Type-based Dependency Analysis for JavaScript

Matthias Keil Peter Thiemann

Institute for Computer Science
University of Freiburg
Freiburg, Germany
{keilr,thiemann}@informatik.uni-freiburg.de

Abstract

Dependency analysis is a program analysis that determines potential data flow between program points. While it is not a security analysis per se, it is a viable basis for investigating data integrity, for ensuring confidentiality, and for guaranteeing sanitization. A noninterference property can be stated and proved for the dependency analysis.

We have designed and implemented a dependency analysis for JavaScript. We formalize this analysis as an abstraction of a tainting semantics. We prove the correctness of the tainting semantics, the soundness of the abstraction, a noninterference property, and the termination of the analysis.

Categories and Subject Descriptors F.3.2 [*LOGICS AND MEANINGS OF PROGRAMS*]: Semantics of Programming Languages—Program analysis; D.3.1 [*PROGRAMMING LANGUAGES*]: Formal Definitions and Theory—Semantics ; D.4.6 [*OPERATING SYSTEMS*]: Security and Protection—Information flow controls

General Terms Security

Keywords Type-based Analysis, Dependency, JavaScript

1. Introduction

Security Engineering is one of the challenges of modern software development. The connected world we live in consists of interacting entities that process distributed private data. This data has to be protected against illegal usage, tampering, and theft.

Web applications are one popular example of such interacting entities. They run in the web browser and consist of program fragments from different sources (e.g., mashups). Such fragments should not be entrusted with sensitive information. However, if a fragment's input data can be shown not to depend on confidential data, then it cannot divulge this data or tamper with it.

A Web application may also be vulnerable to an injection attack. Such an attack arises when data is stored in a database or in the DOM without proper escaping. If an analysis can determine that the input to the database never depends directly on a data source (like an HTML input field), but rather is always filtered by a suitable sanitizer, then many kinds of injection attacks can be avoided.

Dependency analysis is a program analysis that can help in both situations, because it determines potential data flow between program points. Intuitively, there is a dependency between the value in variable x and the value of an expression $e[x]$ containing the variable if substituting different expressions e' for x may change the value of $e[e']$. In our application we label data sources (e.g., as confidential) in a JavaScript program and are interested in identifying the potential sinks reachable from these data sources. For sanitization, we instrument sanitizers with a relabeling operation that modifies the dependency on the original data source to a sanitized dependency. We consider a data sink safe, if it only depends on data that passed through a sanitizer. Other uses of dependency information for optimization or parallelization are possible, but not considered in this work.

We designed and implemented our dependency analysis as an extension of TAJS [15], a type analyzer for JavaScript. The implementation allows us to label data sources with a $\mathbf{trace}^{\ell}$ marker and to indicate relabelings with an $\mathbf{untrace}^{(\mathcal{A}\hookrightarrow\mathcal{A}')}$ marker. The analysis performs an abstract interpretation to approximate the flow of the markers throughout the program. The marker is part of the analyzed type and propagated to all program points that depend on a marked value.

Part of our work consists in establishing the formal underpinnings of the implemented analysis. Thus, we outline a correctness proof for the dependency part of the analysis. For conducting the proofs, we have simplified the domains with respect to the implementation to avoid an overly complex formal system. To this end, we formalize the dynamic semantics of a JavaScript core language, extend that with marker propagation, and then formalize the abstract inter-

PLAS'13, June 20, 2013, Seattle, WA, USA.

pretation of this extended semantics. Both, concrete and abstract semantics are given as big-step semantics. We prove sound marker propagation, sound approximation of the dynamic semantics by its abstract counterpart, noninterference, and the termination of the analysis.

Contributions

- Design and implementation of a type-based dependency analysis based on TAJS [15].
- Formalization of the analysis.
- Proofs of correctness and termination.
- Extension of the analysis for sanitization (Section 7).
- A noninterference theorem (Section 8).

Overview Section 2 considers some example scenarios of our implemented system. Section 3 formalizes a core language, its dynamic semantics, and defines noninterference semantically. Section 4 extends this semantics with tainting. Section 5 defines the corresponding abstraction, Section 6 gives some example applications, and Section 7 defines the extension for sanitization. Section 8 contains our theorems of soundness, noninterference, and termination. Section 9 briefly describes the implementation. Related work is discussed in Section 10 followed by a conclusion.

A technical report [18] is available with all technical details including the proofs.

2. Application Scenario

Web developers rely on third-party libraries for calendars, maps, social networks, and so on. To create a trustworthy application, they should ensure that these libraries do not leak sensitive information of their users.

One way to avoid such leaks is to detect information flow from confidential data sources to untrusted sinks by program analysis and take measures to avoid this flow. Sometimes, this approach is too restrictive, because the data arriving at the sink has been sanitized on the way from the source. Sanitization can take many forms: data may have been encrypted or a username/password combination may have been reduced to a boolean. In such cases, the resulting data still depends on the confidential source, but it can be safely declassified and passed on to an untrusted sink.

An analogous scenario is the avoidance of injection attacks where direct dependencies of database queries or DOM contents from input fields in a Web form should be avoided. However, an indirect dependency via a sanitizer that, in this case, escapes the values suitably is acceptable.

Our dependency analysis addresses both scenarios as illustrated with the following examples.

2.1 Cookies

A web developer might want to ensure that the code does not read sensitive data from cookies and sends it to the net. Technically, it means that data that is passed to network send operations must not depend on document.cookie. This dependency can be checked by our analysis.

```
1  var userHandler = function(uid) {
2     var userData = {name:''};
3     var onSuccess = function(response) {
4        userData = response;
5     };
6
7     if (Cookie.isset(uid)) {
8        Cookie.request(uid, onSuccess);
9     } else {
10       Ajax.request('http:\\example.org', {
11          content : uid
12       }, onSuccess);
13    }
14
15    return {
16       getName : function() {
17          return userData.name;
18       }
19    }
20 };
21 var name1 = userHandler(trace("uid1")).getName();
22 var name2 = userHandler(trace("uid2")).getName();
```

Figure 1. Loading sensitive data.

To label data sources, our implementation reads a configuration file with a list of JavaScript objects that are labeled with a dependency mark before starting the analysis. Any predefined value or function can be marked in this way. For this example, the analyzer is to label document.cookie with t0.

Values that are written to a cookie are labeled by wrapping them in a trace expression. The analysis determines that values returned from cookies are influenced by document.cookie. Furthermore, after writing a marked value to a cookie, each subsequent read operation returns a value that depends on it.

The following code snippet uses a standard library for reading and writing cookies. The comments show the analyzed dependencies of the respective values.

```
1 var val1 = readCookie('test'); // d(val1)={t0}
2 var val2 = trace(4711);        // d(val2)={t1}
3 writeCookie('test', val2);
4 var val3 = readCookie('test'); // d(val3)={t0,t1}
```

Thus, the read value in val1 is influenced by document.cookie. The value in val2 is labeled by a fresh mark t1. Later, this value is written to the cookie. Hence, the result val3 of the last read operation is influenced by document.cookie and val2.

2.2 Application: Sensitive Data

The next example is to illustrate the underpinnings of our analysis and to point out differences to other techniques. Figure 1 shows a code fragment to request the user name corresponding to a user id. This data is either read from a cookie or obtained by an Ajax request.

The function userHandler returns an interface to a user's personal data. The implementation abstracts from the data source by using a callback function onSuccess to handle the results. The code ignores the problem that userData may not be valid before completion of the Ajax request.

To detect all values depending on user information, a developer would mark the id. This mark should propagate to

```
1  loadForeignCode = trace(function() {
2     Array.prototype.foreach = function(callback) {
3        for ( var k = 0; k < this.length; k++) {
4           callback(k, this[k]);
5        }
6     };
7  });
8  loadForeignCode();
9  // [..]
10 var array = new Array(4711, 4712);
11 array.foreach(function(k, v) {
12    result = k + v;
13 });
```

Figure 2. Using foreign Code.

values returned from Cookie.request() and Ajax.request(). Because we are interested in values depending on Cookie.request() and Ajax.request() the interfaces also get marked.

The conditional in line 7 depends on Cookie.isset(uid) and thus on uid and on the cookie interface. The value in userData (line 4) depends on uid, on the cookie interface, and on Cookie.isset(uid) from the Ajax interface. The result name1 depends on userData.name and therefore on the user id, the cookie interface, and potentially on the Ajax interface.

Standard security analyses label values with marks drawn from a security lattice, often just *Low* and *High*. If both sources, the cookie interface and the Ajax interface, are labeled with the same mark, there is no way to distinguish these sources. Dependencies allow us a to formulate security properties on a fine level of granularity that distinguishes different sources without changing the underlying lattice.

Second, our analysis is flow-sensitive. Dependencies are bound to values instead of variable names or parameters. A variable may containt different values depending on different sources during evaluation. In addition, the underlying TAJS implementation already handles aliasing and polyvariant analysis in a satisfactory way.

In the example, the values in name1 and name2 result from the same function but may depend on different sources. The flow-sensitive model retains the independence of the value in name1 and trace("uid2"). Section 6.1 discusses the actual outcome of the analysis.

2.3 Application: Foreign Code

The second scenario (Figure 2) illustrates one way a library can extend existing functionality. This example extends the prototype of Array by a foreach function. Later on, this function is used to iterate over elements.

The goal here is to protect code from being compromised by the libraries used. The function loadForeignCode encapsulates foreign code and is labeled as a source. In consequence, all values created or modified by calling loadForeignCode depend on this function and contain its mark. Because the function in the foreach property gets marked, the values in result also get marked. Therefore, result may be influenced by loading foreign code. See Section 6.2 for the results of the analysis.

```
1  $ = function(id) {
2    return trace(document.getElementById(id).value, "#DOM");
3  }
4  function sanitizer(value) {
5    /* clean up value ... */
6    return untrace(value, "#DOM");
7  }
8  // [...]
9  var input = $("text");
10 var secureInput = sanitizer(input);
11 consumer(secureInput);
```

Figure 3. Analyzing sanitization.

