Automatic Detection of Information Leakage Vulnerabilities in Browser Extensions

Rui Zhao
University of Colorado
Colorado Springs
rzhao@uccs.edu

Chuan Yue
University of Colorado
Colorado Springs
cyue@uccs.edu

Qing Yi
University of Colorado
Colorado Springs
qyi@uccs.edu

ABSTRACT

A large number of extensions exist in browser vendors' online stores for millions of users to download and use. Many of those extensions process sensitive information from user inputs and webpages; however, it remains a big question whether those extensions may accidentally leak such sensitive information out of the browsers without protection. In this paper, we present a framework, LvDetector, that combines static and dynamic program analysis techniques for automatic detection of information leakage vulnerabilities in legitimate browser extensions. Extension developers can use LvDetector to locate and fix the vulnerabilities in their code; browser vendors can use LvDetector to decide whether the corresponding extensions can be hosted in their online stores; advanced users can also use LvDetector to determine if certain extensions are safe to use. The design of LvDetector is not bound to specific browsers or JavaScript engines, and can adopt other program analysis techniques. We implemented LvDetector and evaluated it on 28 popular Firefox and Google Chrome extensions. LvDetector identified 18 previously unknown information leakage vulnerabilities in 13 extensions with a 87% accuracy rate. The evaluation results and the feedback to our responsible disclosure demonstrate that LvDetector is useful and effective.

Categories and Subject Descriptors

D.4.6 [**Software**]: Operating Systems—*Security and Protection.*; H.4.3 [**Information Systems**]: Communications Applications—*Information browsers*

Keywords

Web browser extension; JavaScript; Vulnerability analysis

1. INTRODUCTION

Popular web browsers all support extension mechanisms to help third-party developers extend the functionality of browsers and improve user experience. A large number of extensions exist in browser vendors' online stores for millions of users to download and use. Quite often, those extensions are written in JavaScript; they have higher privileges than regular webpages do, thus have become a popular vector for performing web-based attacks [1, 2].

Because many extensions have security vulnerabilities [1, 2, 3, 4, 11, 18, 22, 24] and some extensions are even malicious, browser vendors have taken stricter measures to control the extensions that can be installed on browsers. For example, Google bans Windows version chrome extensions found outside the Chrome Web Store starting from January 2014, and inspects the extensions in the Chrome Web Store to exclude the malicious ones.

Researchers have extensively studied privilege escalation related vulnerabilities in JavaScript-based extensions and shown that a lack of sufficient security knowledge in developers is one of the main reasons for many vulnerabilities [1, 2, 3, 18, 24]. However, an often overlooked problem is that extensions may accidentally leak users' sensitive information out of the browsers without protection.

Many browser extensions process sensitive information coming from either user inputs or webpages. For example, some extensions save users' website passwords, some extensions remember users' shopping preferences, and some extensions manage users' bookmarks. If such sensitive information is leaked out of the browser without protection, it can be used by unauthorized parties to illegally access users' online accounts, steal their online identities, or track their online behaviors. Therefore, banning extensions that may leak users' sensitive information is also necessary and important.

Yet detecting information leakage in JavaScript-based web browser extensions is especially challenging. One source of the challenges is JavaScript itself, an interpreted prototype-based object-oriented programming language with just-in-time code loading/generation [27, 28, 29, 41] and dynamic uses of functions, fields and prototypes [6, 21, 26]. The other source of the challenges is the highly complex interactions among browser extensions, internal components of browsers, and webpages [2, 3, 18, 24]. Only a handful of solutions have been proposed to address the problem of information leakage in JavaScript-based browser extensions [4, 11, 22]; however, they took either pure dynamic approaches or pure static approaches, thus suffering from many limitations (Section 5).

In this paper, we present a framework, LvDetector, that combines static and dynamic program analysis techniques for automatic detection of information leakage vulnerabilities in legitimate browser extensions. LvDetector focuses on legitimate browser extensions because lots of them are

Copyright is held by the International World Wide Web Conference Committee (IW3C2). IW3C2 reserves the right to provide a hyperlink to the author's site if the Material is used in electronic media.

