Static Taint Analysis of Event-driven Scheme Programs

Jonas De Bleser
Vrije Universiteit Brussel
jdeblese@vub.ac.be

Quentin Stiévenart
Vrije Universiteit Brussel
qstieven@vub.ac.be

Jens Nicolay
Vrije Universiteit Brussel
jnicolay@vub.ac.be

Coen De Roover
Vrije Universiteit Brussel
cderoove@vub.ac.be

ABSTRACT

Event-driven programs consist of event listeners that can be registered dynamically with different types of events. The order in which these events are triggered is, however, non-deterministic. This combination of dynamicity and non-determinism renders reasoning about event-driven applications difficult. For example, it is possible that only a particular sequence of events causes certain program behavior to occur. However, manually determining the event sequence from all possibilities is not a feasible solution. Tool support is in order.

We present a static analysis that computes a sound over-approximation of the behavior of an event-driven program. We use this analysis as the foundation for a tool that warns about potential leaks of sensitive information in event-driven Scheme programs. We innovate by presenting developers a regular expression that describes the sequence of events that must be triggered for the leak to occur. We assess precision, recall, and accuracy of the tool's results on a set of benchmark programs that model the essence of security vulnerabilities found in the literature.

CCS Concepts

•Theory of computation → Program analysis; •Security and privacy → *Software security engineering;*

Keywords

Taint Analysis, Abstract Interpretation, Static Program Analysis, Security Vulnerability, Event-driven Programs

1. INTRODUCTION

Event-driven programs are widely used on both client and server side where external events and their corresponding event listeners determine the program behavior. Analyzing such programs is hard because the order in which events occur is non-deterministic and control flow is not explicitly available. These problems negatively impact the ability of tools to detect security vulnerabilities. Among these vulnerabilities, leaks of confidential information and violations of program integrity remain a continuously growing problem [22].

Static taint analysis has been proposed to detect those vulnerabilities [3, 6, 8, 10, 18]. For *event-driven* programs, the state of the art in static taint analysis either completely ignores events or simulates them in every possible order. Another limitation of the state of the art is the lack of information about which event sequences cause a security vulnerability to occur. As a result, there is still little tool support available to precisely detect such defects.

In this work, we present a static taint analysis to compute the flow of values through event-driven programs in which program integrity or confidentiality of information is violated. We model a small event-driven Scheme language, Scheme$_E$, with an event model similar to JavaScript and support for dynamic prototype-based objects. Through *abstract interpretation* [4], we compute an over-approximation of the set of reachable program states. From this set, we identify security vulnerabilities and summarize event sequences causing these vulnerabilities. This paper makes the following contributions:

- We describe an approach to static taint analysis that is able to detect security vulnerabilities in higher-order, event-driven programs.
- We summarize event sequences leading to security vulnerabilities by means of regular expressions. This provides the developer with a description of the events that have to be triggered for a security vulnerability to occur, thereby facilitating the correction of the vulnerability.
- We investigate the use of k-CFA [15] as context sensitivity in event-driven programs. We measure the influence on the results of our analysis and how it affects the number of false positives.

2. TAINT ANALYSIS

A taint analysis is capable of detecting flows of values in a program that violate program integrity or confidentiality of information. Taint analysis defines such data-flow problems in terms of *sources*, *sinks* and *sanitizers*. A *source* is any origin of taint (e.g., user input or private information). A *sink* is any destination where taint is not allowed to flow to (e.g., database query or log utilities). A *sanitizer* converts taint in such a way that it is no longer considered to be

10th European Lisp Symposium ELS'17, Brussels, Belgium

DOI:

tainted (e.g., by stripping tags or by encryption). A security vulnerability occurs whenever taint flows from a source into a sink, without flowing through a sanitizer.

The program in Listing 1 contains a security vulnerability that leaks a password to the screen. The variable `password` is the source, as it is an origin of confidential information. The function `display` exposes its argument to the screen and is therefore a sink. The function `encrypt` is a sanitizer as it converts its argument to an encrypted equivalent that is allowed to be printed on screen. In this example, a leak of confidential information occurs whenever the second branch of the `if`-statement is executed. The goal of a static taint analysis is to detect this violation without executing the program.

```
1  (define password 'secret)
2  (define encrypt (lambda (x) (AES x)))
3  (if (> (length password) 10)
4    (display (encrypt password))
5    (display password))
```

Listing 1: Example of a security vulnerability that leaks the password to the screen.

2.1 Motivating example

To illustrate the problem we are addressing in this paper consider Listing 2, which exemplifies an event-driven program containing a security vulnerability. This example is written in Scheme$_E$ (Section 3) and represents a form that contains a text input listening to the `keypress` event, together with two buttons that respectively listen to the `clear` and `save` events.

```
1  (define o (object))
2  (define key #f)
3
4  (add-event-listener o 'keypress
5    (lambda (e) (set! key (source 'secret))))
6  (add-event-listener o 'clear
7    (lambda (e) (set! key #f)))
8  (add-event-listener o 'save
9    (lambda (e) (sink key)))
```

Listing 2: An event-driven program consisting of a security vulnerability that leaks confidential information.