2.4 Application: Sanitization

Noninterference is not the only interesting property that can be investigated with the dependency analyzer. To avoid injection attacks, programmers should ensure that only escaped values occur in a database query or become part of an HTML page. Also, a dependency on a secret data source may be acceptable if the data is encrypted before being published. These examples illustrate the general idea of sanitization where a suitable function needs to be interposed in the dataflow between certain sources and sinks.

The concrete example in Figure 3 applies our analysis to the problem. The input is labeled with mark #DOM (line 2). The function in line 4 performs some (unspecified) sanitization and finally applies the untrace function to mark the dependency on the marks identified with #DOM as a sanitized, safe dependency. The argument of the consumer can now be checked for dependencies on unsanitized values. In the example code, the analysis determines that the argument depends on the DOM, but that the dependency is sanitized.

Changing line 10 as indicated below leaves the argument of the consumer with a mixture of sanitized and unsanitized dependencies. This mixture could be flagged as an error.

```
var secureInput =
  i_know_what_i_do ? sanitizer(input) : input;
```

3. Formalization

This section presents the JavaScript core calculus λ_J along with a semantic definition of independence.

3.1 Syntax of λ_J

λ_J is inspired by JavaScript core calculi from the literature [11, 14]. A λ_J expression (Figure 4) is either a constant c (a boolean, a number, a string, **undefined**, or **null**), a variable x, a lambda expression, an application, a primitive operation, a conditional, an object creation, a property reference, a property assignment, or a trace expression.

The trace expression is novel to our calculus. It creates marked values that can be tracked by our dependency analysis. The expression $\mathbf{new}^{\ell}\ e$ creates an object whose prototype is the result of e. The lambda expression, the new expression, and the trace expression carry a unique mark ℓ.

$$
\begin{array}{rcl}
e & ::= & c \mid x \mid \lambda^\ell x.e \mid e(e) \mid \mathbf{op}(e,e) \\
& \mid & \mathbf{if}\ (e)\ e,\ e \mid \mathbf{new}^\ell\ e \mid e[e] \mid e[e] = e \mid \mathbf{trace}^\ell(e)
\end{array}
$$

$$
\begin{array}{llll}
\textit{Location} & \ni\ \xi^\ell & & \\
\textit{Value} & \ni\ v & ::= & c \mid \xi^\ell \\
\textit{Prototype} & \ni\ p & ::= & v \\
\textit{Closure} & \ni\ f & ::= & \emptyset \mid \langle \rho, \lambda^\ell x.e \rangle \\
\textit{Object} & \ni\ o & ::= & \emptyset \mid o[\textit{str} \mapsto v] \\
\textit{Storable} & \ni\ s & ::= & \langle o, f, p \rangle \\
\textit{Environment} & \ni\ \rho & ::= & \emptyset \mid \rho[x \mapsto v] \\
\textit{Heap} & \ni\ \mathcal{H} & ::= & \emptyset \mid \mathcal{H}[\xi^\ell \mapsto s]
\end{array}
$$

Figure 4. Syntax and semantic domains of λ_J.

$$
\begin{array}{lcl}
\langle o, f, p \rangle(\textit{str}) & ::= & \begin{cases} v, & o = o'[\textit{str} \to v] \\ o'(\textit{str}), & o = o'[\textit{str}' \to v] \\ \mathcal{H}(\xi^\ell)(\textit{str}), & o = \emptyset\ \wedge\ p = \xi^\ell \\ \textbf{undefined}, & o = \emptyset\ \wedge\ p = c \end{cases} \\
\langle o, f, p \rangle[\textit{str} \mapsto v] & ::= & \langle o[\textit{str} \mapsto v], f, p \rangle \\
\langle o, f, p \rangle_f & ::= & f \\
\mathcal{H}[\xi^\ell, \textit{str} \mapsto v] & ::= & \mathcal{H}[\xi^\ell \mapsto \mathcal{H}(\xi^\ell)[\textit{str} \mapsto v]] \\
\mathcal{H}[\xi^\ell \mapsto \emptyset] & ::= & \mathcal{H}[\xi^\ell \mapsto \langle \emptyset, \emptyset, \textbf{null} \rangle] \\
\mathcal{H}[\xi^\ell \mapsto o] & ::= & \mathcal{H}[\xi^\ell \mapsto \langle o, \emptyset, \textbf{null} \rangle] \\
\mathcal{H}[\xi^\ell \mapsto f] & ::= & \mathcal{H}[\xi^\ell \mapsto \langle \emptyset, f, \textbf{null} \rangle] \\
\mathcal{H}[\xi^\ell \mapsto p] & ::= & \mathcal{H}[\xi^\ell \mapsto \langle \emptyset, \emptyset, p \rangle]
\end{array}
$$

Figure 5. Abbreviations.

3.2 Semantic domains

Figure 4 also defines the semantic domains of λ_J. A heap maps a location ξ^ℓ to a storable s, which is a triple consisting of an object o, potentially a function closure f (only for function objects), and a value p, which serves as the prototype. The superscript ℓ refers the expression causing the allocation. An object o maps a string to a value. A closure consists of an environment ρ and an expression e. The environment ρ maps a variable to a value v, which may be a base type constant or a location.

Program execution is modeled by a big-step evaluation judgment of the form $\mathcal{H}, \rho \vdash e \Downarrow \mathcal{H}' \mid v$. The evaluation of expression e in an initial heap $\mathcal{H}$ and environment ρ results in the final heap $\mathcal{H}'$ and the value v. We omit its standard definition for space reasons, but show excerpts of an augmented semantics in Section 4.

Figure 5 introduces some abbreviated notation. A property lookup or a property update on a storable $s = \langle o, f, p \rangle$ is relayed to the underlying object. The property access $s(\textit{str})$ returns **undefined** by default if the accessed string is not defined in o and the prototype of s is not a location ξ^ℓ. We write s_f for the closure in s. The notation $\mathcal{H}[\xi^\ell, \textit{str} \mapsto v]$ updates a property of storable $\mathcal{H}(\xi^\ell)$, $\mathcal{H}[\xi^\ell \mapsto o]$ initializes an object, and $\mathcal{H}[\xi^\ell \mapsto f]$ defines a function.

3.3 Independence

The $\mathbf{trace}^\ell$ expression serves to mark a program point as a source of sensitive data. An expression e is independent from that source if the value of the $\mathbf{trace}^\ell$ expression does not influence the final result of e. The first definition formalizes replacing the argument of a $\mathbf{trace}^\ell$ expression.

Definition 1. *The substitution $e[\ell \mapsto \tilde{e}]$ of ℓ in e by $\tilde{e}$ is defined as the homomorphic extension of*

$$
\textbf{\textit{trace}}^\ell(e')[\ell \mapsto \tilde{e}] \equiv \textbf{\textit{trace}}^\ell(\tilde{e}) \tag{1}
$$

Definition 2 (incomplete first attempt)**.** *The expression e is independent from ℓ iff all possible substitutions of ℓ are unobservable.*

$$
\begin{array}{rl}
\forall e_1, e_2 : & \mathcal{H}, \rho \vdash e[\ell \mapsto e_1] \Downarrow \mathcal{H}_1 \mid v \\
\leftrightarrow & \mathcal{H}, \rho \vdash e[\ell \mapsto e_2] \Downarrow \mathcal{H}_2 \mid v
\end{array} \tag{2}
$$

This definition covers both, the terminating and the non-terminating cases. Furthermore, we consider direct dependencies, indirect dependencies, and transitive dependencies, similar to the behavior described by Denning [8, 9]. In Section 8, we complete this definition to make it amenable to proof.

4. Dependency Tracking Semantics

To attach marker propagation for upcoming values we apply definition 2 to the λ_J calculus. The later on derived abstract interpretation is formalized on this extended calculus.

This section extends the semantics of λ_J with mark propagation. The resulting calculus $\lambda_J^{\mathcal{D}}$ *only* provides a baseline calculus for subsequent static analysis. $\lambda_J^{\mathcal{D}}$ is *specifically not* meant to perform any kind of dynamic analysis, where the presence or absence of a mark in a value guarantees some dependency related property.

The calculus extends *Value* to *Tainted Value* $\ni\ \omega ::= v : \kappa$ where $\kappa ::= \emptyset \mid \ell \mid \kappa \bullet \kappa$ is a dependency annotation. *Tainted Value* replaces *Value* in objects and environments. The operation $\bullet$ joins two dependencies. If $\omega = v : \kappa_v$ then write $\omega \bullet \kappa$ for $v : \kappa_v \bullet \kappa$ to apply a dependency annotation to a value.

The big-step evaluation judgment $\mathcal{H}, \rho, \kappa \vdash e \Downarrow \mathcal{H}' \mid \omega$ for $\lambda_J^{\mathcal{D}}$ extends the one for λ_J by a new component κ which tracks the context dependency for expression e. Figure 6 contains its defining inference rules.

