Type-based Dependency Analysis for JavaScript

Technical Report

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]: 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 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 \( \text{trace}^A \) marker and to indicate relabelings with an \( \text{untrace}^A \rightarrow A' \) 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 interpretation 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.

The proofs of soundness, noninterference, and termination are shown in appendix A, B, C, and D.

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.

```javascript
var userHandler = function (uid) {
    var userData = {name: ''};
    var onSuccess = function (response) {
        userData = response;
    };
    if (Cookie.isset(uid)) {
        Cookie.request(uid, onSuccess);
    } else {
        Ajax.request("http://\example.org", {
            content: uid
        }, onSuccess);
    }
    return {
        getName : function () {
            return userData.name;
        }
    };
}
var name1 = userHandler (trace("uid1")), getName();
var name2 = userHandler (trace("uid2")), getName();
```

Figure 1. Loading sensitive data.

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.

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.

```javascript
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
The flow-sensitive model retains the independence of the same function but may depend on different sources. In the example, the values in `result` from `loadForeignCode` depend on the user id, the cookie interface, and potentially on the Ajax interface.

Second, our analysis is flow-sensitive. Dependencies are bound to values instead of variable names or parameters. A variable may contain different values depending on different sources during evaluation. In addition, the underlying TAIS 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 `for-each` 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 `sanitize` in the `for-each` 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.

![Figure 2. Using foreign code.](image)

### 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.

```javascript
var secureInput = sanitize(input);
consumer(secureInput);
```

![Figure 3. Analyzing sanitization.](image)

### 3. Formalization

This section presents the JavaScript core calculus $\lambda_J$ along with a semantic definition of independence.

#### 3.1 Syntax of $\lambda_J$

$\lambda_J$ is inspired by JavaScript core calculi from the literature [11, 14]. A $\lambda_J$ 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 `new 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.

### 3.2 Semantic domains

Figure 4 also defines the semantic domains of λJ. A heap maps a location ξℓ 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 H, ρ ⊢ e ⊣ H′ | v. The evaluation of expression e in an initial heap H and environment ρ results in the final heap 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 = (o, f, p) is relayed to the underlying object. The property access s(str) returns undefined by default if the accessed string is not defined in o and the prototype of s is not a location ξℓ. We write s_f for the closure in s. The notation H[ξℓ, str ← v] updates a property of storable H(ξℓ). H[ξℓ → o] initializes an object, and H[ξℓ → f] defines a function.

### 3.3 Independence

The trace 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 trace expression does not influence the final result of e. The first definition formalizes replacing the argument of a trace expression.

**Definition 1.** The substitution e[ℓ ↦ ℓ′] of ℓ in e by ℓ′ is defined as the homomorphic extension of trace(ℓ′)[ℓ ↦ ℓ′] = trace(ℓ′)[ℓ ↦ ℓ′] (1)

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

∀e₁, e₂ : H, ρ ⊢ e₁[ℓ ↦ e₁₁] ⊣ H₁ | v

↔ H, ρ ⊢ e₁[ℓ ↦ e₂] ⊣ H₂ | v

(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 λJ calculus. The later on derived abstract interpretation is formalized on this extended calculus.

This section extends the semantics of λJ with mark propagation. The result calculus λJ only provides a baseline calculus for subsequent static analysis. λJ 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 ⊢ ω := v : κ where κ := 0 | ℓ | κ • κ is a dependency annotation. Tainted Value replaces Value in objects and environments. The operation • joins two dependencies. If ω = v : κv, then write ω • κ for v : κv • κ to apply a dependency annotation to a value.

The big-step evaluation judgment H, ρ, κ ⊢ e ⊣ H′ | ω for λJ extends the one for λJ 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-Ab) 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. 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 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 \( \text{trace}^\ell (e) \) (DT-TRACE) adds the \( \ell \) annotation to the context of expression \( e \). This addition causes all values created or modified in \( e \) to be marked with \( \ell \) (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 \( \downarrow \) (Section 3).

---

**5. Abstract Analysis**

The analysis is an abstraction of the \( \lambda^P \) calculus. Its basis is the lattice for base type values (Figure 7), which is a simplified adaptation of the lattice of TAJS [15]. \( \mathbb{N} \) is the set of floating point numbers, \( \text{STRING} \) the set of string literals, and the annotation \( \uparrow \) turns a set into a flat lattice by adding a bottom and top element. An element of the analysis lattice is a tuple like \( (\bot, \bot, \text{true}, \bot, "x") \) which represents a value which is either the boolean value \text{true} or the string \text{"x"}. The analysis checks for side effects.

---

**Figure 6.** Inference rules of \( \lambda^P \).

**Figure 7.** Base Type Value Lattice.

**Figure 8.** Abstract Semantic Domains.
"\( x \)". Further, \((\bot, \bot, \bot, \top, \bot)\) represents all possible number values. The abstract semantic domains (Figure 8) are similar to the domains arising from the \( \lambda^P \) calculus except that a set of marks \( \Xi \) abstracts a set of concrete locations \( \xi^p \) where \( \ell \in \Xi \). An abstract value \( \vartheta = \langle \ell, \Xi, D \rangle \) is a triple of a lattice element \( \ell \), object marks \( \Xi \), and dependency \( D \).

Hence, each abstract value represents a set of base type values and a set of objects. We write \( L_\theta \) for the analysis lattice, \( \Xi_\theta \) for the marks, and \( D_\theta \) for the dependency component of the abstract value \( \vartheta \). Each abstract object is identified by the mark \( \ell \) corresponding to the new \( e \) 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 \( \Gamma = \langle \Sigma, D \rangle \) where \( \Sigma \) is the mapping from marks to abstract storables. We write \( \Sigma_\theta \) for the object store, and \( D_\theta \) for the dependency for the state \( \Gamma \). \( \Gamma(\Xi) \) provides a set of storables, denoted by \( \Theta \). The substitution of \( D \) in \( \Gamma \) written \( \Gamma[D \mapsto D'] \equiv \langle \Sigma_\theta, D' \rangle \) replaces the state dependency.