WWW 2015, May 18–22, 2015, Florence, Italy.
ACM 978-1-4503-3469-3/15/05.
http://dx.doi.org/10.1145/2736277.2741134.

used by millions of users [14, 5], thus the impact level of their information leakage vulnerabilities is high. LvDetector does not aim to be sound at the whole program level (Section 3.1); it aims to be a practical and accurate utility by (1) using a dynamic scenario-driven call graph construction scheme to reduce the overall false positives in the analysis as much as possible, and (2) using static analysis based on each dynamically constructed call graph to extensively analyze the corresponding scenario. Extension developers can use LvDetector to locate and fix the vulnerabilities in their code; browser vendors can use LvDetector to decide whether the corresponding extensions can be hosted in their online stores; advanced users can also use LvDetector to determine if certain extensions are safe to use. Note that detecting potentially malicious code or intentional vulnerabilities is out of the scope of the current LvDetector framework.

The design of LvDetector is not bound to specific web browsers or JavaScript engines, and can adopt other program analysis techniques. We implemented LvDetector in Java and evaluated it on 28 popular Firefox and Google Chrome extensions. LvDetector identified 18 previously unknown information leakage vulnerabilities in 13 extensions with a 87% accuracy rate. The evaluation results and the feedback to our responsible disclosure demonstrate that LvDetector is useful and effective.

The main contributions of this work include: (1) a dynamic scenario-driven call graph construction scheme, (2) a formulation of transitive relations and function/program-level static analysis algorithms for effective exploration of information flow paths in browser extensions, (3) a unique framework that combines static and dynamic program analysis techniques for automatic detection of information leakage vulnerabilities in JavaScript-based browser extensions, and (4) an effectiveness evaluation of LvDetector.

The rest of the paper is organized as follows. Section 2 uses an example to illustrate the vulnerability analysis problem targeted by this paper. Section 3 presents the details of the LvDetector framework. Section 4 evaluates the effectiveness of LvDetector. Section 5 discusses the related work. Finally, Section 6 makes a conclusion.

2. MOTIVATING EXAMPLE

Many legitimate browser extensions process sensitive information coming from either user inputs or webpages. If such sensitive information is leaked out of the browser without protection, it can be used by unauthorized parties to illegally access users' online accounts, steal their online identities, or track their online behaviors. Figure 1 depicts a code excerpt of a real example browser extension that manages users' website passwords. In the code excerpt, this extension obtains the website password and username of a user in the save() function, encodes the password and username in the encode() function, and sends the encoded string to the remote server of the extension through the send() method of an XMLHttpRequest object in the post() function.

When this example extension was submitted to the extension web store of a browser vendor, the developers claimed that they cannot know users' website passwords. The browser vendor wants to verify this claim and identify potential information leakage vulnerabilities in this extension, but manually inspecting about 26,000 lines of code in this extension is time and effort consuming. The browser vendor can use LvDetector to easily perform this inspection task in three

```
function save() {
  var pwd = document.getElementById("pwd").value;
  var usr = document.getElementById("usr").value;
  var str = encode(pwd, usr);
  post(str);
}
...
function encode(pwd, usr) {
  return encodeURI(pwd) + encodeURI(usr);
}
...
function post(content) {
  var req = new XMLHttpRequest();
  var url = "https://www.remoteserver.com/"
  req.open("POST", url);
  req.send(content);
}
```

Figure 1: Code excerpt of a real example extension

steps: (1) runs LvDetector to instrument this extension, (2) executes a website password saving scenario using the instrumented extension, and (3) runs LvDetector to automatically detect potential information leakage vulnerabilities and generate a report. The browser vendor can also directly perform the third step by using the execution traces supplied by other LvDetector users (Section 3.2.1).

The generated vulnerability report contains a vulnerable information flow record: the website password assigned to the "pwd" variable in the save() function is propagated through the encode() function to the "content" variable in the post() function, and is leaked out in a send() method call. All the detailed operations in this vulnerable information flow are also provided in the report.

A user's website password should only be known by the user and the corresponding website - sending the unprotected website password to the remote server of the extension allows server-side attackers to directly obtain the user's website login information. LvDetector correctly identifies this vulnerability, providing evidence for the browser vendor to disprove the extension developers' claim.