The evaluation rules (DT-CONST), (DT-VAR), and (DT-ABS) are trivial. Their return values depend on the context. (DT-OP) calculates the result on the value part and combines the dependencies of the involved values. $\Downarrow^v_{\mathbf{op}}$ stands for the application of operator **op**. The rule (DT-NEW) binds the dependency of the evaluated prototype to the returned location. During (DT-APP) the dependency of the value referencing the function is bound to the sub-context. In a similar

(DT-CONST)
$$\frac{}{\mathcal{H}, \rho, \kappa \vdash c \Downarrow \mathcal{H} \mid c : \kappa}$$

(DT-VAR)
$$\frac{}{\mathcal{H}, \rho, \kappa \vdash x \Downarrow \mathcal{H} \mid \rho(x) \bullet \kappa}$$

(DT-ABS)
$$\frac{\xi^\ell \notin dom(\mathcal{H})}{\mathcal{H}, \rho, \kappa \vdash \lambda^\ell x.e \Downarrow \mathcal{H}[\xi^\ell \mapsto \langle \rho, \lambda^\ell x.e \rangle] \mid \xi^\ell : \kappa}$$

(DT-OP)
$$\frac{\mathcal{H}, \rho, \kappa \vdash e_0 \Downarrow \mathcal{H}' \mid v_0 : \kappa_0 \quad \mathcal{H}', \rho, \kappa \vdash e_1 \Downarrow \mathcal{H}'' \mid v_1 : \kappa_1 \quad v_{op} = \Downarrow^v_{\mathbf{op}} (v_0, v_1)}{\mathcal{H}, \rho, \kappa \vdash \mathbf{op}(e_0, e_1) \Downarrow \mathcal{H}'' \mid v_{op} : \kappa_0 \bullet \kappa_1}$$

(DT-NEW)
$$\frac{\mathcal{H}, \rho, \kappa \vdash e \Downarrow \mathcal{H}' \mid v : \kappa_v \qquad \xi^\ell \notin dom(\mathcal{H})}{\mathcal{H}, \rho, \kappa \vdash \mathbf{new}^\ell\ e \Downarrow \mathcal{H}'[\xi^\ell \mapsto v] \mid \xi^\ell : \kappa_v}$$

(DT-APP)
$$\frac{\mathcal{H}, \rho, \kappa \vdash e_0 \Downarrow \mathcal{H}' \mid \xi^\ell : \kappa_0 \quad \langle o, \langle \dot\rho, \lambda^\ell x.e \rangle, p \rangle = \mathcal{H}'(\xi^\ell) \quad \mathcal{H}', \rho, \kappa \vdash e_1 \Downarrow \mathcal{H}'' \mid v_1 : \kappa_1 \quad \mathcal{H}'', \dot\rho[x \mapsto v_1 : \kappa_1], \kappa \bullet \kappa_0 \vdash e \Downarrow \mathcal{H}''' \mid v : \kappa_v}{\mathcal{H}, \rho, \kappa \vdash e_0(e_1) \Downarrow \mathcal{H}''' \mid v : \kappa_v}$$

(DT-IFTRUE)
$$\frac{\mathcal{H}, \rho, \kappa \vdash e_0 \Downarrow \mathcal{H}' \mid v_0 : \kappa_0 \quad v_0 = true \qquad \mathcal{H}', \rho, \kappa \bullet \kappa_0 \vdash e_1 \Downarrow \mathcal{H}''_1 \mid v_1 : \kappa_1}{\mathcal{H}, \rho, \kappa \vdash \mathbf{if}\ (e_0)\ e_1,\ e_2 \Downarrow \mathcal{H}''_1 \mid v_1 : \kappa_1}$$

(DT-IFFALSE)
$$\frac{\mathcal{H}, \rho, \kappa \vdash e_0 \Downarrow \mathcal{H}' \mid v_0 : \kappa_0 \quad v_0 \neq true \qquad \mathcal{H}', \rho, \kappa \bullet \kappa_0 \vdash e_2 \Downarrow \mathcal{H}''_2 \mid v_2 : \kappa_2}{\mathcal{H}, \rho, \kappa \vdash \mathbf{if}\ (e_0)\ e_1,\ e_2 \Downarrow \mathcal{H}''_2 \mid v_2 : \kappa_2}$$

(DT-GET)
$$\frac{\mathcal{H}, \rho, \kappa \vdash e_0 \Downarrow \mathcal{H}' \mid \xi^\ell : \kappa_{\xi^\ell} \quad \mathcal{H}', \rho, \kappa \vdash e_1 \Downarrow \mathcal{H}'' \mid \mathit{str} : \kappa_{\mathit{str}}}{\mathcal{H}, \rho, \kappa \vdash e_0[e_1] \Downarrow \mathcal{H}'' \mid \mathcal{H}''(\xi^\ell)(\mathit{str}) \bullet \kappa_{\xi^\ell} \bullet \kappa_{\mathit{str}}}$$

(DT-PUT)
$$\frac{\mathcal{H}, \rho, \kappa \vdash e_0 \Downarrow \mathcal{H}' \mid \xi^\ell : \kappa_{\xi^\ell} \quad \mathcal{H}', \rho, \kappa \vdash e_1 \Downarrow \mathcal{H}'' \mid \mathit{str} : \kappa_{\mathit{str}} \quad \mathcal{H}'', \rho, \kappa \vdash e_2 \Downarrow \mathcal{H}''' \mid v : \kappa_v \quad \mathcal{H}'''' = \mathcal{H}'''[\xi^\ell, \mathit{str} \mapsto v : \kappa_v \bullet \kappa_{\xi^\ell} \bullet \kappa_{\mathit{str}}]}{\mathcal{H}, \rho, \kappa \vdash e_0[e_1] = e_2 \Downarrow \mathcal{H}'''' \mid v : \kappa_v}$$

(DT-TRACE)
$$\frac{\mathcal{H}, \rho, \kappa \bullet \ell \vdash e \Downarrow \mathcal{H}' \mid v : \kappa_v}{\mathcal{H}, \rho, \kappa \vdash \mathbf{trace}^\ell\ (e) \Downarrow \mathcal{H}' \mid v : \kappa_v}$$

Figure 6. Inference rules of $\lambda^{\mathcal{D}}_J$.

Undefined		::=	$\wp(\{\texttt{undefined}\})$
Null		::=	$\wp(\{\texttt{null}\})$
Bool		::=	$\wp(\{\texttt{true}, \texttt{false}\})$
Num		::=	$\mathit{NUM}^\top_\bot$
String		::=	$\mathit{STRING}^\top_\bot$
Lattice Value	$\ni \mathcal{L}$	::=	*Undefined* $\times$ *Null* $\times$ *Bool* $\times$ *Num* $\times$ *String*

Figure 7. Base Type Value Lattice.

Label	$\ni \Xi$	::=	$\{\ell \dots\}$
Abstract Closure	$\ni \Lambda^\ell$	::=	$\langle \sigma, \lambda^\ell x.e \rangle$
Abstract Object	$\ni \Delta$	::=	$\emptyset \mid \Delta[\mathcal{L} \mapsto \vartheta]$
Abstract Value	$\ni \vartheta$	::=	$\langle \mathcal{L}, \Xi, \mathcal{D} \rangle$
Abstract Storable	$\ni \theta$	::=	$\langle \Delta, \Lambda^\ell, \Xi \rangle$
FunctionStore	$\ni \mathcal{F}$	::=	$\emptyset \mid \mathcal{F}[\ell \mapsto \langle \Gamma, \vartheta, \Gamma, \vartheta \rangle]$
Scope	$\ni \sigma$	::=	$\emptyset \mid \sigma[x \mapsto \vartheta]$
ObjectStore	$\ni \Sigma$	::=	$\emptyset \mid \Sigma[\ell \mapsto \theta]$
State	$\ni \Gamma$	::=	$\langle \Sigma, \mathcal{D} \rangle$
Dependency	$\ni \mathcal{D}$	::=	$\emptyset \mid \ell \mid \mathcal{D} \sqcup \mathcal{D}$

Figure 8. Abstract Semantic Domains.

way (DT-IFTRUE) and (DT-IFFALSE) bind the dependency of the condition to the sub-context. The rule (DT-GET) combines the dependencies of heap location and property reference to the returned value. The rule (DT-PUT) combines these dependencies to the assigned value because the evaluated location and property references affect the write operation and further the value which is accessible at this location.

The trace expression $\mathbf{trace}^\ell\ (e)$ (DT-TRACE) adds the ℓ annotation to the context of expression e. This addition causes all values created or modified in e to be marked with ℓ (e.g. to detect side effects) as stated by the following context dependency lemma.

Lemma 1. *$\mathcal{H}, \rho, \kappa \vdash e \Downarrow \mathcal{H}' \mid v : \kappa_v$ implies that $\kappa \subseteq \kappa_v$.*

The proof is by induction on the relation $\Downarrow$.

5. Abstract Analysis

The analysis is an abstraction of the $\lambda^{\mathcal{D}}_J$ calculus. Its basis is the lattice for base type values (Figure 7), which is a simplified adaptation of the lattice of TAJS [15]. *NUM* is the set of floating point numbers, *STRING* the set of string literals, and the annotation $\cdot^\top_\bot$ turns a set into a flat lattice by adding a bottom and top element. An element of the analysis lattice is a tuple like $\langle \bot, \bot, \texttt{true}, \bot, \texttt{"x"} \rangle$ which represents a value which is either the boolean value `true` or the string `"x"`. Further, $\langle \bot, \bot, \bot, \top, \bot \rangle$ represents all possible number values. The abstract semantic domains (Figure 8) are similar to the domains arising from the $\lambda^{\mathcal{D}}_J$ calculus except that a

set of marks Ξ abstracts a set of concrete locations ξ^ℓ where $\ell \in \Xi$. An abstract value $\vartheta = \langle \mathcal{L}, \Xi, \mathcal{D} \rangle$ is a triple of a lattice element $\mathcal{L}$, object marks Ξ, and dependency $\mathcal{D}$.

Hence, each abstract value represents a set of base type values and a set of objects. We write $\mathcal{L}_\vartheta$ for the analysis lattice, Ξ_ϑ for the marks, and $\mathcal{D}_\vartheta$ for the dependency component of the abstract value ϑ. Each abstract object is identified by the mark ℓ corresponding to the **new**$^\ell$ e expression creating the object. An abstract storable consists of an abstract object, a function closure, and a set of locations representing the prototype. Unlike before, the abstract object maps a lattice element to a value. This mechanism reduces the number of merge operations during the abstract analysis.

The abstract state is a pair $\Gamma = \langle \Sigma, \mathcal{D} \rangle$ where Σ is the mapping from marks to abstract storables. We write Σ_Γ for the object store, and $\mathcal{D}_\Gamma$ for the dependency in Γ. $\Gamma(\Xi)$ provides a set of storables, denoted by Θ. The substitution of $\mathcal{D}$ in Γ written $\Gamma[\mathcal{D} \mapsto \mathcal{D}'] \equiv \langle \Sigma_\Gamma, \mathcal{D}' \rangle$ replaces the state dependency.