We define three event listeners (lines 5, 7, and 9) that manipulate or access the variable `key` defined on line 2. The first event listener (line 5) sets the variable `key` to the value `secret`. This value represents confidential information and therefore is tainted. We indicate this by using the special form `source`, which returns its argument but annotates it with a taint flag behind the scenes. The second event listener (line 7) sets variable `key` to the untainted value `#f`. The third event listener (line 9) leaks the contents of `key` (e.g., prints it to the screen). This is indicated by means of the special form `sink`, which raises an error if its argument is tainted.

Each event listener is registered (lines 4, 6, and 8) on object `o` through the special form `add-event-listener`. This special form takes three arguments: the object on which the event listener is registered, the event that triggers the listener to be executed, and the function to execute when the event occurs.

Registration enables the event listeners to become executable, and in Listing 2 some execution orders may lead to security vulnerabilities. This is the case whenever a `keypress` event is immediately followed by a `save` event. By taking into account the execution of event listeners, there now exists a flow between the source (line 5) and the sink (line 9) through variable `key` (line 2). This flow causes the tainted value `secret` to be leaked.

However, information about the execution of event listeners is not explicitly available from the source code, making the flow between event listeners implicit. A naive abstract interpretation of event-driven programs that ignores the execution of event listeners will not detect that there is a flow from and to variable `key` from every event listener. As a result, such an analysis would not detect the leak of confidential information in Listing 2. We tackle this problem in the following sections.

3. AN EVENT-DRIVEN FLAVOR OF SCHEME

We start by introducing Scheme$_E$, the language on which we perform static taint analysis. Scheme$_E$ is a small Scheme language that supports higher-order functions, objects, events, and taint. The syntax of the language is shown in Figure 1.

$$
\begin{aligned}
var \in Var &= \text{a set of identifiers} \\
num \in \mathbb{N} &= \text{a set of numbers} \\
str \in String &= \text{a set of strings} \\
s \in Symbol &= \text{a set of symbols} \\
b \in \mathbb{B} &::= \texttt{\#t} \mid \texttt{\#f} \\
l \in Lambda &= (\texttt{lambda}\ (var)\ e_{body}) \\
e \in Exp &::= var \mid num \mid b \mid l \mid str \mid s \\
&\mid (e_f\ e_{arg}) \\
&\mid (\texttt{set!}\ var\ e_{val}) \\
&\mid (\texttt{if}\ e_{cond}\ e_{cons}\ e_{alt}) \\
&\mid (\texttt{letrec}\ ((var\ e))\ e_{body}) \\
&\mid (\texttt{source}\ e_{val}) \\
&\mid (\texttt{sanitizer}\ e_{val}) \\
&\mid (\texttt{sink}\ e_{val}) \\
&\mid (\texttt{object}) \\
&\mid (\texttt{define-data-property}\ e_{obj}\ s_{\text{name}}\ e_{val}) \\
&\mid (\texttt{define-accessor-property}\ e_{obj}\ s_{\text{name}}\ e_g\ e_s) \\
&\mid (\texttt{get-property}\ e_{obj}\ s_{\text{name}}) \\
&\mid (\texttt{set-property}\ e_{obj}\ s_{\text{name}}\ e_{val}) \\
&\mid (\texttt{delete-property}\ e_{obj}\ s_{\text{name}}) \\
&\mid (\texttt{event}\ s_{event}) \\
&\mid (\texttt{add-event-listener}\ e_{obj}\ s_{event}\ e_{listener}) \\
&\mid (\texttt{remove-event-listener}\ e_{obj}\ s_{event}\ e_{listener}) \\
&\mid (\texttt{dispatch-event}\ e_{obj}\ e_{arg}) \\
&\mid (\texttt{emit}\ e_{obj}\ e_{arg}) \\
&\mid (\texttt{event-queue})
\end{aligned}
$$

Figure 1: Grammar of Scheme$_E$.

3.1 Objects and Properties

Scheme$_E$ supports Javascript-like objects consisting of properties that are maintained in a map relating property names to their respective values. There exist two kinds of properties in JavaScript: data and accessor properties. The former associate property names with values, while the latter associate them with a getter and setter function (i.e., allowing side-effects). The special form `object` instantiates an object without any properties. Accessing a property which does not exist on an object yields `#f` as default value. Properties can be added (`define-(data|accessor)-property`), accessed (`get-property`), deleted (`delete-property`), or modified (`set-property`) at run time.

3.2 Events and Event Listeners

Event listeners (also referred to as event handlers or callbacks) are functions that are registered for a specific event on an object and are executed whenever such an event is dispatched as a result of an action (e.g., clicking a button or pressing a key). In general, event-driven programs do not terminate because they listen for events indefinitely (i.e., they enter an *event loop*). The behavior of event-driven programs is largely determined by the execution of event listeners.