To handle recursive function calls we introduce a global function store \( F \), which maps a mark \( \ell \) to two pairs of state \( \Gamma \) and value \( \vartheta \). Functions are also identified by marks \( \ell \). 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 \( F(\ell)_{in} \) to select the input, and \( F(\ell)_{out} \) for the output. The substitutions \( F(\ell, In \mapsto \langle \Gamma, \vartheta \rangle) \) and \( F(\ell, Out \mapsto \langle \Gamma, \vartheta \rangle) \) denotes the store update operation on input or output pairs.

Its inference is stated by the following lemma.

**Lemma 2** (Function Store). \( \forall F, \Lambda^\ell, \Gamma, \vartheta : \Gamma(\ell) \subseteq F(\ell)_{in} \) and \( \Lambda^\ell = (\delta, \Lambda^\ell, x.e) \) then \( \Gamma, \vartheta[x \mapsto \vartheta] \vdash e \Downarrow \Gamma' | \vartheta' \) and \( (\Gamma', \vartheta') \subseteq F(\ell)_{out} \).

The proof is by induction on the derivation of \( \Gamma''[D \mapsto D_{\ell'' \cup \ell_{\theta}}] \vdash \vartheta_{\ell''}(\Xi_0), \vartheta_1 \Downarrow \Gamma'' \mid \vartheta \).

The **trace** expression registers the dependency from \( \ell \) 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 \( \alpha : Tainted \ Value \rightarrow Abstract \ Value \) is defined as:

\[
\alpha(v : \kappa) := \begin{cases}
\langle \bot, \{\ell \} \rangle, \{\ell \in \kappa}\} & \text{if } v = \ell \\
\langle c, \emptyset, \{\ell \in \kappa}\} & \text{if } v = c
\end{cases}
\]

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

\[
\Downarrow^\vartheta_{op} (\vartheta_0, \vartheta_1) := \Downarrow^v_{op} (v_0, v_1) \mid v_0 \in \vartheta_0, v_1 \in \vartheta_1
\]

\[
\Downarrow^\vartheta_{op} (\alpha(v)) = \alpha(v_{op}) \equiv \alpha(v_{op})
\]

**Figure 9.** Inference rules for program interpretation.

**Figure 11.** Inference rules for function application.

\[
\Downarrow^\vartheta (v_0, v_1) = v_{op} \rightarrow \Downarrow^\vartheta (\alpha(v_0), \alpha(v_1)) \equiv \alpha(v_{op})
\]

\[
\Downarrow^\vartheta (v_0, v_1) = v_{op} \rightarrow \Downarrow^\vartheta (\alpha(v_0), \alpha(v_1)) \equiv \alpha(v_{op})
\]

\[
\Downarrow^\vartheta (v_0, v_1) = v_{op} \rightarrow \Downarrow^\vartheta (\alpha(v_0), \alpha(v_1)) \equiv \alpha(v_{op})
\]

\[
\Downarrow^\vartheta (v_0, v_1) = v_{op} \rightarrow \Downarrow^\vartheta (\alpha(v_0), \alpha(v_1)) \equiv \alpha(v_{op})
\]

\[
\Downarrow^\vartheta (v_0, v_1) = v_{op} \rightarrow \Downarrow^\vartheta (\alpha(v_0), \alpha(v_1)) \equiv \alpha(v_{op})
\]

\[
\Downarrow^\vartheta (v_0, v_1) = v_{op} \rightarrow \Downarrow^\vartheta (\alpha(v_0), \alpha(v_1)) \equiv \alpha(v_{op})
\]

\[
\Downarrow^\vartheta (v_0, v_1) = v_{op} \rightarrow \Downarrow^\vartheta (\alpha(v_0), \alpha(v_1)) \equiv \alpha(v_{op})
\]

\[
\Downarrow^\vartheta (v_0, v_1) = v_{op} \rightarrow \Downarrow^\vartheta (\alpha(v_0), \alpha(v_1)) \equiv \alpha(v_{op})
\]

\[
\Downarrow^\vartheta (v_0, v_1) = v_{op} \rightarrow \Downarrow^\vartheta (\alpha(v_0), \alpha(v_1)) \equiv \alpha(v_{op})
\]

\[
\Downarrow^\vartheta (v_0, v_1) = v_{op} \rightarrow \Downarrow^\vartheta (\alpha(v_0), \alpha(v_1)) \equiv \alpha(v_{op})
\]

\[
\Downarrow^\vartheta (v_0, v_1) = v_{op} \rightarrow \Downarrow^\vartheta (\alpha(v_0), \alpha(v_1)) \equiv \alpha(v_{op})
\]

\[
\Downarrow^\vartheta (v_0, v_1) = v_{op} \rightarrow \Downarrow^\vartheta (\alpha(v_0), \alpha(v_1)) \equiv \alpha(v_{op})
\]

\[
\Downarrow^\vartheta (v_0, v_1) = v_{op} \rightarrow \Downarrow^\vartheta (\alpha(v_0), \alpha(v_1)) \equiv \alpha(v_{op})
\]

\[
\Downarrow^\vartheta (v_0, v_1) = v_{op} \rightarrow \Downarrow^\vartheta (\alpha(v_0), \alpha(v_1)) \equiv \alpha(v_{op})
\]

\[
\Downarrow^\vartheta (v_0, v_1) = v_{op} \rightarrow \Downarrow^\vartheta (\alpha(v_0), \alpha(v_1)) \equiv \alpha(v_{op})
\]

\[
\Downarrow^\vartheta (v_0, v_1) = v_{op} \rightarrow \Downarrow^\vartheta (\alpha(v_0), \alpha(v_1)) \equiv \alpha(v_{op})
\]

\[
\Downarrow^\vartheta (v_0, v_1) = v_{op} \rightarrow \Downarrow^\vartheta (\alpha(v_0), \alpha(v_1)) \equiv \alpha(v_{op})
\]
Figure 10. Inference rules for abstract interpretation.

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 $\lambda^D_P$.

