, 2008), ACM, pp. 171–180.