In Scheme$_E$, event listeners are added and removed by the special forms `add-event-listener` and `remove-event-listener`, respectively. New events can be created through the special form `event`, which takes a symbol denoting the event type as argument.

Listing 3 illustrates Scheme$_E$'s support for objects and events. Two properties are defined on object `o`: data property `var` (line 2) and accessor property `result` (line 3) with its getter and setter functions (lines 4 and 5). Accessing property `result` will return the value of `var` multiplied by 2, while setting it will cause `var` to be changed to the given value.

Two event listeners for events `modify` and `resets` are registered on object `o` (lines 7 and 9). The former event listener (line 7) modifies the accessor property `result` to become 2. The latter resets the property `var` so that its value becomes 0. To simulate events directly, we use the special form `dispatch-event`. This special form dispatches events *synchronously* and represents the occurrence of a particular event on an object at that specific moment in time in the program. As a result, the corresponding event listeners on the target object are immediately executed.

First, a `modify` event is dispatched (line 12) and causes the value of `var` to become 2. Accessing `result` on the next line will therefore return 4. Second, a `reset` event is dispatched (line 15) and causes the value of `var` to become 0, as indicated on line 16. Third, the event listener registered for `modify` event is removed on line 18. Any event that occurs after the removal has no effect. This is reflected by the value of `result`, because accessing it on line 20 still results in the value 0 instead of 4. Finally, the property `var` is removed from the object on line 22, and accessing it returns the default value `#f`.

```
1  (define o (object))
2  (define-data-property o 'var 0)
3  (define-accessor-property o 'result
4    (lambda () (* 2 (get-property o 'var)))
5    (lambda (x) (set-property! o 'var x)))
6
7  (define modify (lambda () (set-property o 'result 2)))
8  (add-event-listener o 'modify modify)
9  (define reset (lambda () (set-property o 'var 0)))
10 (add-event-listener o 'reset reset)
11
12 (dispatch-event o (event 'modify)) ; var = 2
13 (get-property o 'result) ; 4
14
15 (dispatch-event o (event 'reset)) ; var = 0
16 (get-property o 'result) ; 0
17
18 (remove-event-listener o 'modify modify)
19 (dispatch-event o (event 'modify)) ; NOP
20 (get-property o 'result) ; 0
21
22 (delete-property o 'var)
23 (get-property o 'var) ; #f
```

Listing 3: Example program illustrating the features of Scheme$_E$.

3.3 Taint

Besides objects and events, Scheme$_E$ features special forms to explicitly define sources (`source`), sinks (`sink`), and sanitizers (`sanitizer`). Listing 4 shows how to use these to define that variable `v` is a source, function `display` is a sink, and `encrypt` is a sanitizer.

```
1  (define v (source 1))
2  (define display (lambda (x) (sink x)))
3  (define encrypt (lambda (x) (sanitizer x)))
```

Listing 4: Defining sources, sinks and sanitizers.

4. STATIC TAINT ANALYSIS OF EVENT-DRIVEN PROGRAMS

We explain how to detect vulnerabilities through abstract interpretation (Section 4.1), how to model such vulnerabilities in event-driven programs (Section 4.2), and how to report them in a user-friendly way through regular expressions describing event sequences (Section 4.3).

4.1 Abstract interpretation in the context of event-driven programs

To statically analyze event-driven programs, we perform abstract interpretation using the technique of Abstracting Abstract Machines (AAM) [21]. From the operational semantics of Scheme$_E$ defined as an abstract machine, we derive an *abstract* version of this semantics as an *abstract abstract machine*. This machine can then be used to perform abstract interpretation of event-driven programs. The result of an abstract interpretation is an *abstract state graph* in which nodes represent program states and edges represent transitions between program states. This graph contains every possible program behavior that can occur during concrete interpretation, but possibly also spurious behavior due to over-approximation. A single abstract state can represent multiple concrete states and is either the evaluation of an expression, the result of an evaluation, or an error state. Figure 2 depicts a fragment of an example abstract state graph.

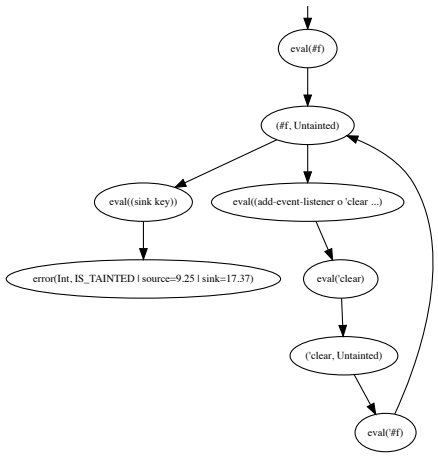

Figure 2: Abstract state graph resulting from abstract interpretation.