To handle recursive function calls we introduce a global function store $\mathcal{F}$, which maps a mark ℓ to two pairs of state Γ and value ϑ. Functions are also identified by marks ℓ. The function store contains the merged result of the last evaluation for each function. The first pair $\Gamma_{In}, \vartheta_{In}$ represents the input state and input parameter of all heretofore taken function calls, the second one $\Gamma_{Out}, \vartheta_{Out}$ the output state and return value. For further use we write $\mathcal{F}(\ell)_{In}$ to select the input, and $\mathcal{F}(\ell)_{Out}$ for the output. The substitutions $\mathcal{F}[\ell, In \mapsto \langle \Gamma, \vartheta \rangle]$ and $\mathcal{F}[\ell, Out \mapsto \langle \Gamma, \vartheta \rangle]$ denotes the store update operation on input or output pairs.

Its inference is stated by the following lemma.

Lemma 2 (Function Store)**.** $\forall \mathcal{F}, \Lambda^\ell, \Gamma, \vartheta$: *If* $\langle \Gamma, \vartheta \rangle \sqsubseteq \mathcal{F}(\ell)_{In}$ *and* $\Lambda^\ell = \langle \dot{\sigma}, \lambda^\ell x.e \rangle$ *then* $\Gamma, \dot{\sigma}[x \mapsto \vartheta] \vdash e \Downarrow \Gamma' \mid \vartheta'$ *and* $\langle \Gamma', \vartheta' \rangle \sqsubseteq \mathcal{F}(\ell)_{Out}$.

The proof is by induction on the derivation of $\Gamma''[\mathcal{D} \mapsto \mathcal{D}_{\Gamma''} \sqcup \mathcal{D}_0] \vdash^\Theta_{\text{APP}} \Gamma''(\Xi_0), \vartheta_1 \Downarrow \Gamma''' \mid \vartheta$.

The **trace**$^\ell$ expression registers the dependency from ℓ on all values that pass through it.

The abstraction is defined as relation between $v \in$ *Tainted Value* and $\vartheta \in$ *Abstract Value*.

Definition 3 (Abstraction)**.** *The abstraction* α : *Tainted Value* $\rightarrow$ *Abstract Value is defined as:*

$$\alpha(v : \kappa) ::= \begin{cases} \langle \bot, \{\ell\}, \{\ell | \ell \in \kappa\} \rangle & v = \xi^\ell \\ \langle c, \emptyset, \{\ell | \ell \in \kappa\} \rangle & v = c \end{cases} \tag{3}$$

Definition 4 (Abstract Operation)**.** *The abstract operation* $\Downarrow^\vartheta_{\boldsymbol{op}}$ *is defined in terms of the concrete operation* $\Downarrow^v_{\boldsymbol{op}}$ *as usual:*

$$\Downarrow^\vartheta_{\boldsymbol{op}} (\vartheta_0, \vartheta_1) ::= \bigsqcup \{ \Downarrow^v_{\boldsymbol{op}} (v_0, v_1) \mid v_0 \in \vartheta_0, v_1 \in \vartheta_1 \} \tag{4}$$

(A-PROGRAM)
$$\frac{\Gamma_\bot, \sigma_\bot \vdash e \Downarrow \Gamma \mid \vartheta \quad \vdash^{\mathcal{R},\mathcal{Q}}_{\mathbf{P}} \langle \mathcal{F}_\bot, \Gamma_\bot, \vartheta_\bot \rangle, \langle \mathcal{F}, \Gamma, \vartheta \rangle, e \Downarrow \Gamma' \mid \vartheta'}{\vdash e \Downarrow \Gamma' \mid \vartheta'}$$

(P-ITERATION-NOTEQUALS)
$$\frac{\Gamma_\bot, \sigma_\bot \vdash e \Downarrow \Gamma' \mid \vartheta' \quad \vdash^{\mathcal{R},\mathcal{Q}}_{\mathbf{P}} \mathcal{R}', \langle \mathcal{F}, \Gamma', \vartheta' \rangle, e \Downarrow \mathcal{Q}}{\vdash^{\mathcal{R},\mathcal{Q}}_{\mathbf{P}} \mathcal{R}, \mathcal{R}', e \Downarrow \mathcal{Q}}$$

(P-ITERATION-EQUALS)
$$\frac{}{\vdash^{\mathcal{R},\mathcal{Q}}_{\mathbf{P}} \mathcal{R}, \mathcal{R}, e \Downarrow \mathcal{R}}$$

Figure 9. Inference rules for program interpretation.

(APP-ITERATION)
$$\frac{\Gamma \vdash^{\Lambda^\ell}_{\text{APP}} \Lambda^\ell, \vartheta \Downarrow \Gamma' \mid \vartheta' \quad \Gamma' \vdash^{\Theta}_{\text{APP}} \Theta, \vartheta \Downarrow \Gamma'' \mid \vartheta''}{\Gamma \vdash^{\Theta}_{\text{APP}} \langle \Delta, \Lambda^\ell, \Xi \rangle; \Theta, \vartheta \Downarrow \Gamma'' \mid \vartheta' \sqcup \vartheta''}$$

(APP-ITERATION-EMPTY)
$$\frac{}{\Gamma \vdash^{\Theta}_{\text{APP}} \emptyset, \vartheta \Downarrow \Gamma \mid \vartheta_\bot}$$

(APP-STORE-SUBSET)
$$\frac{\langle \Gamma, \vartheta \rangle \sqsubseteq \mathcal{F}(\ell)_{In} \qquad \langle \Gamma', \vartheta' \rangle = \mathcal{F}(\ell)_{Out}}{\Gamma \vdash^{\Lambda^\ell}_{\text{APP}} \Lambda^\ell, \vartheta \Downarrow \Gamma' \mid \vartheta'}$$

(APP-STORE-NONSUBSET)
$$\frac{\begin{array}{cc} \langle \Gamma, \vartheta \rangle \not\sqsubseteq \mathcal{F}(\ell)_{In} & \langle \dot{\sigma}, \lambda^\ell x.e \rangle = \Lambda^\ell \\ \langle \bar{\Gamma}, \bar{\vartheta} \rangle = \mathcal{F}(\ell)_{In} \sqcup \langle \Gamma, \vartheta \rangle & \mathcal{F}[\ell, In \mapsto \langle \bar{\Gamma}, \bar{\vartheta} \rangle] \\ \bar{\Gamma}, \dot{\sigma}[x \mapsto \bar{\vartheta}] \vdash e \Downarrow \bar{\Gamma}' \mid \bar{\vartheta}' & \mathcal{F}[\ell, Out \mapsto \langle \bar{\Gamma}', \bar{\vartheta}' \rangle] \end{array}}{\Gamma \vdash^{\Lambda^\ell}_{\text{APP}} \Lambda^\ell, \vartheta \Downarrow \bar{\Gamma}' \mid \bar{\vartheta}'}$$

Figure 11. Inference rules for function application.

This definition implies that:

$$\begin{array}{l} \Downarrow^v_{\boldsymbol{op}} (v_0, v_1) = v_{op} \rightarrow \\ \Downarrow^\vartheta_{\boldsymbol{op}} (\alpha(v_0), \alpha(v_1)) \sqsupseteq \alpha(v_{op}) \end{array} \tag{5}$$

Figures 9, 10, 11, 12, and 13 show the inference rules for the big-step evaluation judgment of the abstract semantics. It has the form $\Gamma, \sigma \vdash e \Downarrow \Gamma' \mid \vartheta$. State Γ and scope σ analyze expression e and result in state Γ' and value ϑ. We use notations similar to Figure 5.

The global program rule (A-PROGRAM) (Figure 9) relies on two auxiliary rules to repeatedly evaluate the program until the analysis state, an element of *Analysis Lattice* consisting of $\mathcal{F}$, Γ and ϑ, becomes stable. In the figure, $\mathcal{R}, \mathcal{Q}$ range over *Analysis Lattice* and write $\mathcal{F}_\bot$, $\Gamma_\bot$ and $\sigma_\bot$ for the empty instances of the components.

In Figure 10, the rules for constants (A-CONST) and variables (A-VAR) work similarly as in $\lambda^{\mathcal{D}}_J$.

(A-CONST)
$$\frac{}{\Gamma, \sigma \vdash c \Downarrow \Gamma \mid \langle c, \emptyset, \mathcal{D}_\Gamma \rangle}$$

(A-VAR)
$$\frac{}{\Gamma, \sigma \vdash x \Downarrow \Gamma \mid \sigma(x) \sqcup \mathcal{D}_\Gamma}$$

(A-OP)
$$\frac{\Gamma, \sigma \vdash e_0 \Downarrow \Gamma' \mid \vartheta_0 \quad \Gamma', \sigma \vdash e_1 \Downarrow \Gamma'' \mid \vartheta_1 \quad \langle \mathcal{L}, \Xi \rangle = \Downarrow^{\vartheta}_{\mathbf{op}} (\vartheta_0, \vartheta_1)}{\Gamma, \sigma \vdash \mathbf{op}(e_0, e_1) \Downarrow \Gamma'' \mid \langle \mathcal{L}, \Xi, \mathcal{D}_{\vartheta_0} \sqcup \mathcal{D}_{\vartheta_1} \rangle}$$

(A-NEW-NONEXISTING)
$$\frac{\ell \notin dom(\Gamma) \qquad \Gamma, \sigma \vdash e \Downarrow \Gamma' \mid \langle \mathcal{L}, \Xi, \mathcal{D} \rangle}{\Gamma, \sigma \vdash \mathbf{new}^\ell e \Downarrow \Gamma'[\ell \mapsto \Xi] \mid \langle \mathcal{L}_\perp, \{\ell\}, \mathcal{D}_\Gamma \sqcup \mathcal{D} \rangle}$$

(A-NEW-EXISTING)
$$\frac{\ell \in dom(\Gamma) \qquad \Gamma, \sigma \vdash e \Downarrow \Gamma' \mid \langle \mathcal{L}, \Xi, \mathcal{D} \rangle}{\Gamma, \sigma \vdash \mathbf{new}^\ell e \Downarrow \Gamma'[\ell \mapsto \Gamma(\ell) \sqcup \langle \emptyset, \Lambda^\ell_\perp, \Xi \rangle] \mid \langle \mathcal{L}_\perp, \{\ell\}, \mathcal{D}_\Gamma \sqcup \mathcal{D} \rangle}$$