The object and function creation rules are also omitted. They check if an object or function, referenced by $\ell$, 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 $\lambda^D_P$ the trace expression (A-TRACE) assigns mark $\ell$ 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 \( \Gamma \) and parameter \( \vartheta \), is already subsumed by the stored input. In that case (APP-STORE-SUBSET), the stored result, consisting of output state \( \Gamma \) and return value \( \vartheta \), is used. 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) show its inference. Before writing a property, the analyzer 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-I\(\Phi\)), (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 6.1) and Foreign Code (Section 6.2) from the introduction from an abstract analysis point of view.

6.1 Application: Sensitive Data

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

The Ajax request cannot be evaluated. So, response in \texttt{onSuccess} is a value containing the location of an unspecified object like \( \emptyset([\bot, \bot, \bot, \bot, \bot]) \mapsto \langle [\top, \bot, \bot, \bot, \bot], \emptyset, \ell_\bot \rangle \). If the call to \	exttt{onSuccess} are already covered by the first input.

By calling the \texttt{userHandler} with “uid2” a new mark \( \ell_2 \) is introduced. This call is not covered by the first one so that the function is reanalyzed with the merged value \( \langle [\top, \bot, \bot, \bot, \bot], \emptyset, \ell_\bot \rangle \). After the analysis has stabilized, \texttt{name1} also depends on \( \ell_2 \).

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 \( \sigma \) and source location \( \ell \) to prevent such inaccuracies.

6.2 Application: Foreign Code

The trace expression in line 1 (Section 6.3) marks the sub-context for creating the \texttt{forEach} function. The returning location that points to the function is augmented with this mark. By calling \texttt{loadForeignCode} the mark is bound to the callees context and finally to the value referencing the \texttt{forEach} function.

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

Unlike many other security analyses, the objects \texttt{Array} and \texttt{Array.prototype} do not receive marks. If the analysis can determine the updated property exactly, as is the case with
\[ e ::= \ldots | \text{trace}^c A (e, c) | \text{untrace}^{A \mapsto A'} (e, c) \]

Figure 14. Extended syntax of \(\lambda_f^j\).

\[
\begin{align*}
\text{(DT-Trace-Classified)} & : \quad \mathcal{H}, \rho, \kappa \vdash e \downarrow \mathcal{H}' | v : \kappa_v \\
\text{H, \rho, \kappa} & \vdash \text{trace}^{e \mapsto A} (e, c) \downarrow \mathcal{H}' | v : \kappa_v
\end{align*}
\]

\[
\begin{align*}
\text{(DT-Untrace)} & : \quad \mathcal{H}, \rho, \kappa \vdash \text{untrace}^{(A \mapsto A')} (e, c) \downarrow \mathcal{H}' | v : \kappa'
\end{align*}
\]

Figure 15. Inference rules of \(\lambda_f^j\).

\begin{align*}
\text{(A-Trace-Classified)} & : \quad \Gamma[D \mapsto D_T \cup \ell^{A^C}], \sigma \vdash e \downarrow \Gamma' | \vartheta \\
\Gamma, \sigma & \vdash \text{trace}^{e \mapsto A} (e, c) \downarrow (\Sigma_T, D_T) | \vartheta
\end{align*}

\begin{align*}
\text{(A-Untrace)} & : \quad \Gamma, \sigma \vdash \text{untrace}^{(A \mapsto A')} (e, c) \downarrow (\Sigma_T, D_T) | (\vartheta, \Xi, \Delta')
\end{align*}

Figure 16. Inference rules for abstract trace.

\[
\text{foreach}, \text{then no other properties can be affected by the update (expect the length). Such an abstract update occurs if the property name is independent from the input. Otherwise, the update happens on a approximated set of property names, all of which are marked by this update.}
\]

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 \text{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 \(\lambda_f^j\) extends \(\lambda_f^j\) (Figure[13]). In \(\lambda_f^j\) marks are classified according to a finite set of modes. They are further augmented by an identifier that can be referred to in the \text{trace} and \text{untrace} expressions. The operator \text{trace}^\ell A generates a mark in mode \(A\) and the \text{untrace} operator changes the mode of all \(\ell\)-marks according to the sanitization method applied (this distinction is ignored in the example). In the calculus, this change is expressed by the \text{untrace}^{A \mapsto A'} expression, where \(A\) ranges over an unspecified set of modes.

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

For each \(\ell\), the mark propagation is like in Section 4 (see Figure 15). In the calculus, and that the abstract analysis terminates.

8. Technical Results

To prove the soundness of our abstract analysis we show termination insensitive noninterference. The required steps are proving noninterference for the \(\lambda_f^j\) calculus, showing that the abstract analysis provides a correct abstraction of the \(\lambda_f^j\) 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 \(\hat{b} \ ::= [] | b[\xi^\ell \mapsto \xi']\) is a partial mapping on locations where \(\hat{b}(\xi^\ell)\) carries the same mark \(\ell\) as \(\xi^\ell\).

It extends to values by \(\hat{b}(c) = c\).

In the upcoming definitions, the dependency annotation \(\kappa\) contains the marks created by the selected \text{trace} expression, the body of which may be substituted.

Further, we introduce equivalence relations for each element affected by the \(\ell\) substitution.

Definition 6. Two marked values are \(\hat{b}, \kappa\)-equivalent \(v_0, v_1 : \kappa_0 \equiv_{\hat{b}, \kappa} v_1 : \kappa_1\) if they are equal as long as their marks are disjoint from \(\kappa\).

\[
\kappa \cap \kappa_0 = \emptyset \land \kappa \cap \kappa_1 = \emptyset \Rightarrow \hat{b}(v_0) = v_1
\]

(6)

Definition 7. Two environments \(\rho_0, \rho_1\) are \(\hat{b}, \kappa\)-equivalent \(\rho_0 \equiv_{\hat{b}, \kappa} \rho_1\) if \(R := \text{dom}(\rho_0) = \text{dom}(\rho_1)\) and they contain equivalent values.

\[
\forall x \in R : \rho_0(x) \equiv_{\hat{b}, \kappa} \rho_1(x)
\]

(7)

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

\[
\kappa = \{\ell_0, \ldots, \ell_n\} \Rightarrow \exists e'_0 \ldots \exists e'_n : e_0 = e_1[\ell_0 \mapsto e'_0] \ldots [\ell_n \mapsto e'_n]
\]

(8)
Definition 9. Two storables $s_0, s_1$ are $\beta, \kappa$-equivalent \[
\langle o_0, (p_0, \lambda x.e_0), p_0 \rangle \equiv_{\beta, \kappa} \langle o_1, (p_1, \lambda x.e_1), p_1 \rangle \text{ if } S := \text{dom}(o_0) = \text{dom}(o_1) \text{ and they only differ in values } c : \kappa_c \text{ with any intersection with } \kappa.