To detect security vulnerabilities, the abstract interpretation keeps track of the flow of tainted values. An error

state is generated whenever a tainted value reaches a sink. For example, an error state is represented as the left leaf node in Figure 2. This state provides information about the tainted value such as the type of the value (i.e., `Int`), the line and column number of the source (i.e., `source=9.25`) and the sink (i.e., `sink=17.37`) involved in the vulnerability, together with the precision with which the vulnerability was detected. If the analysis detects a taint violation with full precision, the error `IS_TAINTED` is produced. If it detects that a taint violation *may* occur, the error `MAYBE_TAINTED` is produced instead.

As illustrated in Section 2.1, naively performing abstract interpretation of event-driven programs may miss security vulnerabilities if events are not taken into account. By ignoring events, the resulting abstract state graph does not contain behaviors related to the execution of event listeners. Such a graph is an *unsound* approximation of the program behavior and may not contain every security vulnerability present in the program.

In order to obtain a sound static analysis for event driven programs, it is crucial to execute event listeners. A sound static analysis ensures that a given event-driven program is free of vulnerabilities under *any* input and sequence of events if the analysis detects no possible vulnerability during abstract interpretation.

We describe our approach to simulate events in the next section.

4.2 Simulating events

Scheme$_E$ provides the special form `emit` to *asynchronously* trigger an event. As opposed to `dispatch-event`, any corresponding event listeners are not directly executed. Instead, the emitted event is scheduled in the global *event queue* and program execution continues right after the call to `emit`. At the end of the program, when the call stack is empty, `(event-queue)` is used to initiate the event loop. The event loop continually extracts a single event from the queue and executes its corresponding registered event listeners in registration order.

Because the registered event listeners are executed according to which event is consumed from the event queue during abstract interpretation, the final abstract state graph includes the behavior of the event listeners. This approach enables us to detect security vulnerabilities, including the ones that occur as a result of program flow through event listeners. To keep track of which event has been executed, we annotate the edges of the abstract graph with information about the triggered event. This information is used in a later stage (Section 4.3) to generate regular expressions that describe event sequences leading to a particular security vulnerability. We call the resulting graph an *abstract event graph*.

Because the order in which events are triggered is largely non-deterministic, a naive approach is to assume that every event can be triggered in any order at any time. However, it is computationally expensive to explore the whole event space for non-trivial programs in this manner, and it may result in many false positives. This can be mitigated partially with domain knowledge about the semantics of the program and how events occur. Madsen et al. [11] presents a modeling approach to support events and provide the abstract semantics of an event queue. We follow their approach in order to reduce the search space by explicitly emitting events in the program.

We extend our motivational example (Listing 2) with a model that indicates which events can occur and when they can occur. Listing 5 depicts such a model where the events `clear` and `save` can never occur at the start of the program. That is because buttons registered for these events are disabled as long as there has not been any `keypress` event. We specify this behavior by only emitting a `keypress` event at line 18, before calling `event-queue`. Whenever a `keypress` event has been triggered, any other type of event can occur. We model this by emitting every event (lines 6–8) in the event listener registered for `keypress`. We also model that a `save` event can never occur after a `clear` event. This is specified by only emitting `keypress` and `clear` at lines 12 and 13 in the event listener registered for `clear`. Finally, we model the fact that a `save` event implies termination of the program by not emitting any event from the listener at line 16. We consider termination to be the disappearance of the form once the save button is pressed. While explicit event modeling requires some effort, it reduces the search space and avoids exploring spurious event sequences.

```
1  (define o (object))
2  (define key #f)
3
4  (add-event-listener o 'keypress
5    (lambda ()
6      (emit o (event 'keypress))
7      (emit o (event 'clear))
8      (emit o (event 'save))
9      (set! key (source 'secret))))
10 (add-event-listener o 'clear
11   (lambda ()
12     (emit o (event 'keypress))
13     (emit o (event 'clear))
14     (set! key #f)))
15 (add-event-listener o 'save
16   (lambda () (sink key)))
17
18 (emit o (event 'keypress))
19 (event-queue)
```

Listing 5: An event-driven program consisting of explicitly modeled event sequences and a leak of confidential information.

While detecting security vulnerabilities in event-driven programs is important, knowing *why* and *how* they occur is at least as important. Manually inspecting the abstract event graph for event sequences of interest is not an option, given the complexity of event-driven programs and their resulting event graphs. We tackle this problem in the next section.

4.3 Computing event sequences

Manually deriving event sequences that lead to security vulnerabilities is not trivial. This is because programs typically consist of many events, as well as many sources, sanitizers, and sinks. We address this problem by automatically generating regular expressions describing the sequence of events required for a security vulnerability to occur. Event sequences provide valuable information to developers detecting and fixing these vulnerabilities.

We start from the observation that the abstract event graph is equivalent to a non-deterministic finite automaton with ϵ-transitions (ϵ-NFA). This because each state can have zero, one, or more successor states, and non-annotated edges in the graph (i.e., control-flow not induced by triggering of events) correspond to ϵ-transitions.