3. OVERALL FRAMEWORK

Our key objective is to design LvDetector as a framework that can be easily used by ***analysts*** (extension developers, browser vendors, or advanced users) to automatically detect information leakage vulnerabilities in browser extensions.

3.1 Design Overview and Rationale

The overall workflow of LvDetector can be organized into three phases as shown in Figure 2.

The first phase, *call graph and variable use analysis*, starts with an *instrumentation* component that takes a browser extension as the input and instruments the extension for collecting execution traces. The *call graph analysis* component collects the traces generated from each scenario-driven execution of the instrumented extension to build a call graph. Meanwhile, the *SSA builder* component builds the SSA IR (Static Single Assignment form Intermediate Representation [10]) of each function in the extension source code, and the SSA IRs [10] in turn are fed into the *variable use analysis* component to generate variable use graphs. The variable use analysis component will automatically identify (1) commonly used cryptographic functions (e.g., AES encryption/decryption and SHA hash functions), (2) ***source variables*** that accept values from either user inputs or webpages

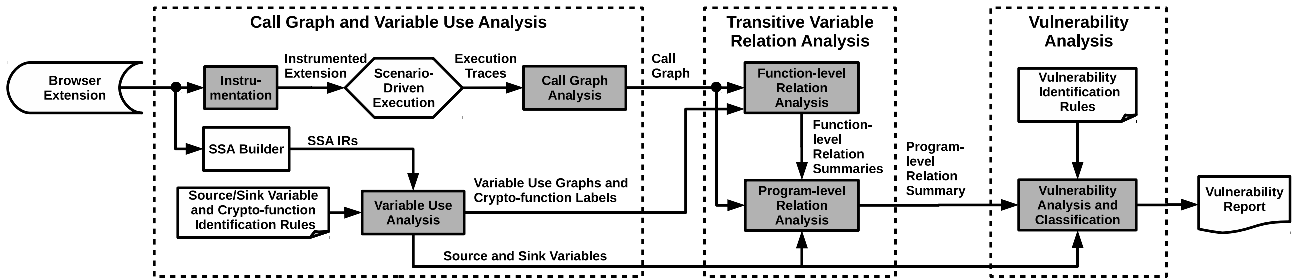


Figure 2: The overall workflow of the LvDetector framework (the shaded components are ours)

(e.g., through form fields), and (3) ***sink variables*** that contain values either saved to the local disk (e.g., through the setItem() method call of the localStorage object in HTML5) or sent across the network (e.g., through the send() method call of an XMLHttpRequest object). Sink variables are extracted from the sink statements, which are either common or specific to different browsers and are XMLHttpRequest, window object, SQLite database, file, local storage, bookmark, and password manager related statements. These criteria for identifying standard/nonstandard cryptographic functions, source variables, and sink variables in Google Chrome and Firefox extensions are included in a rule file.

The second phase, *transitive variable relation analysis*, computes a transitive summary of the relations among each pair of the source and sink variables. Specifically, the *function-level relation analysis* component iteratively computes a function-level relation summary for each function based on its variable use graph, the call graph, and the labeled cryptographic functions; the *program-level relation analysis* component computes the program-level relation summary based on the call graph and the function-level relation summaries.

Finally, the third phase, *vulnerability analysis*, identifies all the potential vulnerable information flows that may lead to sensitive information leakage. It analyzes vulnerabilities based on the program-level relation summary and the source-sink variable pairs, and generates an intuitive report with a list of classified vulnerability records for each scenario-driven execution.

The overall workflow takes a hybrid approach to analyze JavaScript browser extensions. It uses scenario-driven execution traces to dynamically and accurately construct a call graph; it then statically performs variable use analysis and transitive variable relation analysis based on SSA IR [10] to summarize the overall information flows among variables both within a single function and across function boundaries. The dynamic aspects of our approach accurately capture intricate across-function-boundary information flows that often occur in JavaScript extensions due to reflection, function objects, event handlers, asynchronous calls, DOM interactions, and so on. The static aspects of our approach extensively extract both explicit and implicit information flows within each function. This hybrid approach is superior to pure static approaches by effectively reducing false positives in the construction of call graphs [26], which are often the foundation of the overall program analysis. This approach is not bound to specific web browsers or JavaScript engines; it is superior to pure dynamic approaches by avoiding users' or browsers' responses to runtime alerts, incomplete information flow exploration, runtime overhead, and browser-specific instrumentation [1, 11, 31, 36].