\]

(9)

\[
\rho_0 \equiv_{\beta, \kappa} \rho_1 \land \lambda x.e_0 \equiv_{\beta, \kappa} \lambda x.e_1
\]

(10)

\[
\langle b(p_0) \rangle = p_1 \quad (11)
\]

Definition 10. Two heaps $H_0, H_1$ are $\beta, \kappa$-equivalent $H_0 \equiv_{\beta, \kappa} H_1$ if they only differ in values $x : \kappa_x$ with any intersection with $\kappa$ or in one-sided locations.

\[
\forall e \in \text{dom}(b) : H_0(\xi^e) \equiv_{\beta, \kappa} H_1(b(\xi^e))
\]

(12)

Now, the noninterference theorem can be stated as follows.

Theorem 1. Suppose $H, \rho, \kappa \vdash e \Downarrow H' \mid x : \kappa_x$. If $\ell \not\in \kappa$ and $H \equiv_{\beta, \{e(\xi^e)\}} H'$ and $\rho \equiv_{\beta, \{e(\xi^e)\}} \rho'$ then $H, \rho, \kappa \vdash e \Downarrow H'' \mid \bar{v} : \kappa_v$ with $e = e(\ell \mapsto \bar{e})$ and $e \equiv_{\beta, \{e(\xi^e)\}} \bar{e}$ and $H'' \equiv_{\beta, \{e(\xi^e)\}} H'$ and $v : \kappa_v \equiv_{\beta, \{e(\xi^e)\}} \bar{v} : \kappa_v$ for some $\ell$ extending $\beta$.

The proof is by induction on the evaluation $\Downarrow$ (Section B).

8.2 Correctness

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

Definition 11. The consistency relation $\prec_c$ is defined by:

\[
k \prec_c D \Leftrightarrow k \subseteq D
\]

\[
c \prec_c L \Leftrightarrow c \in L
\]

\[
\xi^e \prec_c \Xi \Leftrightarrow \ell \in \Xi
\]

\[
v \prec_c \vartheta \Leftrightarrow \begin{cases} 
\xi^e \prec_c \Xi, & v = \xi^e \\
\ell \prec_c L, & v = c 
\end{cases}
\]

\[
v : k \prec_c \vartheta \Leftrightarrow \begin{cases} 
k \prec_c D, & v \prec_c \vartheta 
\end{cases}
\]

\[
o \prec_c \Delta \Leftrightarrow \forall \str \in \text{dom}(\alpha) : \exists \cl \in \text{dom}(\Delta) : 
\str \prec_c L \land o(\str) \prec_c \Delta(\cl) \land 
\forall \str \notin \text{dom}(\alpha) : \text{undefined} \prec_c \Delta(\cl)
\]

\[
\rho \prec_c \sigma \Leftrightarrow \forall x \in \text{dom}(\rho) : x \in \text{dom}(\sigma)
\land \rho(x) \prec_c \sigma(x)
\]

\[
f \prec_c \Lambda^e \Leftrightarrow \langle \rho, \lambda x.e_f \rangle = f \land \langle \sigma, \lambda x.e_M \rangle = \Lambda^e
\]

\[
s \prec_c \theta \Leftrightarrow \langle o, f, p \rangle = s \land \langle \Delta, \Lambda^e, \Xi \rangle = \theta
\]

\[
\mathcal{H} \prec_c \Gamma \Leftrightarrow \forall \ell \in \text{dom}(H) : \ell \in \Sigma_{\Gamma}
\land \mathcal{H}(\xi^e) \prec_c \Sigma_{\Gamma}(\ell)
\]

Showing adherence to the inference of $\lambda^T_D$ 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 H \mid \omega \text{ and } \vdash e \Downarrow \Gamma \mid \vartheta \text{ implies that } \langle \mathcal{H}, \omega \rangle \prec_c (\Gamma, \vartheta)

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

Lemma 4 (Property Reference). \forall H, \xi^e, \str, \Gamma, \Xi, \mathcal{L} : H \prec_c \Gamma, \xi^e \prec_c \Xi, \str \prec_c \mathcal{L}, \text{ and } \Gamma \vdash \mathcal{L}(\Xi) \downarrow \vartheta \implies H(\xi^e(\str)) \prec_c \vartheta

The proof is by definition (11) and by induction on the derivation of $\Gamma'' \vdash_{\text{UT}} \Gamma''(\Xi), \str \downarrow \vartheta$.

Lemma 5 (Property Assignment). \forall H, \xi^e, \str, \omega, \Gamma, \Xi, \mathcal{L}, \vartheta : H \prec_c \Gamma, \xi^e \prec_c \Xi, \str \prec_c \mathcal{L}, \omega \prec_c \vartheta \text{ and } \Gamma \vdash_{\text{UT}} \Xi, \mathcal{L}, \vartheta \downarrow \Gamma' \implies H[\xi^e, \str \mapsto \omega] \prec_c \Gamma'

The proof is by definition (11) and by induction on the derivation of $\Gamma'' \vdash_{\text{UT}} \Xi, \str, \langle \mathcal{L}, \Xi, D \rangle \downarrow \Gamma''$.

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

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

The proof is by induction on the evaluation of $e$ (Section 14).

8.3 Termination

Finally, we want to guarantee termination of our analysis.

Theorem 3. For each $\Gamma, \sigma, \text{ and } e$, there exist $\Gamma', \vartheta$ such that $\Gamma, \sigma \vdash e \Downarrow \Gamma' \mid \vartheta$.

For the proof (Section 14), 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 $\Gamma$ has finite height, all fixpoint computations in the abstract semantics terminate.

9. Implementation

The implementation extends TAJS\(^1\), the type analyzer for JavaScript. TAJS accepts standard JavaScript\(^1\) for parsing.