This automaton $(Q, \Sigma, \delta, q_0, F)$ consists of the set of abstract states Q, a transition function $\delta(q, a)$ where $q \in Q$ and $a \in \Sigma$ is either an event or ϵ. The initial state q_0 is the root state of the abstract state graph, and the set of final states F includes every error state.

This observation enables us to convert the abstract event graph to regular expressions in three steps:

1. Convert the ϵ-NFA to an NFA by calculating the ϵ-closure for each state.
2. Convert the NFA to a minimal deterministic finite automaton (DFA).
3. Convert the DFA to regular expressions for every combination of source and sink.

Conversion to NFA.

The function $ECLOSURE(Q) = \{s \mid q \in Q \wedge s \in \delta(q, \epsilon)\}$ calculates the ϵ-closure for each state of the automaton. Given this information, we can eliminate all ϵ-transitions because they do not contribute to the final regular expression. This step results in an ϵ-free NFA and reduces the number of states because most transitions are indeed ϵ-transitions.

Conversion to minimal DFA.

Any NFA can be converted into its corresponding unique minimal DFA [16]. We opt for Brzozowski's algorithm to perform this conversion because it outperforms other algorithms in many cases [2], despite its exponential character. This algorithm minimizes an ϵ-free NFA into a minimal DFA where both automatons accept the same language L. It does so by reversing the directions of the transitions in an NFA (*rev*), and then converting it into an equivalent DFA that accepts the reverse language L^R using the powerset construction method (*dfa*). The process is repeated a second time to obtain a minimalistic DFA that accepts language L. This algorithm is performed by the function *minimize*.

$$minimize(fa) = dfa(rev(dfa(rev(fa))))$$

Extracting regular expressions.

Given a minimal DFA, we can convert it into a regular expression using several methods [14]. We opt for the *transitive closure method* because of its systematic characteristic. First, an $n \times n \times n$ matrix from a given DFA $\langle Q, \Sigma, \delta, q_0, F\rangle$ with n states is built. We define $R^k_{i,j}$ as the regular expression for the words generated by traversing the DFA from state q_i to q_j while using intermediate states $\{q_1 \dots q_k\}$. We compute this regular expression in every iteration from 1 to k as follows:

$$R^k_{i,j} = R^{k-1}_{i,j} + R^{k-1}_{i,k} \ . \ {R^{k-1}_{k,k}}^* \ . \ R^{k-1}_{k,j}$$

The final outcome $R^k_{i,j}$ is the regular expression that describes all the event sequences that lead to a particular security vulnerability.

We apply these steps to the example described in Section 4.2 and show the resulting regular expressions below. The first regular expression (1) is generated from the program using the naive approach in which every possible event ordering is explored. The second regular expression (2) is generated by means of the model using explicitly modeled event sequences (Listing 5). We abbreviate the events to their first letter for brevity. A + indicates choice, . indicates concatenation, and * indicates repetition (Kleene star operator). From both expressions, it is clear that the event sequence leading to the leak *ends* with a `keypress` event followed by a `save` event.

Figure 3 depicts the second regular expression as an automaton.

$$(k + ((c + s).(c + s)^*.k)).(k + (c.((c + s)^*.k)))^*.s \quad (1)$$
$$(k.k^*.c.(c + (k.k^*.c + c^*.k.k^*.c))^*.(c^* + k)^*)^*.k.k^*.s \quad (2)$$

Even in this simple example the generated event sequences are rather long and complex. While all possibilities are important (e.g. when multiple unique event sequences may lead to a leak), we deem the shortest possible event sequence to be the most important one in order to patch the security vulnerability. In this example, this is the sequence `keypress.save`.

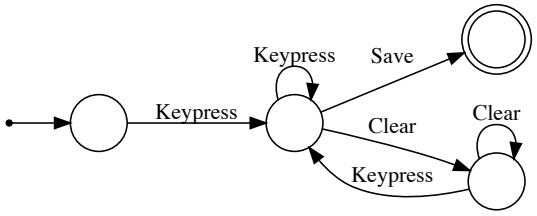

Figure 3: Finite state automaton of the regular expression that represents event sequences leading to the security vulnerability in Listing 5.

5. IMPLEMENTATION

We implemented the static taint analysis for event-driven programs discussed in this paper as a proof of concep[1]. We make use of the modular framework Scala-AM [17] to perform static analysis based on systematic abstraction of abstract machines (AAM) [21]. The implementation supports Scheme$_E$, our small Scheme language with support for prototype-based objects, events, and taint that we described in Section 3. We incorporated an existing library for the manipulation of finite state automata [13] to obtain a minimal DFA from an abstract event graph by computing the ϵ-closure and applying the DFA minimization as described in Section 4.3.