Most analysis tools for statically typed programming languages choose to be sound rather than complete. However, due to the complexity and dynamic features of the JavaScript language (Section 1), achieving soundness in the static analysis of the full JavaScript language is very difficult or impossible [16, 25, 26]. Meanwhile, LvDetector bases its static analysis on the call graphs constructed from the scenario-driven execution traces, which may not cover all the possible execution paths in the program. Due to these reasons, LvDetector does not aim to be sound at the whole program level; it aims to be a practical and accurate utility. Note that scenario-driven execution traces can be more extensively collected as discussed in Section 3.2.1.

3.2 Call Graph and Variable Use Analysis

In this phase, LvDetector performs browser extension instrumentation, and call graph and variable use analysis.

3.2.1 Instrumentation and Call Graph Analysis

While call graph construction has been commonly used in whole program analysis of C and Java code [15, 38], accurately constructing call graphs for JavaScript code is very challenging due to its extremely dynamic (1) code loading and generation [27, 28, 29, 41], (2) uses of functions, fields, and prototypes [6, 21, 26], and (3) interactions with other components of the browsers and webpages [2, 3, 18, 24].

To accurately construct call graphs that are the foundation of the overall analysis, we take an instrumentation approach to dynamically extract call relations among different functions within a browser extension. As shown in Figure 2, this *instrumentation* component takes a browser extension as input, automatically inserts program tracing statements to the extension, and outputs the instrumented extension.

Specifically, it (1) formats the source code of the extension so that each line contains one JavaScript statement, (2) adds unique prototype names to the functions (including methods) that do not have explicit ones so that all the functions can be uniquely identified, (3) inserts print statements before each function/method call so that the detailed callsite information such as the prototype name of the caller, the call statement, and the callsite position can be recorded, and (4) inserts a print statement at the entry point of each function definition so that the detailed information about the callee can be recorded. Because these transformations are simple and minimal, they do not interfere with the original program functionality and semantics. In the cases that some extensions use the dynamic features of JavaScript such as the eval() function to obfuscate their original source code, this instrumentation component uses the Closure Compiler [7] and the ScriptEngine class in Java to evaluate the eval() statements and de-obfuscate the source code before performing the aforementioned transformations. The de-obfuscated extension source code does not further contain any eval() as observed in our experiments

(Section 4), indicating that JavaScript in legitimate browser extensions rarely uses multi-level obfuscation.

An analyst can install and run such an instrumented extension to generate the execution traces for each particular use scenario. Because the execution traces only contain the call relations and do not contain any information from users, they can also be shared (e.g., in a repository along with the extensions) among the analysts to further cover more execution paths of the extension. For example, extension developers can run LvDetector and contribute execution traces based on their test cases, browser vendors can run LvDetector and contribute execution traces based on their inspection tasks, and advanced users can run LvDetector and contribute execution traces based on their trial runs. All these traces can be leveraged to automatically perform or replicate the actual vulnerability analysis.

The *call graph analysis* component analyzes the dynamically generated execution traces to build a call graph that precisely reflects the actual call relations in the real use scenario. The output call graph is a directed graph. Its nodes and edges are all the functions and call relations traversed in a scenario-driven execution, respectively. Such a call graph can accurately capture the complex and dynamic function/method calls that often occur in JavaScript extensions due to reflection, function objects, event handlers, asynchronous calls, DOM interactions, and so on.

3.2.2 Variable Use Analysis

The purpose of this component is to construct a graph that precisely defines the immediate value flow relations among variables in each function, based on an SSA IR [10] constructed from the source code of the browser extension.