(A-ABS-NONEXISTING)
$$\frac{\ell \notin dom(\Gamma) \qquad \mathcal{F}[\ell \mapsto \langle \Gamma_\perp, \vartheta_\perp, \Gamma_\perp, \vartheta_\perp \rangle]}{\Gamma, \sigma \vdash \lambda^\ell x.e \Downarrow \Gamma[\ell \mapsto \langle \sigma, \lambda^\ell x.e \rangle] \mid \langle \mathcal{L}_\perp, \{\ell\}, \mathcal{D}_\Gamma \rangle}$$

(A-ABS-EXISTING)
$$\frac{\ell \in dom(\Gamma) \qquad \langle \dot\sigma, \lambda^\ell x.e \rangle = \Gamma(\ell)_{\Lambda^\ell}}{\Gamma, \sigma \vdash \lambda^\ell x.e \Downarrow \Gamma[\ell \mapsto \langle \sigma \sqcup \dot\sigma, \lambda^\ell x.e \rangle] \mid \langle \mathcal{L}_\perp, \{\ell\}, \mathcal{D}_\Gamma \rangle}$$

(A-APP)
$$\frac{\Gamma, \sigma \vdash e_0 \Downarrow \Gamma' \mid \langle \mathcal{L}_0, \Xi_0, \mathcal{D}_0 \rangle \quad \Gamma', \sigma \vdash e_1 \Downarrow \Gamma'' \mid \vartheta_1 \quad \Gamma''[\mathcal{D} \mapsto \mathcal{D}_{\Gamma''} \sqcup \mathcal{D}_0] \vdash^{\Theta}_{\text{APP}} \Gamma''(\Xi_0), \vartheta_1 \Downarrow \Gamma''' \mid \vartheta}{\Gamma, \sigma \vdash e_0(e_1) \Downarrow \langle \Sigma_{\Gamma'''}, \mathcal{D}_\Gamma \rangle \mid \vartheta}$$

(A-GET)
$$\frac{\Gamma, \sigma \vdash e_0 \Downarrow \Gamma' \mid \langle \mathcal{L}_0, \Xi_0, \mathcal{D}_0 \rangle \quad \Gamma', \sigma \vdash e_1 \Downarrow \Gamma'' \mid \langle \mathtt{str}, \Xi_1, \mathcal{D}_1 \rangle \quad \Gamma'' \vdash^{\Theta}_{\text{GET}} \Gamma''(\Xi_0), \mathtt{str} \Downarrow \vartheta}{\Gamma, \sigma \vdash e_0[e_1] \Downarrow \Gamma'' \mid \langle \mathcal{L}_\vartheta, \Xi_\vartheta, \mathcal{D}_0 \sqcup \mathcal{D}_1 \sqcup \mathcal{D}_\vartheta \rangle}$$

(A-PUT)
$$\frac{\Gamma, \sigma \vdash e_0 \Downarrow \Gamma' \mid \langle \mathcal{L}_0, \Xi_0, \mathcal{D}_0 \rangle \quad \Gamma', \sigma \vdash e_1 \Downarrow \Gamma'' \mid \langle \mathtt{str}, \Xi_1, \mathcal{D}_1 \rangle \quad \Gamma'', \sigma \vdash e_2 \Downarrow \Gamma''' \mid \vartheta \quad \Gamma''' \vdash^{\Xi}_{\text{PUT}} \Xi_0, \mathtt{str}, \langle \mathcal{L}_\vartheta, \Xi_\vartheta, \mathcal{D}_0 \sqcup \mathcal{D}_1 \sqcup \mathcal{D}_\vartheta \rangle \Downarrow \Gamma''''}{\Gamma, \sigma \vdash e_0[e_1] = e_2 \Downarrow \Gamma'''' \mid \vartheta}$$

(A-IFTRUE)
$$\frac{\Gamma, \sigma \vdash e_0 \Downarrow \Gamma' \mid \langle \mathcal{L}_0, \Xi_0, \mathcal{D}_0 \rangle \quad \mathcal{L}_0 = \mathtt{true} \quad \Gamma'[\mathcal{D} \mapsto \mathcal{D}_{\Gamma'} \sqcup \mathcal{D}_0], \sigma \vdash e_1 \Downarrow \Gamma'' \mid \vartheta_1}{\Gamma, \sigma \vdash \mathbf{if}\ (e_0)\ e_1,\ e_2 \Downarrow \langle \Sigma_{\Gamma''}, \mathcal{D}_\Gamma \rangle \mid \vartheta_1}$$

(A-IF)
$$\frac{\Gamma, \sigma \vdash e_0 \Downarrow \Gamma' \mid \langle \mathcal{L}_0, \Xi_0, \mathcal{D}_0 \rangle \quad \mathcal{L}_0 \neq \mathtt{true} \wedge \mathcal{L}_0 \neq \mathtt{false} \quad \Gamma'[\mathcal{D} \mapsto \mathcal{D}_{\Gamma'} \sqcup \mathcal{D}_0], \sigma \vdash e_1 \Downarrow \Gamma''_1 \mid \vartheta_1 \quad \Gamma'[\mathcal{D} \mapsto \mathcal{D}_{\Gamma'} \sqcup \mathcal{D}_0], \sigma \vdash e_2 \Downarrow \Gamma''_2 \mid \vartheta_2}{\Gamma, \sigma \vdash \mathbf{if}\ (e_0)\ e_1,\ e_2 \Downarrow \langle \Sigma_{\Gamma''_1} \sqcup \Sigma_{\Gamma''_2}, \mathcal{D}_\Gamma \rangle \mid \vartheta_1 \sqcup \vartheta_2}$$

(A-IFFALSE)
$$\frac{\Gamma, \sigma \vdash e_0 \Downarrow \Gamma' \mid \langle \mathcal{L}_0, \Xi_0, \mathcal{D}_0 \rangle \quad \mathcal{L}_0 = \mathtt{false} \quad \Gamma'[\mathcal{D} \mapsto \mathcal{D}_{\Gamma'} \sqcup \mathcal{D}_0], \sigma \vdash e_2 \Downarrow \Gamma'' \mid \vartheta_2}{\Gamma, \sigma \vdash \mathbf{if}\ (e_0)\ e_1,\ e_2 \Downarrow \langle \Sigma_{\Gamma''}, \mathcal{D}_\Gamma \rangle \mid \vartheta_2}$$

(A-TRACE)
$$\frac{\Gamma[\mathcal{D} \mapsto \mathcal{D}_\Gamma \sqcup \ell], \sigma \vdash e \Downarrow \Gamma' \mid \vartheta}{\Gamma, \sigma \vdash \mathbf{trace}^\ell\ (e) \Downarrow \langle \Sigma_{\Gamma'}, \mathcal{D}_\Gamma \rangle \mid \vartheta}$$

Figure 10. Inference rules for abstract interpretation.

The object and function creation rules are also omitted. They check if an object or function, referenced by ℓ, already exists. In this case the object or function creation has to merge the prototypes or scopes.

The rule (A-OP) is also standard. As in $\lambda^{\mathcal{D}}_J$ the **trace**$^\ell$ expression (A-TRACE) assigns mark ℓ to the sub-state. The rules for the conditional (A-IF), (A-IFTRUE), and (A-IFFALSE) have to handle the case that it is not possible to distinguish between $true$ and $false$. In this case both branches have to be evaluated and the results merged.

Similar problems arise in function application, property reference, and property assignment. Each value can refer to a set of objects including a set of prototypes. Therefore each referenced function has to be evaluated (A-APP) and a property has to be read from (A-GET) or written to (A-PUT) all objects. Results have to be merged. The auxiliary rules are shown in figure 11, 12, and 13.

The rules (APP-ITERATION) and (APP-ITERATION-EMPTY) iterate over all referenced functions. Function application relies on the function store $\mathcal{F}$. Before evaluating the function body, the analyzer checks if the input, consisting of Γ and parameter ϑ, is already subsumed by the stored input. In that case (APP-STORE-SUBSET), the stored result, consisting of output state Γ and return value ϑ, is used.

(GET-ITERATION)

$$\frac{\Gamma \vdash^{\Delta}_{\text{GET}} \Delta, \mathcal{L} \Downarrow \vartheta \quad \Gamma \vdash^{\Theta}_{\text{GET}} \Theta, \mathcal{L} \Downarrow \vartheta' \quad \Gamma \vdash^{\Theta}_{\text{GET}} \Gamma(\Xi), \mathcal{L} \Downarrow \vartheta''}{\Gamma \vdash^{\Theta}_{\text{GET}} \langle \Delta, \Lambda^{\ell}, \Xi \rangle; \Theta, \mathcal{L} \Downarrow \vartheta \sqcup \vartheta' \sqcup \vartheta''}$$

(GET-ITERATION-EMPTY)

$$\frac{}{\Gamma \vdash^{\Theta}_{\text{GET}} \emptyset, \mathcal{L} \Downarrow \vartheta_{\bot}}$$

(GET-INTERSECTION)

$$\frac{\mathcal{L} \sqcap \mathcal{L}_i \neq \bot \quad \Gamma \vdash^{\Theta}_{\text{GET}} \Delta, \mathcal{L} \Downarrow \vartheta'}{\Gamma \vdash^{\Delta}_{\text{GET}} (\mathcal{L}_i : \vartheta_i); \Delta, \mathcal{L} \Downarrow \vartheta_i \sqcup \vartheta'}$$

(GET-NONINTERSECTION)

$$\frac{\mathcal{L} \sqcap \mathcal{L}_i = \bot \quad \Gamma \vdash^{\Theta}_{\text{GET}} \Delta, \mathcal{L} \Downarrow \vartheta'}{\Gamma \vdash^{\Delta}_{\text{GET}} (\mathcal{L}_i : \vartheta_i); \Delta, \mathcal{L} \Downarrow \vartheta'}$$

(GET-EMPTY)

$$\frac{}{\Gamma \vdash^{\Delta}_{\text{GET}} \emptyset, \mathcal{L} \Downarrow \langle \langle \top, \bot, \bot, \bot, \bot \rangle, \emptyset, \emptyset \rangle}$$

Figure 12. Inference rules for property reference.