Abstract counting [12] is enabled by default in our implementation. This to improve precision by avoiding unnecessary *joining* of values when it is safe to do so. Under abstraction, the abstract machine represents the *unbounded* heap memory by a map that relates a *finite* number of addresses to values. This results in possibly different values being allocated at the same *abstract* addresses. Such values are then *joined* in order to remain over-approximative in the interpretation. Suppose variable `x` is allocated at address a and represents a tainted value. Allocating a variable `y` representing an untainted value at the same address a will cause the values of `x` and `y` to join (i.e., to merge), so that the value at address a now *may* be tainted. This over-approximated

[1] https://github.com/jonas-db/aam-taint-analysis

value is then used in the remainder of the interpretation and may lead to false positives.

6. PRELIMINARY EXPERIMENTS

To measure the applicability of our approach, we extracted synthetic benchmarks from larger programs. These benchmarks are described in Table 1. We include multiple benchmarks that contain no security violation to assess to which extent our approach produces false positives. These benchmarks therefore enable us to determine the precision of our analysis. We investigate the increase of precision and accuracy that follows from the use of call-site-sensitivity (k-CFA) [15] as context sensitivity for the analysis. This context sensitivity is well-suited for programs with functions calls, and we observed that it is not uncommon for event listeners to call auxiliary functions.

The results of our experiments are shown in Table 1. The time it takes to produce these results is at most 1.3 seconds. Our analysis did not have any false negatives because it is sound, and thus has a recall of 100%. For each benchmark, we indicate whether the analysis detected a leak with maximal precision (*Must*) or not (*May*), or whether it detected no leak (/). We also indicate whether the results are correct (✓) or not (✗). For brevity, we only provide the shortest event sequence instead of the full regular expression. We conclude from this table that increasing the context sensitivity (i.e., increasing the value of k) results in less false positives in our experiments, while also increasing the precision and accuracy with which leaks are detected in the two first benchmarks. However, the analysis was unable to detect that benchmark programs `manylst` and `delprop` do *not* contain a leak. These false positives are a consequence of over-approximations due to abstract interpretation, and are a common side-effect of static analysis in general. However, the developer can be certain of the absence of security vulnerabilities by *only* verifying (e.g., simulating with increased polyvariance or manual inspection) the generated set of event sequences, as opposed to *every* possible combination of events. We discuss `lstfunc` and `rmlst` to understand how context sensitivity can contribute to less false positives and more precise results.

Context sensitivity and its influence on results.

Listing 6 shows the example `lstfunc` in which we model the scenario where event `a` is emitted, followed by event `b`. Each event listener calls the function `f`, which is a known sink. The first call to `f` on line 7 with untainted value 2 does not result in a security vulnerability. The second call on line 4, however, does result in a security leak.

```
1  (define window (object))
2  (define f (lambda (x) (sink x)))
3  (add-event-listener window 'b
4    (lambda (e) (f (source #f))))
5  (add-event-listener window 'a
6    (lambda (e)
7      (f 2)
8      (emit window (event 'b))))
9  (emit window (event 'a))
10 (event-queue)
```

Listing 6: `lstfunc` example, containing a security vulnerability.

Our analysis detects this leak but not with full precision. This because the parameter `x` is allocated twice to the same address, which causes the untainted value 2 and the tainted value `#f` to join. Hence, our monovariant analysis ($k = 0$) detects that `x` *may* be tainted. On the other hand, our polyvariant analysis with $k \geqslant 1$ is able to distinguish between the two calls to `f` and detects the security leak with full precision.

Listing 7 shows `rmlst` where a finite event sequence `a.b.c.d` is modeled. The example consists of multiple event listeners, each registered for one specific event. The problematic event listener defined on line 7 is registered for the event `d` on line 14 and could cause a leak if the property `p` of `oa` is tainted. However, this listener is removed on line 25 before any vulnerability can occur (i.e., this example does *not* contain a leak).

Without context sensitivity (0-CFA), a leak is detected by the analysis. The reason is related to the allocation and subsequent *joining* of objects, even though this occurs in two different event listeners (line 10 and 18). Two objects are joined whenever they are allocated at the same address, and a new abstract object is created that represents all properties and all registered event listeners of both objects.

Function `f` (line 4) calls function `g` (line 3) which allocates a new object. With 1-CFA, the analysis can differentiate between the two different calls to `f` on line 11 and 19, but not between the calls to `g` performed by `f`. Because of this, the allocation of the second object (by means of calling `f` on line 19) will join with the previously allocated object (by means of calling `f` on line 11). The definition of the tainted property `p` on line 12 will thus apply to both objects. Note that `o1` is aliased by `oa` which means that, due to joining, `oa` will point to *both* objects.