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\(^e\) 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\(^1\).

The functions trace\(^e\) and untrace\(^\text{un} \rightarrow \text{a} \rightarrow \text{a} \) have to be defined as identity functions before the instrumented code can run in a standard JavaScript engine.

\(^1\)http://www.brics.dk/TAJS/
Table 1. Google V8 Benchmark Suite.

| 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.

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

9.2 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 \texttt{High} and \texttt{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, 22]. In these systems, input value types are enhanced with security levels. Well-typed programs guarantee that no \texttt{High} value flows into a \texttt{Low} output and thus noninterference between high inputs and low outputs [10]. Similar to the soundness property of our dependency analysis changes on \texttt{High} inputs are unobservable in \texttt{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 [8, 19]. There are also uses in program understanding, software testing, and debugging. Tainting can also be augmented with static analysis to increase its effectiveness [2].

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.

\footnote{\url{http://v8.googlecode.com/svn/data/benchmarks/v7/run.html}}
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. Amtorf 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. Felmetser, 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. Hedín 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 classifiers. 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.
A. Context Dependency

Proof of Lemma 1.\footnote{This proof is not provided in the image.}

\[ \mathcal{H}, \rho, \kappa \vdash e \Downarrow \mathcal{H}' \mid v_e : \kappa_e \] implies that \( \kappa \subseteq \kappa_e \). By induction on the derivation of \( \mathcal{H}, \rho, \kappa \vdash e \Downarrow \mathcal{H}' \mid v_e : \kappa_e \).

Case (DT-CONST): \( e \equiv c; \ v_e \equiv c; \ k_e \equiv k. \) Claim holds because \( k \subseteq k \).

Case (DT-VAR): \( e \equiv x; \ v_e \equiv v; \ k_e \equiv k, \kappa \mid v : k_v = \rho(x). \) Claim holds because \( k \subseteq k \).

Case (DT-OP): \( e \equiv \text{op}(e_0, e_1); \ v_e \equiv v_{op}; \ k_e \equiv k_0 \cdot k_1. \)

By inversion

\[ \mathcal{H}, \rho, \kappa \vdash e_0 \Downarrow \mathcal{H}' \mid v_0 : k_0 \quad (13) \]

holds. Induction yields \( \kappa \subseteq k_0 \) and \( \kappa \subseteq k_1 \). Claim holds because \( k \subseteq k \).

Case (DT-ABS): \( e \equiv \lambda x.e; \ v_e \equiv \xi; \ k_e \equiv k. \) Claim holds because \( k \subseteq k \).

Case (DT-NEW): \( e \equiv \text{new}\xi \cdot e_p; \ v_e \equiv \xi; \ k_e \equiv k. \) By inversion

\[ \mathcal{H}, \rho, \kappa \vdash e_p \Downarrow \mathcal{H}' \mid v : k_v \quad (15) \]

holds. Induction yields \( \kappa \subseteq k_v \). Claim holds because \( k \subseteq k \).

Case (DT-APP): \( e \equiv e_0(e_1); \ v_e \equiv v; \ k_e \equiv k. \) By inversion

\[ \mathcal{H}', \rho \circlearrowleft \alpha \Downarrow \mathcal{H}'' \mid \xi^\ell : \kappa\xi^\ell \quad (16) \]

holds where \( \rho \circlearrowleft \alpha \) is the closure obtained from evaluating \( e_0 \). It follows that \( k \subseteq k \).

Case (DT-GET): \( e \equiv e_0[e_1]; \ v_e \equiv v; \ k_e \equiv k_0 \cdot k_\xi \cdot k_{str} \mid v : k_v = \mathcal{H}''(\xi^\ell)(\text{str}). \)

By inversion

\[ \mathcal{H}, \rho, k \vdash e_0 \Downarrow \mathcal{H}' \mid \xi^\ell : k\xi^\ell \quad (17) \]

\[ \mathcal{H}', \rho, k \vdash e_1 \Downarrow \mathcal{H}'' \mid \text{str} : k_{str} \quad (18) \]

holds. Induction yields \( \kappa \subseteq k\xi^\ell \) and \( \kappa \subseteq k_{str}. \) Claim holds because \( k \subseteq k \).

Case (DT-PUT): \( e \equiv e_0[e_1] = e_2; \ v_e \equiv v; \ k_e \equiv k_v. \) By inversion

\[ \mathcal{H}', \rho, k \vdash e_2 \Downarrow \mathcal{H}' \mid \xi^\ell : k_v \quad (19) \]

holds. Induction yields \( \kappa \subseteq k_v. \) Claim holds because \( k \subseteq k \).

Case (DT-IFTRUE): \( e \equiv \text{if} (e_0 \ e_1, e_2; \ v_0 \equiv v_1; \ k_e \equiv k_1. \)

By inversion

\[ \mathcal{H}', \rho, k \vdash e_0 \Downarrow \mathcal{H}' \mid v_1 : k_1 \quad (20) \]

holds. It follows that \( k \subseteq k \cdot k_0 \subseteq k_1 \).

Case (DT-IFFALSE): Analogous to case (DT-IFTRUE).

Case (DT-TRACE): \( e \equiv \text{trace}\ell (e'); \ v_e \equiv v; \ k_e \equiv k_v. \) By inversion

\[ \mathcal{H}, \rho, k \cdot \ell \Downarrow e' \Downarrow \mathcal{H}' \mid v : k_v \quad (21) \]

holds. It follows that \( k \subseteq k \cdot \ell \subseteq k_v \).

B. Noninterference

Proof of Theorem 1.\footnote{This proof is not provided in the image.}

Suppose \( \mathcal{H}, \rho, k \vdash e \Downarrow \mathcal{H}' \mid v : k_v. \) If \( \ell \notin k \) and \( \mathcal{H} \equiv_{\gamma_{\ell}(t \in R)} \mathcal{H}' \) and \( \rho \equiv_{\gamma_{\ell}(t \in R)} \rho' \) then \( \mathcal{H}, \rho, \kappa \vdash e \Downarrow \mathcal{H}' \mid v : \kappa_v \) with \( e' = e[t \mapsto e] \) and \( \mathcal{H}' \equiv_{\gamma_{\ell'}(t \in R)} \mathcal{H} \) and \( v : k_v \equiv_{\gamma_{\ell'}(t \in R)} v \equiv_{\gamma_{\ell'}(t \in R)} v \).