(PUT-ITERATION)

$$\frac{\Gamma \vdash^{\ell}_{\text{PUT}} \ell, \mathcal{L}, \vartheta \Downarrow \Gamma' \quad \Gamma' \vdash^{\Xi}_{\text{PUT}} \Xi, \mathcal{L}, \vartheta \Downarrow \Gamma''}{\Gamma \vdash^{\Xi}_{\text{PUT}} \ell; \Xi, \mathcal{L}, \vartheta \Downarrow \Gamma''}$$

(PUT-ITERATION-EMPTY)

$$\frac{}{\Gamma \vdash^{\Xi}_{\text{PUT}} \emptyset, \mathcal{L}, \vartheta \Downarrow \Gamma}$$

(PUT-ASSIGNMENT-INDOM)

$$\frac{\mathcal{L} \in dom(\Gamma(\ell))}{\Gamma \vdash^{\ell}_{\text{PUT}} \ell, \mathcal{L}, \vartheta \Downarrow \Gamma[\ell, \mathcal{L} \mapsto \Gamma(\ell)(\mathcal{L}) \sqcup \vartheta]}$$

(PUT-ASSIGNMENT-NOTINDOM)

$$\frac{\mathcal{L} \notin dom(\Gamma(\ell))}{\Gamma \vdash^{\ell}_{\text{PUT}} \ell, \mathcal{L}, \vartheta \Downarrow \Gamma[\ell, \mathcal{L} \mapsto \vartheta]}$$

Figure 13. Inference rules for property assignment.

Otherwise the function body is evaluated (APP-STORE-NONSUBSET) and the store is updated with the result.

For read and write operations the rules (GET-ITERATION), (GET-ITERATION-EMPTY), (PUT-ITERATION) and (PUT-ITERATION-EMPTY) iterate in a similar way over all references. An abstract object maps a lattice element to a value in case a reference is not a singleton value. All entries having an intersection with the reference are affected by the read operation. The prototype-set has to be involved. (GET-INTERSECTION), (GET-NONINTERSECTION) and (GET-EMPTY) shows its inference. Before writing a property, the analyser checks if the property already exists. In this case (PUT-ASSIGNMENT-INDOM), the values get merged. Otherwise (PUT-ASSIGNMENT-NOTINDOM) the value gets assigned. The actual implementation uses a more refined lattice to improve precision.

The abstract interpretation over-approximates the dependencies. The merging of results in (A-IF), (A-APP), (A-GET), and (A-PUT) may cause false positives. While some marked values may be independent from the mark's source, unmarked values are guaranteed to be independent.

6. Applying the Analysis

This section reconsiders the examples Sensitive Data (Section 2.2) and Foreign Code (Section 2.3) from the introduction from an abstract analysis point of view.

6.1 Application: Sensitive Data

Given the newly created mark ℓ_1, the function userHandler is initially called with $\langle\langle \bot, \bot, \bot, \bot, \texttt{uid1} \rangle, \emptyset, \ell_1 \rangle$. If the result of calling Cookie. isset can be determined to be `false`, then the dependencies associated with `false` (ℓ_1 and ℓ_c — resulting from the cookie interface) are bound to the conditional's context.

The Ajax request cannot be evaluated. So, response in onSuccess is a value containing the location of an unspecified object like $\emptyset[\langle \bot, \bot, \bot, \bot, \top \rangle \mapsto \langle\langle \bot, \bot, \bot, \bot, \top \rangle, \emptyset, \ell_a \rangle]$ augmented with ℓ_a. In this case, all further calls to onSuccess are already covered by the first input.

By calling the userHandler with "uid2" a new mark ℓ_2 is introduced. This call is not covered by the first one so that the function is reanalyzed with the merged value $\langle\langle \bot, \bot, \bot, \bot, \mathit{String} \rangle, \emptyset, \{\ell_1, \ell_2\} \rangle$. After the analysis has stabilized, name1 also depends on ℓ_2.

The example illustrates that merging functions can result in conservative results. The implementation has a more refined function store which is indexed by a pair of scope σ and source location ℓ to prevent such inaccuracies.

6.2 Application: Foreign Code

The **trace** expression in line 1 (Section2.3) marks the subcontext for creating the foreach function. The resulting location that points to the function is augmented with this mark. By calling loadForeigenCode the mark is bound to the callees context and finally to the value referencing the foreach function.

By iterating over the array elements (line 11) the dependency annotation is forwarded to the value occurring in result .

Unlike many other security analyses, the objects Array and Array. prototype do not receive marks. If the analysis can determine the updated property exactly, as is the case with foreach, then no other properties can be affected by the update (expect the length). Such an abstract update occurs if the property name is independent from the input. Otherwise, the update happens on a approximated set of property names, all of which are marked by this update.

$$e \quad ::= \quad \ldots \mid \mathbf{trace}^{\ell,\mathcal{A}}(e,\ c) \mid \mathbf{untrace}^{(\mathcal{A}\hookrightarrow\mathcal{A}')}(e,\ c)$$

Figure 14. Extended syntax of $\lambda_J^{\mathcal{A}}$.

(DT-TRACE-CLASSIFIED)
$$\frac{\mathcal{H}, \rho, \kappa \bullet \ell^c \ \vdash\ e \ \Downarrow\ \mathcal{H}' \mid v : \kappa_v}{\mathcal{H}, \rho, \kappa \ \vdash\ \mathbf{trace}^{\ell,\mathcal{A}}\ (e,\ c) \ \Downarrow\ \mathcal{H}' \mid v : \kappa_v}$$

(DT-UNTRACE)
$$\frac{\mathcal{H}, \rho, \kappa \ \vdash\ e \ \Downarrow\ \mathcal{H}' \mid v : \kappa_v \qquad \kappa' = \kappa_v[\ell^{\mathcal{A},c} \mapsto \ell^{\mathcal{A}',c}]}{\mathcal{H}, \rho, \kappa \ \vdash\ \mathbf{untrace}^{(\mathcal{A}\hookrightarrow\mathcal{A}')}\ (e,\ c) \ \Downarrow\ \mathcal{H}' \mid v : \kappa'}$$

Figure 15. Inference rules of $\lambda_J^{\mathcal{A}}$.

(A-TRACE-CLASSIFIED)
$$\frac{\Gamma[\mathcal{D} \mapsto \mathcal{D}_\Gamma \sqcup \ell^{\mathcal{A},c}], \sigma \ \vdash\ e \ \Downarrow\ \Gamma' \mid \vartheta}{\Gamma, \sigma \ \vdash\ \mathbf{trace}^{\ell,\mathcal{A}}\ (e,\ c) \ \Downarrow\ \langle \Sigma_{\Gamma'}, \mathcal{D}_\Gamma \rangle \mid \vartheta}$$

(A-UNTRACE)
$$\frac{\Gamma, \sigma \ \vdash\ e \ \Downarrow\ \Gamma' \mid \vartheta \qquad \mathcal{D}'_\vartheta = \mathcal{D}_\vartheta[\ell^{\mathcal{A},c} \mapsto \ell^{\mathcal{A}',c}]}{\Gamma, \sigma \ \vdash\ \mathbf{untrace}^{(\mathcal{A}\hookrightarrow\mathcal{A}')}\ (e,\ c) \ \Downarrow\ \langle \Sigma_{\Gamma'}, \mathcal{D}_\Gamma \rangle \mid \langle \mathcal{L}_\vartheta, \Xi_\vartheta, \mathcal{D}'_\vartheta \rangle}$$

Figure 16. Inference rules for abstract trace.

6.3 Further sample applications

We also applied our analysis to real-world examples like the *JavaScript Cookie Library with jQuery bindings and JSON support*[1] (version 2.2.0) and the *Rye*[2] library (version 0.1.0), a JavaScript library for DOM manipulation.

These libraries were augmented by wrapping several functions and objects using the trace function. The analysis successfully tracks the flow of the thus marked values, which pop up in the expected places.

7. Dependency Classification

To cater for dependency classification, the accompanying formal framework $\lambda_J^{\mathcal{A}}$ extends $\lambda_J^{\mathcal{D}}$ (Figure 14). In $\lambda_J^{\mathcal{A}}$ marks are classified according to a finite set of modes. They are further augmented by an identifier that can be referred to in the trace and untrace expressions. The operator $\mathbf{trace}^{\ell,\mathcal{A}}$ generates a mark in mode $\mathcal{A}$ and the untrace operator changes the mode of all ℓ-marks according to the sanitization method applied (this distinction is ignored in the example). In the calculus, this change is expressed by the $\mathbf{untrace}^{(\mathcal{A}\hookrightarrow\mathcal{A}')}$ expression, where $\mathcal{A}$ ranges over an unspecified set of modes.

[1] http://code.google.com/p/cookies/

[2] http://ryejs.com/

Marks $\kappa ::= \ldots \mid \ell^{\mathcal{A},c}$ are extended by an new mark-type, a location classified with a class $\mathcal{A}$ and identifier c.

The mark propagation is like in Section 4 (see Figure 15). Rule (DT-TRACE-CLASSIFIED) augments the sub-context with the new classified mark. (DT-UNTRACE) substitutes location $\ell^{\mathcal{A},c}$ by a declassified location $\ell^{\mathcal{A}',c}$.

In the analysis, $\tau ::= \ell \mid \ell^{\mathcal{A},c}$ replaces ℓ in $\mathcal{D}$. Rule (A-TRACE-CLASSIFIED) (Figure 16) generates new dependencies and (A-UNTRACE) substitutes $\mathcal{A}$ by $\mathcal{A}'$ in all locations ℓ labeled with c.

8. Technical Results

To prove the soundness of our abstract analysis we show termination insensitive noninterference. The required steps are proving noninterference for the $\lambda_J^{\mathcal{D}}$ calculus, showing that the abstract analysis provides a correct abstraction of the $\lambda_J^{\mathcal{D}}$ calculus, and that the abstract analysis terminates.

8.1 Noninterference

Proving noninterference requires relating different substitution instances of the same expression. As they may evaluate differently, we need to be able to cater for differences in the heap, for example, with respect to locations.