For each function, its ***variable use graph*** is a directed graph with nodes representing all the variables defined/used in the function, and edges representing the operations used to propagate values among variables. The direction of an edge represents the value flow direction. Since the input program is converted to its SSA IR, each variable is statically and precisely defined once and thus is associated with a single value. Therefore, the static definition and uses of every variable in the program can be precisely correlated.

Our *variable use analysis* directly employs the output from an existing SSA builder [8]. The IR output of the SSA builder contains mappings between SSA variables and the original JavaScript variables, and mappings between SSA instructions and the original JavaScript statements. This mapping information will be used in the *vulnerability analysis* phase to generate intuitive vulnerability reports.

The main step of the *variable use analysis* is to extract the operands and operators from the instructions in SSA IR. Each operand represents a unique variable in SSA IR, and each operator represents an operation that may propagate values among variables. The operations include object field reference, getters/setters, string operation, array access, binary/unary operation, global variable reference, assignment operation, Φ-function [10], and function call. The variable use graph is then constructed in a straightforward fashion to precisely record such immediate explicit and implicit (via Φ-functions) value flow relations. Meanwhile, a list of global variable references will also be maintained. This list will be used in the *transitive variable relation analysis* phase to compute information flows across functions. Based source/sink variable and crypto-function identification rules in the rule file (Section 3.1), this component also automatically identifies all the source/sink variables and cryptographic functions, and feeds them to the next two analysis phases.

Figure 3(a) illustrates the three variable use graphs for the code excerpt in Figure 1. Here the edge $v_{10} \xrightarrow{+} v_{13}$ in the variable use graph for the encode() function represents a value flow from v_{10} to v_{13} through a string concatenation operation, and the edge $v_3 \xrightarrow{encode()} v_7$ in the variable use graph for the save() function represents a value flow from v_3 to v_7 through the *encode*() function call.

3.3 Transitive Variable Relation Analysis

This phase summarizes the transitive relations between each pair of source and sink variables at both the function-level and the program-level.

3.3.1 Function-level Relation Analysis

This component iteratively computes a function-level relation summary for each function based on its variable use graph, the call graph, and the labeled cryptographic functions. Such a summary contains the transitive relations between each pair of variables in that function.

We formulate the dynamically generated ***call graph*** as G in Formula 1. We categorize the original operators in a variable use graph into a set of abstract operators, $Operator$, defined in Formula 2. For example, the string concatenation and substring operators are categorized as "STRING_OP", the arithmetic operators are categorized as "BINARY_OP", the calls to the labeled encryption functions are categorized as "ENCRYPT", the calls to the labeled decryption functions are categorized as "DECRYPT", the calls to the JavaScript global functions (e.g., encodeURI()) are categorized as "JS_GLOBAL", and the calls to all other JavaScript functions are initially categorized as "UNKNOWN". The Φ-function used in SSA IR [10] is categorized as "Φ". We define the ***updated variable use graph*** for function f as F^f in Formula 3, in which each original operator in a variable use graph is replaced with its corresponding abstract operator to simplify the graph representation. In Formula 4, E represents the updated variable use graphs of all the functions in the call graph G.

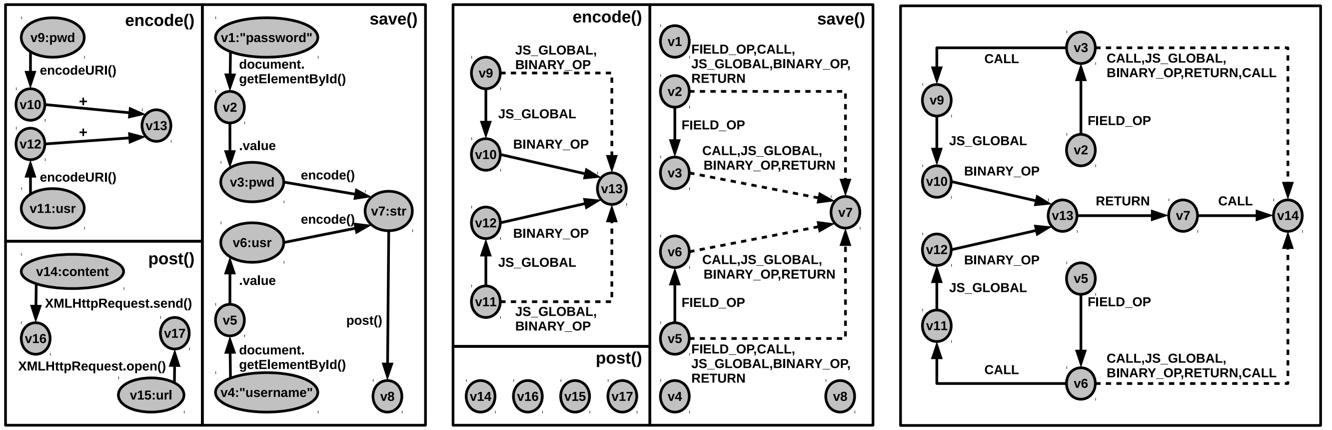
$$G = \{f_i \xrightarrow{s} f_j \mid s\ is\ a\ callsite\ from\ function\ f_i\ to\ f_j\} \quad (1)$$