By induction on the derivation of \( \mathcal{H}, \rho, \kappa \vdash e \Downarrow \mathcal{H}' \mid v : k \) and \( \mathcal{H}, \rho, \kappa \vdash e \Downarrow \mathcal{H}' \mid v : k_v \).

Case (DT-CONST): \( e \equiv c. \) By definition (3) claim holds because \( c = c \mapsto c \equiv_{\gamma_{\ell}(t \in R)} c : k \).

Case (DT-VAR): \( e \equiv x; \ \forall \ell \notin k \) yields

Subcase \( \ell \notin k: \) By definition (3), (7) claim holds because \( \rho \equiv_{\gamma_{\ell}(t \in R)} \rho \equiv_{\gamma_{\ell}(t \in R)} \rho \).

Subcase \( \ell \in k: \) From definition (3), (7) claim holds because \( e \in \rho \equiv_{\gamma_{\ell}(t \in R)} \rho \equiv_{\gamma_{\ell}(t \in R)} \rho \).

Case (DT-OP): \( e \equiv \text{op}(e_0, e_1). \) By inversion

\[ \forall \ell \notin k : \mathcal{H}, \rho, \kappa \vdash e_0 \Downarrow \mathcal{H}' \mid v \equiv_{\gamma_{\ell'}(t \in R)} v \]

holds. \( \forall \ell \notin k \) yields

Subcase \( \ell \not\in k_0 \land \ell \not\in k_0 \land \ell \not\in k_1 \land \ell \not\in k_1: \) By definition (6) we obtain

\[ \ell \not\in k_0 \land \ell \not\in k_0 \to v_0 = v_0 \]

and this leads to:

\[ \Downarrow_{op} (v_0, v_1) \equiv_{k_0} k_1 \equiv_{\gamma_{\ell}(t \in R)} k_1 \]

Subcase \( \ell \in k_0 \lor \ell \in k_0 \lor \ell \in k_1 \lor \ell \in k_1: \) Again by (6)

\[ \Downarrow_{op} (v_0, v_1) \equiv_{k_0} k_1 \equiv_{\gamma_{\ell}(t \in R)} k_1 \]

holds.

Claim holds because \( \Downarrow_{op} (v_0, v_1) \equiv_{k_0} k_1 \equiv_{\gamma_{\ell'}(t \in R)} \Downarrow_{op} (v_0, v_1) \equiv_{k_0} k_1 \equiv_{\gamma_{\ell'}(t \in R)} k_1 \).

Case (DT-ABS): \( e \equiv \lambda x.e \) where location \( s \) with \( s \notin \text{dom}(H) \) and location \( s \notin \text{dom}(H) \) extends the renaming \( b \) with \( b' \equiv s \). By definition (6) \( s \equiv_{\gamma_{\ell'}(t \in R)} \) \( s \equiv_{\gamma_{\ell'}(t \in R)} \) holds. Claim holds because:

\[ \mathcal{H}[s \rightarrow \rho, \lambda x.e] \equiv_{\gamma_{\ell'}(t \in R)} \mathcal{H}[s \rightarrow \rho, \lambda x.e] \]

(28)
Case (DT-NEW): $e \equiv \text{new}^\ell e_p$ where location $\xi^\ell$ with $\xi^\ell \notin \text{dom}(\mathcal{H})$ and location $\xi^\ell$ with $\xi^\ell \notin \text{dom}(\mathcal{H})$ extends the renaming $b$ with $b' = b[\xi^\ell \mapsto \xi']$. By definition (6) $\xi^\ell : \kappa \equiv_{\nu, \{\ell | (e_p \in \kappa)} \xi^\ell : \kappa$ holds. By inversion
\[ \forall \ell \notin \bar{\kappa} : \bar{\mathcal{H}}, \bar{\rho}, \bar{\kappa} \vdash e_p[\ell \mapsto \bar{e}] \Downarrow \bar{\mathcal{H}}' | \bar{v} : \bar{\kappa}_v \] holds. Claim holds because:
\[ \mathcal{H}'[\xi^\ell \mapsto v] \equiv_{\nu, \{\ell | (e_p \in \kappa)} \bar{\mathcal{H}}'[\xi^\ell \mapsto \bar{v}] \]
(30)

Case (DT-APP): $e \equiv e_0(e_1)$. By inversion
\[ \forall \ell \notin \bar{\kappa} : \bar{\mathcal{H}}, \bar{\rho}, \bar{\kappa} \vdash e_0(\ell \mapsto \bar{e}) \Downarrow \mathcal{H}' | \xi^\ell : \bar{\kappa}_0 \]
(31)

\[ \forall \ell \notin \bar{\kappa} : \bar{\mathcal{H}}', \bar{\rho}, \bar{\kappa} \vdash e_1[\ell \mapsto \bar{e}] \Downarrow \mathcal{H}' | \bar{v}_1 : \bar{\kappa}_1 \]
(32)

holds. $\forall \ell \notin \bar{\kappa}$ yields

Subcase $\ell \notin \kappa_0 \land \ell \notin \bar{\kappa}_0$: By definition (6) we obtain
\[ \ell \notin \kappa_0 \land \ell \notin \bar{\kappa}_0 \rightarrow b(\xi^\ell) = \bar{\xi}^\ell \]
and this leads by definition (6), (10) to:
\[ \langle \bar{\rho}, \lambda^\ell x.\bar{e} \rangle = \mathcal{H}(\xi^\ell) \land \langle \bar{\rho}, \lambda^\ell x.\bar{e} \rangle = \bar{\mathcal{H}}(\bar{\xi}^\ell) \]
\[ \rightarrow \bar{\rho} \equiv_{\nu, \{\ell | (e_p \in \kappa)} \bar{\rho} \land \lambda^\ell x.\bar{e} \equiv_{\nu, \{\ell | (e_p \in \kappa)} \lambda^\ell x.\bar{e} \] (34)

Hence, $\bar{\rho}[x \mapsto \bar{v}_1 : \kappa_1] \equiv_{\nu, \{\ell | (e_p \in \kappa)} \bar{\rho}[x \mapsto \bar{v}_1 : \bar{\kappa}_1]$ and $\bar{e} \equiv_{\nu, \{\ell | (e_p \in \kappa)} \bar{e}[x \mapsto \bar{v}] \downarrow \bar{e}[x \mapsto \bar{v}]$ complies the postcondition to apply the induction where