Whenever the event `c` occurs, the event listener is removed from the object `oa`. This is not the case when the object points to an address that represents multiple objects because removing it would be unsound since we do not know which event listener was registered to which object. As a result, emitting the event `d` on line 25 causes the event listener on line 7 to execute. the tainted property `p` of object `oa` reaches the sink.

```
1  (define oa #f)
2  (define window (object))
3  (define (g) (object))
4  (define (f) (g))
5
6  (define listener
7    (lambda (e) (sink (get-property oa 'p))))
8
9  (add-event-listener window 'a
10   (lambda (e)
11     (let ((o1 (f)))
12       (set-property o1 'p (source 1))
13       (add-event-listener o1 'd listener)
14       (set! oa o1)
15       (emit window (event 'b)))))
16
17 (add-event-listener window 'b
18   (lambda (e)
19     (f)
20     (emit window (event 'c))))
21
22 (add-event-listener window 'c
23   (lambda (e)
24     (remove-event-listener oa 'd listener)
25     (dispatch-event oa (event 'd))))
26
27 (emit window (event 'a))
28 (event-queue)
```

Listing 7: `rmlst` example, where no leak is present due to the removal of an event listener.

						0-CFA		1-CFA		2-CFA	
Name	Description	LOC	Leaks	Listeners	Emits	Result	Regex	Result	Regex	Result	Regex
`lstfunc`	Event listeners call the same function	9	1	2	2	May ✓	ba	Must ✓	ba	Must ✓	ba
`samelst`	Same event listener for different events	22	1	2	2	May ✓	ba	Must ✓	ba	Must ✓	ba
`nestedlst`	Event listener calls nested function	14	0	2	2	May ✗	ba	/ ✓	/	/ ✓	/
`objjoin`	Event listeners call factory method	28	0	4	4	May ✗	abcd	May ✗	abcd	/ ✓	/
`delprop`	Event listeners delete object property	21	0	2	2	May ✗	aa	May ✗	aa	May ✗	aa
`funccalls`	Nested registration of event listeners	11	0	2	2	May ✗	ab	/ ✓	/	/ ✓	/
`rmlst`	Removing an event listener	26	0	4	4	May ✗	abcd	May ✗	abcd	/ ✓	/
`manylst`	Multiple event listeners for an event	16	0	2	3	May ✗	ac	May ✗	ac	May ✗	ac
					Precision	25%		33%		50%	
					Recall	100%		100%		100%	
					Accuracy	25%		50%		75%	

Table 1: Precision, recall, and accuracy with k-CFA. *Result* describes whether the result is detected with full precision (*Must*) or not (*May*). It also indicates whether the result is correct (✓) or not (✗). *Regex* is the shortest event sequence that leads to the security vulnerability. The use of / means that no regular expressions are generated and thus no vulnerabilities were found. The gray areas indicate correct results and shows how precision increases with increased context sensitivity.

With 2-CFA, the analysis can distinguish between the two calls to `g`, and it correctly detects that there are no leaks.

7. RELATED WORK

To the best of our knowledge there exists no precise static taint analysis to detect security vulnerabilities in the context of higher-order event-driven programs written in a dynamically-typed language. Arzt et al. [3] present FlowDroid, a static taint analysis for Android based on the static analysis framework SOOT [20]. It is aware of the event-driven lifecycle of Android and user-defined event listeners. However, their approach does not support higher-order functions. Jovanovic et al. [8] introduce Pixy which aims to detect cross-site scripting vulnerabilities (XSS) in PHP 4. However, they do not support objects, events or higher-order functions. Guarnieri et al. [6] present Actarus, a blended (i.e., a combination of static and dynamic) taint analysis for JavaScript to detect client-side vulnerabilities. Their approach is based on the static analysis framework WALA [5]. However, being a blended analysis, it depends on run-time information. Tripp et al. [18] present TAJ, a static taint analysis for Java 6 without support for higher-order functions. It targets four security vulnerabilities in web applications, including cross-site scripting (XSS), command injection, malicious file executions and information leakage.

There is existing work related to static analysis of event-driven JavaScript programs using abstract interpretation. Liang and Might [10] present a static taint analysis for Python using abstract interpretation. However, their analysis does not support event-driven programs. Another difference is that we opt to maintain a product of abstract values and taint in a single abstract store instead of using two separate stores. Tripp et al. [19] present Andromeda, a demand-driven analysis tool for Java, JavaScript and .NET that has been successfully used in a commercial product, but has no support for event-driven programs. Jensen et al. [7] present TAJS which is a tool to detect type-related and data-flow related programming errors in event-driven JavaScript web applications. It is capable of detecting the absence of object properties or unreachable code and has support for the HTML DOM. While the tool has support for events, it does not track each individual event separately. Their approach consists of merging events in several categories such as load, mouse, keyboard, etc. This decision is a trade-off in terms of performance but leads to less precise event information. Kashyap et al. [9] present JSAI, a tool with support for the HTML DOM and events. We notice that both TAJS and JSAI simulate an event queue where event listeners are executed in every possible order. To avoid exploring the complete search space of events, Madsen et al. [11] present a modeling approach to support events. They do not implement a taint analysis but rather focus on detecting dead event listeners, dead emits and mismatched synchronous and asynchronous calls in Node.js. To model event-driven programs, they require the developer to explicitly place emit statements in the program. The proposed abstract event queue will then be filled with these events and enables the tool to explore the flow of events. While this approach leads to a smaller search space, it requires some knowledge about the semantics of the program. Nevertheless, this work inspired our implementation of an event queue used in our abstract machine.