$$\begin{aligned} Operator = \{&ENCRYPT,\ DECRYPT,\ JS_GLOBAL,\\ &PROTOTYPE,\ CONSTRUCT,\ FIELD_OP,\\ &ATTRIBUTE_OP,\ ARRAY_OP,\ STRING_OP\\ &UNARY_OP,\ BINARY_OP,\ \Phi,\ UNKNOWN\} \end{aligned} \quad (2)$$

$$\begin{aligned} F^f = \{&x^f \xrightarrow{op} y^f \mid x,\ y\ are\ variables\ referenced\ in\ f,\\ &f \in G,\ relation\ y = op(x)\ is\ in\ f,\ op \in Operator\} \end{aligned} \quad (3)$$

$$E = \bigcup_{f \in G} F^f \quad (4)$$

The value of a variable in a caller function may be passed to a variable in a callee function; we use an abstract operator $CALL$ to represent this type of value passing operation, and define $C_{forward}$ in Formula 5 as the set of all such call value flows extracted from the call graph G. The value of a variable in a caller function may be also updated with the return value from a callee function; we use an abstract operator $RETURN$ to represent this type of value return operation, and define $C_{backward}$ in Formula 6 as the set of all such



(a) Variable use graphs (b) Function-level relation analysis result (c) Program-level relation analysis result

Figure 3: The analysis results for the code excerpt in Figure 1. The dashed lines in (b) and (c) represent the computed transitive relations; to simplify the figure, we only kept the operators and omitted the variables in the labels of those dashed lines, and we only drew the two newly computed transitive relations in (c).

return value flows extracted from the call graph G. In Formula 7, E^+ defines the ***extended variable use graphs***, and it is the union of E, $C_{forward}$, and $C_{backward}$. In Formula 8, $Operator^+$ defines the extended set of abstract operators, and it is the union of $Operator$ and $\{CALL, RETURN\}$.

We use Formula 9 to define the transitive relation summary Q^f for function f, which is the set of transitive relations between each pair of variables x and y in the same function f. Each ***transitive relation*** is represented by a sequence of abstract operators and variables through which a value is passed from x to y. In Formula 10, Q defines the ***function-level relation summaries*** for all the functions in the call graph G.

$$C_{forward} = \{arg^{f_i} \xrightarrow{CALL} para^{f_j} \mid f_i \xrightarrow{s} f_j \in G, \\ arg^{f_i} \textit{ is the argument of the callsite } s \textit{ in } f_i, \\ para^{f_j} \textit{ is the corresponding parameter of } f_j\} \quad (5)$$

$$C_{backward} = \{ret^{f_j} \xrightarrow{RETURN} rec^{f_i} \mid f_i \xrightarrow{s} f_j \in G, \\ ret^{f_j} \textit{ is the variable returned in } f_j, \ rec^{f_i} \textit{ accepts} \\ \textit{the returned value from callsite } s \textit{ in } f_i\} \quad (6)$$