\[ \forall \ell \notin \bar{\kappa} : \bar{\mathcal{H}}'', \bar{\rho}[x \mapsto \bar{v}_1 : \kappa_1], \bar{\kappa} \vdash e[\ell \mapsto \bar{e}] \downarrow \mathcal{H}'' | \bar{v} : \bar{\kappa}_v \] (35)

Subcase $\ell \in \kappa_0 \lor \ell \notin \bar{\kappa}_0$: By lemma (1) applied on $\kappa \bullet \kappa_0 \subseteq \kappa_v$ and $\kappa \bullet \bar{\kappa}_0 \subseteq \bar{\kappa}_v$ the claim holds because:
\[ v : \kappa_v \equiv_{\nu, \{\ell | (e_p \in \kappa)} \bar{v} : \bar{\kappa}_v \]
\[ \mathcal{H}'' \equiv_{\nu, \{\ell | (e_p \in \kappa)} \bar{\mathcal{H}}'' \]
(37)

Case (DT-GET): $e \equiv e_0[e_1]$. By inversion
\[ \forall \ell \notin \bar{\kappa} : \bar{\mathcal{H}}, \bar{\rho}, \bar{\kappa} \vdash e_0(\ell \mapsto \bar{e}) \Downarrow \mathcal{H}' | \xi^\ell : \bar{\kappa}_0 \] (38)

\[ \forall \ell \notin \bar{\kappa} : \bar{\mathcal{H}}', \bar{\rho}, \bar{\kappa} \vdash e_1[\ell \mapsto \bar{e}] \Downarrow \mathcal{H}' | \bar{v}_1 : \bar{\kappa}_1 \] (39)

hold.

Subcase $\ell \in \kappa_0 \land \ell \notin \bar{\kappa}_0$: By definition (6) we obtain
\[ \ell \in \kappa_0 \land \ell \notin \bar{\kappa}_0 \rightarrow \mathcal{H}''[\xi^\ell \mapsto v] \equiv_{\nu, \{\ell | (e_p \in \kappa)} \bar{\mathcal{H}}''[\xi^\ell \mapsto \bar{v}] \]
(40)

\[ \ell \notin \kappa_0 \land \ell \notin \bar{\kappa}_0 \rightarrow str = \bar{str} \]
(41)

which follows that
\[ \mathcal{H}''[\xi^\ell \mapsto str] \equiv_{\nu, \{\ell | (e_p \in \kappa)} \bar{\mathcal{H}}''[\xi^\ell \mapsto \bar{str}] \]
(42)
Case (DT-IFFalse): Analogous to case (DT-IFTrue).

Case (DT-Trace): \( e \equiv \text{trace}^\ell (e) \). By inversion

\[
\forall e' \not\in \kappa: \mathcal{H}', \rho, \kappa \cdot e' \mapsto e [e' \mapsto e] \downarrow \mathcal{H}' | \bar{v} : \kappa_v
\]

holds. \( \forall \ell \not\in \kappa \) yields

Subcase \( e' \neq \ell \): Claim holds by inversion (DT-Trace).

Subcase \( e' = \ell \): By definition (1) we obtain:

\[
\forall e' \not\in \kappa: \mathcal{H}', \rho, \kappa \cdot e' \mapsto e \downarrow \mathcal{H}' | \bar{v} : \kappa_v
\]

By lemma (1) applied on \( \kappa \cdot e \leq \kappa_v \) the claim holds because:

\[
\ell \in \kappa_v \rightarrow v : \kappa_v \equiv_{\rho, \ell \not\in \kappa} \bar{v} : \kappa_v
\]

\[
\ell \in \kappa_v \rightarrow \mathcal{H}' \equiv_{\rho, \ell \not\in \kappa} \mathcal{H}'
\]

\[\square\]

C. Correctness

First, we state some auxiliary lemmas.

Lemma 6 (Subset consistency on states). \( \mathcal{H} \prec_C \Gamma_0 \) implies \( \mathcal{H} \prec_C \Gamma_0 \sqcup \Gamma_1 \) with \( \Gamma_0 \sqsubseteq \Gamma_0 \sqcup \Gamma_1 \).

Proof of Lemma: By definition (11).

Lemma 7 (Subset consistency on values). \( \omega \prec_C \vartheta_0 \) implies \( \omega \prec_C \vartheta_0 \sqcup \vartheta_1 \) with \( \vartheta_0 \sqsubseteq \vartheta_0 \sqcup \vartheta_1 \).

Proof of Lemma: By definition (11).

Lemma 8 (Consistency on dependencies). \( v : \kappa_v \prec_C \vartheta \land \kappa \prec_C \mathcal{D} \) implies \( v : \kappa_v \cdot \kappa \prec_C \langle \vartheta, \Xi_0, \mathcal{D}_\vartheta \sqcup \mathcal{D} \rangle \).

Proof of Lemma: By definition (11).

Lemma 9 (Property Update). \( \forall s, \theta, \mathcal{L}, \vartheta \mid s \prec_C \theta \) implies \( s \prec_C \theta \sqcup \theta \).

Proof of Lemma: By definition (11).

Proof of Theorem: Suppose that \( \mathcal{H}, \rho, \kappa \vdash e \downarrow \mathcal{H}' | v \) then \( \forall \Gamma, \sigma \) with \( \mathcal{H} \prec_C \Gamma, \rho \prec_C \sigma \) and \( \kappa \prec_C \mathcal{D}_\Gamma \):

\( \Gamma, \sigma \vdash e \downarrow \Gamma' | \vartheta \) with \( \mathcal{H}' \prec_C \Gamma' \) and \( v \prec_C \vartheta \). Proof by induction on the derivation of \( \mathcal{H}, \rho, \kappa \vdash e \downarrow \mathcal{H}' | \omega \) and \( \Gamma, \sigma \vdash e \downarrow \Gamma' | \vartheta \).