8. CONCLUSION AND FUTURE WORK

In this work, we outline an approach to statically detect security vulnerabilities in event-driven Scheme programs. We propose the event-driven language Scheme$_E$ and use abstract interpretation as a technique to compute an over-approximation of the program's behavior. We use a three-step process to generate regular expressions that describe event sequences that lead to a particular security vulnerability. Event sequences provide valuable information to developers detecting and fixing these vulnerabilities. We also investigate the effect of context sensitivity, more precisely k-CFA, on the results of the analysis. Our results show that our technique can detect security vulnerabilities in event-driven programs and that higher precision can be achieved with increased context sensitivity.

As future work, we envision to investigate techniques that are able to avoid exploration of spurious event sequences. We also want to implement our technique for JavaScript. We deem this language support to be a continuation of our work because we closely followed the semantics of objects and events in JavaScript. For larger programs, we foresee that the size of the abstract state graph grows rapidly because many event sequences have to be explored. The size could be reduced by applying a macro-step evaluation [1] (i.e., a single node per event listener that may consist of multiple states) instead of a small-step evaluation (i.e., a single node per state). Another improvement can be to avoid the

exploration of all permutations of event listeners. However, according to Madsen et al. [11] it is uncommon for a single object to have multiple event listeners registered for the same event. Madsen et al. [11] also proposes two context sensitivities specific to event-driven programs. We will implement these as future work and investigate whether one of the context sensitivities (or a combination thereof) can further improve the results of the analysis. The concept of event bubbling and event capturing is another problem that affects program security, but requires a hierarchical relationship between objects.

Although our approach is able to detect security vulnerabilities in event-driven programs, the non-deterministic behavior of events remains a computational challenge that influences the ability to detect vulnerabilities. Techniques are needed to reduce the search space and to further improve the precision of taint analysis in event-driven programs. Our work provides foundations toward this goal.

Acknowledgements

Quentin Stiévenart is funded by the *GRAVE* project of the Research Foundation - Flanders (FWO). Jens Nicolay is funded by the SeCloud project sponsored by Innoviris, the Brussels Institute for Research and Innovation.

References

[1] Gul A Agha, Ian A Mason, Scott F Smith, and Carolyn L Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(01):1–72, 1997.

[2] Marco Almeida, Nelma Moreira, and Rogério Reis. On the performance of automata minimization algorithms. In *Proceedings of the 4th Conference on Computation in Europe: Logic and Theory of Algorithms*, pages 3–14, 2007.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Notices*, volume 49, pages 259–269. ACM, 2014.

[4] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[5] Stephen Fink and Julian Dolby. WALA – T.J. Watson libraries for analysis., 2006. `http://wala.sourceforge.net`.

[6] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the world wide web from vulnerable javascript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 177–187. ACM, 2011.

[7] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proceedings of 16th International Static Analysis Symposium (SAS)*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.

[8] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6–pp. IEEE, 2006.

[9] Vineeth Kashyap, Kyle Dewey, Ethan A Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. Jsai: A static analysis platform for javascript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 121–132. ACM, 2014.

[10] Shuying Liang and Matthew Might. Hash-flow taint analysis of higher-order programs. In *Proceedings of the 7th Workshop on Programming Languages and Analysis for Security*, page 8. ACM, 2012.

[11] Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static analysis of event-driven node. js javascript applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 505–519. ACM, 2015.

[12] Matthew Might and Olin Shivers. Improving flow analyses via γcfa: abstract garbage collection and counting. In *ACM SIGPLAN Notices*, volume 41, pages 13–25. ACM, 2006.

[13] Anders Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2010. `http://www.brics.dk/automaton/`.

[14] Christoph Neumann. Converting deterministic finite automata to regular expressions, 2005. `http://liacs.leidenuniv.nl/ bonsanguem-m/FI2/DFA_to_RE.pdf`.

[15] Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie-Mellon University, 1991.

[16] Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[17] Quentin Stiévenart, Maarten Vandercammen, J Nicolay, W De Meuter, and C De Roover. Scala-am: A modular static analysis framework. In *Proceedings of the 16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM*, volume 16, 2016.

[18] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. *ACM Sigplan Notices*, 44 (6):87–97, 2009.

[19] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *International Conference on Fundamental Approaches to Software Engineering*, pages 210–225. Springer, 2013.

[20] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot – a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[21] David Van Horn and Matthew Might. Abstracting abstract machines. In *ACM Sigplan Notices*, volume 45, pages 51–62. ACM, 2010.

[22] Dave Wichers. OWASP top ten project – a list of the 10 most critical web application security risks, 2013. `https://www.owasp.org/index.php`.