Definition 5. *A renaming* $\flat ::= \emptyset \mid \flat[\xi^\ell \mapsto \xi^{\ell\prime}]$ *is a partial mapping on locations where* $\flat(\xi^\ell)$ *carries the same mark* ℓ *as* ξ^ℓ.

It extends to values by $\flat(c) = c$.

In the upcoming definitions, the dependency annotation κ contains the marks created by the selected $\mathbf{trace}^\ell$ expression, the body of which may be substituted.

Further, we introduce equivalence relations for each element affected by the ℓ substitution.

Definition 6. *Two marked values are* $\flat, \kappa$*-equivalent* $v_0 : \kappa_0 \equiv_{\flat,\kappa} v_1 : \kappa_1$ *if they are equal as long as their marks are disjoint from* κ.

$$\kappa \cap \kappa_0 = \emptyset \ \wedge\ \kappa \cap \kappa_1 = \emptyset \ \Rightarrow\ \flat(v_0) = v_1 \tag{6}$$

Definition 7. *Two environments* ρ_0, ρ_1 *are* $\flat, \kappa$*-equivalent* $\rho_0 \equiv_{\flat,\kappa} \rho_1$ *if* $R := dom(\rho_0) = dom(\rho_1)$ *and they contain equivalent values.*

$$\forall x \in R :\ \rho_0(x) \equiv_{\flat,\kappa} \rho_1(x) \tag{7}$$

Definition 8. *Two expressions* e_0, e_1 *are* $\flat, \kappa$*-equivalent* $e_0 \equiv_{\flat,\kappa} e_1$ *iff they only differ in the argument of* $\mathbf{trace}^\ell(e')$ *subexpressions with* $\ell \in \kappa$.

$$\begin{aligned} &\kappa = \{\ell_0, ..., \ell_n\} \ \Rightarrow \\ &\exists e'_0 \ldots \exists e'_n :\ e_0 \ =\ e_1[\ell_0 \mapsto e'_0] \ldots [\ell_n \mapsto e'_n] \end{aligned} \tag{8}$$

Definition 9. *Two storables* s_0, s_1 *are* $\flat, \kappa$*-equivalent* $\langle o_0, \langle \rho_0, \lambda^\ell x.e_0 \rangle, p_0 \rangle \ \equiv_{\flat,\kappa}\ \langle o_1, \langle \rho_1, \lambda^\ell x.e_1 \rangle, p_1 \rangle$ *if* $S :=$

$dom(o_0) = dom(o_1)$ *and they only differ in values* $c : \kappa_c$ *with any intersection with* κ*.*

$$\forall str \in S : \; o_0(str) \equiv_{\flat,\kappa} o_1(str) \tag{9}$$

$$\rho_0 \equiv_{\flat,\kappa} \rho_1 \;\wedge\; \lambda^\ell x.e_0 \equiv_{\flat,\kappa} \lambda^\ell x.e_1 \tag{10}$$

$$\flat(p_0) = p_1 \tag{11}$$

Definition 10. *Two heaps* $\mathcal{H}_0, \mathcal{H}_1$ *are* $\flat, \kappa$*-equivalent* $\mathcal{H}_0 \equiv_{\flat,\kappa} \mathcal{H}_1$ *if they only differ in values* $x : \kappa_x$ *with any intersection with* κ *or in one-sided locations.*

$$\forall \xi^\ell \in dom(\flat) : \; \mathcal{H}_0(\xi^\ell) \equiv_{\flat,\kappa} \mathcal{H}_1(\flat(\xi^\ell)) \tag{12}$$

Now, the noninterference theorem can be stated as follows.

Theorem 1. *Suppose* $\mathcal{H}, \rho, \kappa \vdash e \Downarrow \mathcal{H}' \mid v : \kappa_v$*. If* $\ell \notin \bar{\kappa}$ *and* $\mathcal{H} \equiv_{\flat,\{\ell|\ell\notin\bar{\kappa}\}} \tilde{\mathcal{H}}$ *and* $\rho \equiv_{\flat,\{\ell|\ell\notin\bar{\kappa}\}} \tilde{\rho}$ *then* $\tilde{\mathcal{H}}, \tilde{\rho}, \kappa \vdash \bar{e} \Downarrow \tilde{\mathcal{H}}' \mid \tilde{v} : \tilde{\kappa}_v$ *with* $\bar{e} = e[\ell \mapsto \tilde{e}]$ *and* $e \equiv_{\flat,\{\ell|\ell\notin\bar{\kappa}\}} \bar{e}$ *and* $\mathcal{H}' \equiv_{\flat',\{\ell|\ell\notin\bar{\kappa}\}} \tilde{\mathcal{H}}'$ *and* $v : \kappa_v \equiv_{\flat',\{\ell|\ell\notin\bar{\kappa}\}} \tilde{v} : \tilde{\kappa}_v$*, for some* $\flat'$ *extending* $\flat$*.*

The proof is by induction on the evaluation $\Downarrow$.

8.2 Correctness

The abstract analysis is a correct abstraction of the $\lambda_J^{\mathcal{D}}$ calculus. To formalize correctness, we introduce a consistency relation that relates semantic domains of the concrete dependency tracking semantics of $\lambda_J^{\mathcal{D}}$ with the abstract domains.

Definition 11. *The consistency relation* $\prec_{\mathcal{C}}$ *is defined by:*

$$\begin{array}{lcl}
\kappa \prec_{\mathcal{C}} \mathcal{D} & \Leftrightarrow & \kappa \subseteq \mathcal{D} \\
c \prec_{\mathcal{C}} \mathcal{L} & \Leftrightarrow & c \in \mathcal{L} \\
\xi^\ell \prec_{\mathcal{C}} \Xi & \Leftrightarrow & \ell \in \Xi \\
v \prec_{\mathcal{C}} \vartheta & \Leftrightarrow & \begin{cases} \xi^\ell \prec_{\mathcal{C}} \Xi_\vartheta, & v = \xi^\ell \\ c \prec_{\mathcal{C}} \mathcal{L}_\vartheta, & v = c \end{cases} \\
v : \kappa \prec_{\mathcal{C}} \vartheta & \Leftrightarrow & \kappa \prec_{\mathcal{C}} \mathcal{D}_\vartheta \;\wedge\; v \prec_{\mathcal{C}} \vartheta \\
o \prec_{\mathcal{C}} \Delta & \Leftrightarrow & \forall str \in dom(o) : \; \exists \mathcal{L} \in dom(\Delta) : \\
 & & str \prec_{\mathcal{C}} \mathcal{L} \;\wedge\; o(str) \prec_{\mathcal{C}} \Delta(\mathcal{L}) \;\wedge \\
 & & \forall str \notin dom(o) : \; \textbf{\textit{undefined}} \prec_{\mathcal{C}} \Delta(\mathcal{L}) \\
\rho \prec_{\mathcal{C}} \sigma & \Leftrightarrow & \forall x \in dom(\rho) : \; x \in dom(\sigma) \\
 & & \wedge\, \rho(x) \prec_{\mathcal{C}} \sigma(x) \\
f \prec_{\mathcal{C}} \Lambda^\ell & \Leftrightarrow & \langle \rho, \lambda^\ell x.e_f \rangle = f \;\wedge\; \langle \sigma, \lambda^\ell x.e_{\Lambda^\ell} \rangle = \Lambda^\ell \\
 & & \rightarrow\; \rho \prec_{\mathcal{C}} \sigma \;\wedge\; \lambda^\ell x.e_f = \lambda^\ell x.e_{\Lambda^\ell} \\
s \prec_{\mathcal{C}} \theta & \Leftrightarrow & \langle o, f, p \rangle = s \;\wedge\; \langle \Delta, \Lambda^\ell, \Xi \rangle = \theta \\
 & & \rightarrow\; o \prec_{\mathcal{C}} \Delta \;\wedge\; f \prec_{\mathcal{C}} \Lambda^\ell \;\wedge\; p \prec_{\mathcal{C}} \Xi \\
\mathcal{H} \prec_{\mathcal{C}} \Gamma & \Leftrightarrow & \forall \xi^\ell \in dom(\mathcal{H}) : \; \ell \in \Sigma_\Gamma \\
 & & \wedge\, \mathcal{H}(\xi^\ell) \prec_{\mathcal{C}} \Sigma_\Gamma(\ell)
\end{array}$$

Showing adherence to the inference of $\lambda_J^{\mathcal{D}}$ requires to proof that consistent heaps, environments, and values produce a consistent heap and value.

Lemma 3 (Program)**.** $\forall e : \emptyset, \emptyset, \emptyset \vdash e \;\Downarrow\; \mathcal{H} \mid \omega$ *and* $\vdash e \Downarrow \Gamma \mid \vartheta$ *implies that* $\langle \mathcal{H}, \omega \rangle \prec_{\mathcal{C}} \langle \Gamma, \vartheta \rangle$

Given by theorem (2) and definition (11).

Lemma 4 (Property Reference)**.** $\forall \mathcal{H}, \xi^\ell, str, \Gamma, \Xi, \mathcal{L} : \mathcal{H} \prec_{\mathcal{C}} \Gamma$*,* $\xi^\ell \prec_{\mathcal{C}} \Xi$*,* $str \prec_{\mathcal{C}} \mathcal{L}$*, and* $\vdash^{\Theta}_{\text{GET}} \Gamma(\Xi), \mathcal{L} \;\Downarrow\; \vartheta$ *implies* $\mathcal{H}(\xi^\ell)(str) \prec_{\mathcal{C}} \vartheta$

The proof is by definition (11) and by induction on the derivation of $\Gamma'' \vdash^{\Theta}_{\text{GET}} \Gamma''(\Xi_0), \texttt{str} \Downarrow \vartheta$.