Case (DT-Const): \( e \equiv c \). Claim holds because \( c : \kappa \prec_C \langle \varnothing, \emptyset, \mathcal{D} \rangle \).

Case (DT-VAR): \( e \equiv x \). Claim holds because \( \rho \prec_C \sigma \) implies \( \rho(x) \cdot \kappa \prec_C \sigma(x) \sqcup \mathcal{D}_\Gamma \).

Case (DT-OP): \( e \equiv \text{op}(e_0, e_1) \) where by inversion

\[
\Gamma, \sigma \vdash e_0 \downarrow \Gamma' | \vartheta_0
\]

\[
\Gamma', \sigma \vdash e_1 \downarrow \Gamma'' | \vartheta_1
\]

holds. Claim holds by definition (4).

Case (DT-ABS): \( e \equiv \lambda x. e \). Claim holds by definition (11).

Case (DT-New): \( e \equiv \text{new}^\ell e_p \) where by inversion

\[
\Gamma, \sigma \vdash e_p \downarrow \Gamma' | \vartheta
\]

holds. Claim holds by definition (11).

Case (DT-APP): \( e \equiv e_0(e_1) \) where by inversion

\[
\Gamma, \sigma \vdash e_0 \downarrow \Gamma' | \langle \mathcal{L}_0, \Xi_0, \mathcal{D}_0 \rangle
\]

\[
\Gamma', \sigma \vdash e_1 \downarrow \Gamma'' | \vartheta_1
\]

holds. By definition (11) we obtain \( \exists \ell \in \Xi_0 : \xi^\ell \prec_C \ell \).

Subcase \( e \in \Xi_0 : \xi^\ell \prec_C \ell \) Case holds by definition (11) implies \( \mathcal{H}''(\xi^\ell)(\text{str}) \prec_C \vartheta \).

Case (DT-PUT): \( e \equiv e_0[e_1] = e_2 \) where by inversion

\[
\Gamma, \sigma \vdash e_0 \downarrow \Gamma' | \langle \mathcal{L}_0, \Xi_0, \mathcal{D}_0 \rangle
\]

\[
\Gamma', \sigma \vdash e_1 \downarrow \Gamma'' | \langle \text{str}, \Xi_1, \mathcal{D}_1 \rangle
\]

holds. Claim holds because \( \mathcal{H}''(\xi^\ell)(\text{str}) \prec_C \vartheta \).

Case (DT-IFTrue): \( e \equiv \text{if}(e_0) e_1, e_2 \) where by inversion

\[
\Gamma, \sigma \vdash e_0 \downarrow \Gamma' | \langle \mathcal{L}_0, \Xi_0, \mathcal{D}_0 \rangle
\]

\[
\Gamma', \sigma \vdash e_1 \downarrow \Gamma'' | \vartheta_1
\]

holds. By definition (11) we obtain \( \text{true} \prec_C \mathcal{L}_0 \).

Subcase \( \mathcal{L}_0 = \text{true} \). Claim holds by inversion.

Subcase \( \mathcal{L}_0 = \text{bool}.1 \). Claim holds by lemma (6) and (7).

Case (DT-IFFalse): Analogous to case (DT-IFTrue).

Case (DT-Trace): \( e \equiv \text{trace}^\ell (e) \) where by inversion

\[
\Gamma | \mathcal{D} \rightarrow \mathcal{D}_\Gamma \sqcup \ell \mid \sigma \vdash e \downarrow \Gamma' | \vartheta
\]

holds. Claim holds by definition (11).
D. Termination

**Lemma 10** (Monotonicity). *The inference system in Fig. 10 is monotone.*

**Proof of Lemma 10** We have to show monotonicity, that is, for all \( \Gamma, \tilde{\Gamma}, \sigma, \tilde{\sigma}, \vartheta, \tilde{\vartheta}, e \) with \( \Gamma \sqsubseteq \tilde{\Gamma}, \sigma \sqsubseteq \tilde{\sigma} \):

\[
\Gamma, \sigma \vdash e \downarrow \Gamma' | \vartheta \land \tilde{\Gamma}, \tilde{\sigma} \vdash e \downarrow \tilde{\Gamma}' | \tilde{\vartheta} \rightarrow \Gamma' \sqsubseteq \tilde{\Gamma}' \land \vartheta \sqsubseteq \tilde{\vartheta}
\]

(75)

The proof is by induction on \( \Gamma, \sigma \vdash e \downarrow \Gamma' | \vartheta \). It is easy to see that each rule preserves monotonicity if the recursive uses of the judgment do so, too.

**Lemma 11** (Ascending Chain Condition). *For each subject program, the analysis reaches a finite subset of the partially ordered set (Analysis Lattice, \( \sqsubseteq \)).*

**Proof sketch of Lemma 11** Inspection of Fig. 8 shows that the finiteness of many components of the abstract semantic domain depend on the finiteness of the number of labels \( \ell \) in a program. The sole exception is the abstract object, which is a mapping from lattice values to abstract values. Because the String sublattice of Lattice Value has infinitely many values, the domain of abstract objects is infinite and has infinite height.

We argue that each abstract object only uses a finite number of its fields by looking at the possible arguments to get and put properties. Non-\( \top \) values of String are generated by constants in the program and by primitive operations applied to constants. Disregarding recursion, there are only finitely many such constants and operations, so that only a finite sublattice of Abstract Object is exercised.

Adding recursion does not change this picture because the function store subsumes function calls. If a call to a function has a fixed string argument that is different to the argument of previous calls, then the analysis subsumes these arguments to \( \top \) and analyzes the function body with the subsumed argument.

**Proof of Theorem 3** For each \( \Gamma, \sigma, \) and \( e \), there exist \( \Gamma' \) and \( \vartheta \) such that \( \Gamma, \sigma \vdash e \downarrow \Gamma' | \vartheta \). The computation terminates with a fixpoint because all inference rules are monotone (Lemma 10) and the analysis lattice has finite height (Lemma 11).