Lemma 5 (Property Assignment)**.** $\forall \mathcal{H}, \xi^\ell, str, \omega, \Gamma, \Xi, \mathcal{L}, \vartheta : \mathcal{H} \prec_{\mathcal{C}} \Gamma$*,* $\xi^\ell \prec_{\mathcal{C}} \Xi$*,* $str \prec_{\mathcal{C}} \mathcal{L}$*,* $\omega \prec_{\mathcal{C}} \vartheta$ *and* $\Gamma \vdash^{\Xi}_{\text{PUT}} \Xi, \mathcal{L}, \vartheta \Downarrow \Gamma'$ *implies* $\mathcal{H}[\xi^\ell, str \mapsto \omega] \prec_{\mathcal{C}} \Gamma'$

The proof is by definition (11) and by induction on the derivation of $\Gamma''' \vdash^{\Xi}_{\text{PUT}} \Xi_0, \texttt{str}, \langle \mathcal{L}_\vartheta, \Xi_\vartheta, \mathcal{D}_0 \sqcup \mathcal{D}_1 \sqcup \mathcal{D}_\vartheta \rangle \Downarrow \Gamma''''$.

The following correctness theorem relates the concrete semantics to the abstract semantics.

Theorem 2. *Suppose that* $\mathcal{H}, \rho, \kappa \vdash e \Downarrow \mathcal{H}' \mid x$ *then* $\forall \Gamma, \sigma$ *with* $\mathcal{H} \prec_{\mathcal{C}} \Gamma$*,* $\rho \prec_{\mathcal{C}} \sigma$ *and* $\kappa \prec_{\mathcal{C}} \mathcal{D}_\Gamma$*:* $\Gamma, \sigma \vdash e \Downarrow \Gamma' \mid \vartheta$ *with* $\mathcal{H}' \prec_{\mathcal{C}} \Gamma'$ *and* $x \prec_{\mathcal{C}} \vartheta$*.*

The proof is by induction on the evaluation of e.

8.3 Termination

Finally, we want to guarantee termination of our analysis.

Theorem 3. *For each* Γ*,* σ*, and* e*, there exist* Γ' *and* ϑ *such that* $\Gamma, \sigma \vdash e \Downarrow \Gamma' \mid \vartheta$*.*

For the proof, we observe that all rules of the abstract system in Section 5 are monotone with respect to all their inputs. As the analysis lattice for Γ has finite height, all fixpoint computations in the abstract semantics terminate.

9. Implementation

The implementation extends TAJS[3], the type analyzer for JavaScript. TAJS accepts standard JavaScript [14] .

The abstract interpretation of values and the analysis state are extended by a set of dependency annotations, according to the description in Section 5. As shown in Section 3.1 values can be marked by using the **trace**$^\ell$ expression, which is implemented as a built-in function. A configuration file can be used to trace values produced by JavaScript standard operations or DOM functions. The DOM environment gets constructed during the initialization of TAJS and is available as as normal code would be. The functionality and the dependency propagation for these operations is hard coded.

The extended dependency set has no influence on the lattice structure and does not compromise the precision of the type analyzer. Some notes about the precision can be found in the original work of TAJS [15].

The functions **trace**$^\ell$ and **untrace**$^{(\mathcal{A} \hookrightarrow \mathcal{A}')}$ have to be defined as identity functions before the instrumented code can run in a standard JavaScript engine.

TAJS handles all language features like prototypes, iterations, and exceptions. The specification in Figure 10 simplifies the implementation in several respects. To support the

[3] `http://www.brics.dk/TAJS/`

Benchmark	TAJS	TbDA
Richards (539 lines)	1596	2890
DeltaBlue (880 lines)	3471	4031
Crypto (1689 lines)	3637	7527
RegExp (4758 lines)	3710	4104
Splay (394 lines)	1598	2521
Navier Stokes (387 lines)	2794	3118

Figure 17. Google V8 Benchmark Suite.

conditional to properly account for indirect information flow, the control flow graph had to be extended with special dependency push and pop nodes to encapsulate sub-graphs and to add or remove state dependencies.

The type analyzer provides an over-approximation according to the principles described in Section 3. The analysis result shows the set of traced values and the set of values, which are potentially influenced by them.

There are several ways to use the analyzer. First, a value can be marked and its influence and usage can be determined. This feature can be used to prevent private data from illegal usage and theft. Second, the **trace**$^{\ell}$ function may be used to encapsulate foreign code. As a result of this encapsulation each value which is modified due to the foreign code is highlighted by the analysis. An inspection of the results can show breaches of integrity.

Our implementation is based on an early version of TAJS. The current TAJS version includes support for further language features including `eval` [16]. The dependency analysis can benefit from these extensions by merging it into the current development branch of TAJS.

9.1 Runtime Evaluation

We evaluated the performance impact of our extension by analyzing programs from the Google V8 Benchmark Suite[4]. The programs we selected range from about 400 to 5000 lines of code and perform tasks like an OS kernel simulation, constraint solving, or extraction of regular expressions. The tests were run on a MacBook Pro with 2 GHz Intel Core i7 processor with 8 GB memory.

Figure 17 shows the particular benchmarks together with the averaged time (in milliseconds) to run the analysis and to print the output. The **TAJS** column shows the timing of the original type analyzer without dependency extension. The **TbDA** column shows the timing of our extended version. The figures demonstrate that the dependency analysis leads to a slowdown between 12% and 106%. Two further benchmarks (*RayTrace* and *EarleyBoyer*) did not run to completion because of compatibility problems caused by the outdated version of TAJS underlying our implementation.

[4] `http://v8.googlecode.com/svn/data/benchmarks/v7/run.html`

10. Related Work

Information flow analysis was pioneered by Denning's work [8, 9] which models different security levels as values in a lattice containing elements like *High* and *Low* and which suggests an analysis as an abstract interpretation of the propagation of these levels through the program. Zanioli and others [24] present a recent example of such an analysis with an emphasis on constructing an expressive analysis domain.

Many authors have taken up this approach and transposed it to type theoretic and logical settings [1, 3, 13, 21, 23].

In these systems, input value types are enhanced with security levels. Well-typed programs guarantee that no *High* value flows into a *Low* output and thus noninterference between high inputs and low outputs [10]. Similar to the soundness property of our dependency analysis changes on *High* inputs are unobservable in *Low* outputs. Dependencies are related to security types, but more flexible [2]. They can be analyzed before committing to a fixed security lattice.

Security aspects of JavaScript programs have received much attention. Different approaches focus on static or dynamic analysis techniques, e.g. [6, 12, 17], or attempt to make guarantees by reducing the functionality [20]. The analysis for dependencies is no security analysis per se, but the analysis results express information that is relevant for confidentiality and integrity concerns.

Dependency analysis can be seen as the static counterpart to data tainting (e.g., [7]). Tainting relies on augmenting the run-time representation of a value with information about its properties (like its confidentiality level). Users of the value first check at run time if that use is granted according to some security policy. Dynamic tainting approaches have been successfully used to address security attacks, including buffer overruns, format string attacks, SQL and command injections, and cross-site scripting. Tainting semantics are also used for automatic sanitizer placement [5, 19]. There are also uses in program understanding, software testing, and debugging. Tainting can also be augmented with static analysis to increase its effectiveness [22].

Dynamic languages like JavaScript have many peculiarities that make program analysis and the interpretation of its results challenging [4]. TAJS [15] and hence our analysis can handle almost all dynamic features of JavaScript.

11. Conclusion

We have designed a type-based dependency analysis for JavaScript, proved its soundness and termination, and demonstrated that independence ensures noninterference. We have implemented the analysis as an extension of the open-source JavaScript analyzer TAJS. This approach ensures that our analysis can be applied to real-world JavaScript programs.

While a dependency analysis is not a security analysis, it can form the basis for investigating noninterference. This way, its results can be used to ensure confidentiality and integrity, as well as verify the correct placement of sanitizers.

References

[1] M. Abadi. Access control in a core calculus of dependency. *Electron. Notes Theor. Comput. Sci.*, 172:5–31, April 2007.

[2] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In A. :Aiken, editor, *Proceedings 26th Annual ACM Symposium on Principles of Programming Languages*, pages 147–160, San Antonio, Texas, USA, Jan. 1999. ACM Press.

[3] T. Amtoft and A. Banerjee. Information flow analysis in logical form. In *Proc. 11th Static Analysis Symposium, SAS'04*, pages 33–36. Springer, 2004.

[4] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Computer Security Foundations Symposium, 2009. CSF '09. 22nd IEEE*, pages 43 –59, july 2009. doi: 10.1109/CSF.2009.22.

[5] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Privacy*, pages 387–401, Oakland, California, USA, May 2008. IEEE Computer Society.

[6] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 50–62, New York, NY, USA, 2009. ACM.

[7] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proc. 2007 Symposium on Software Testing and Analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM.

[8] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19:236–243, May 1976.

[9] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20:504–513, July 1977.

[10] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[11] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *Proc. 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 126–150, Berlin, Heidelberg, 2010. Springer-Verlag.

[12] D. Hedin and A. Sabelfeld. Information-flow security for a core of javascript. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 3–18. IEEE, 2012.

[13] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 365–377, New York, NY, USA, 1998. ACM.

[14] E. International. *Standard ECMA-262*, volume 3. 1999.

[15] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proc. 16th Static Analysis Symposium, SAS'09*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.

[16] S. H. Jensen, P. A. Jonsson, and A. Møller. Remedying the eval that men do. In *Proc. 21st International Symposium on Software Testing and Analysis (ISSTA)*, July 2012.

[17] S. Just, A. Cleary, B. Shirley, and C. Hammer. Information flow analysis for javascript. In *Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients*, PLASTIC '11, pages 9–18, New York, NY, USA, 2011. ACM.

[18] M. Keil and P. Thiemann. Type-based dependency analysis for javascript. Technical report, Institute for Computer Science, University of Freiburg, 2013.

[19] B. Livshits and S. Chong. Towards fully automatic placement of security sanitizers and declassifiers. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 385–398, New York, NY, USA, 2013. ACM.

[20] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Safe active content in sanitized javascript. Technical report, Tech. Rep., Google, Inc, 2008.

[21] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21:2003, 2003.

[22] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proc. Network and Distributed System Security Symposium (NDSS)*, volume 42, 2007.

[23] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4:167–187, January 1996.

[24] M. Zanioli, P. Ferrara, and A. Cortesi. Sails: Static analysis of information leakage with Sample. In *Proc. 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1308–1313, New York, NY, USA, 2012. ACM.