$$E^+ = E \cup C_{forward} \cup C_{backward} \quad (7)$$

$$Operator^+ = Operator \cup \{CALL, \ RETURN\} \quad (8)$$

$$Q^f(x,y) = \{(x^f, op_1, v_1^{f_1}, op_2, v_2^{f_2}, \cdots, v_{k-1}^{f_{k-1}}, op_k, y^f) \mid \\ f, f_1, f_2, \ldots, f_{k-1} \in G, \ op_1, op_2, \ldots op_k \in Operator^+, \\ x^f \xrightarrow{op_1} v_1^{f_1}, \ v_1^{f_1} \xrightarrow{op_2} v_2^{f_2}, \ \ldots, \ v_{k-1}^{f_{k-1}} \xrightarrow{op_k} y^f \in E^+\} \quad (9)$$

$$Function\ level\ relation\ summaries\ Q = \bigcup_{f \in G} Q^f \quad (10)$$

Figure 4 illustrates the function-level relation analysis algorithm for computing Q. The algorithm consists of two procedures. The Compute-ExtendedVariableUseGraphs procedure constructs the variable use graphs E (Formula 4) from line 2 to line 4, constructs $C_{forward}$ (Formula 5) from line 5 to line 8, constructs $C_{backward}$ (Formula 6) from line 9 to line 12, and finally returns the extended variable use graphs E^+ (Formula 7) at line 14.

The Compute-FunctionLevelRelations procedure initializes each transitive relation summary Q^f for function f with its updated variable use graph F^f at line 2 in the first for loop. In the following do-while loop, for each function f in the post-order traversal of G, this procedure updates Q^f with the newly computed transitive relations for each pair of variables in that function f from line 6 to line 7. The post-order traversal is used at line 5 so that callee functions are analyzed prior to their caller functions whenever possible. This update is an iterative process, and the do-while loop terminates when no more update occurs to any Q^f. The union of all the Q^fs is returned at line 10.

The *compute_transitive_summary* sub-procedure is capable of summarizing paths and cycles to compute transitive relations on a graph, based on the transitive operations defined for a given problem. In this sub-procedure, cycles are summarized using their equivalent directed acyclic graphs (DAGs) [40], and "UNKNOWN" operators are replaced with their corresponding transitive relations in the callee functions; therefore, the do-while loop from line 4 to line 8 ***must terminate***. Many existing transitive closure computation algorithms such as [40, 39] could be adapted to implement this sub-procedure. We implement this sub-procedure in our framework by adapting the algorithm in [40], which is an efficient algorithm with a time complexity linear to the number of nodes and edges in the input graph.

Figure 3(b) illustrates the *function-level relation analysis* result for the code excerpt in Figure 1. For example, the computed transitive relation from v_3 to v_7 is labeled with "CALL,JS_GLOBAL,BINARY_OP,RETURN"; we only kept these operators and omitted the variables in the label to simplify the figure.

3.3.2 Program-level Relation Analysis

The purpose of the *program-level relation analysis* is to compute program-level relation summary based on the call graph and the function-level relation summaries. Specifically, it aims to further summarize transitive relations between each pair of the specified source and sink variables, irrespective of whether the pair of variables are defined in the same function or in different functions.

It is important to note that partial cross-function relations (i.e., $C_{forward}$ and $C_{backward}$) have been included in the function-level relation analysis algorithm shown in Figure 4. Computing function-level relation summaries based on small-size and localized extended variable use graphs be-

```
Compute-ExtendedVariableUseGraphs (P, G)
      // P: program; G: call graph.
1     E = C_forward = C_backward = ∅;
2     for each function f ∈ G do
3         F^f = get_updated_var_use_graph(P, f);
4         E = E ∪ F^f;
5     for each edge f_i --s--> f_j ∈ G do
6         arg^{f_i} = argument_of_callsite(s);
7         para^{f_j} = parameter_of_function(f_j);
8         C_forward = C_forward ∪ (arg^{f_i} --CALL--> para^{f_j});
9     for each edge f_i --s--> f_j ∈ G do
10        ret^{f_j} = return_var_of_function(f_j);
11        rec^{f_i} = accept_return_value_var(s);
12        C_backward = C_backward ∪ (ret^{f_j} --RETURN--> rec^{f_i});
13    E^+ = E ∪ C_forward ∪ C_backward;
14    return E^+;

Compute-FunctionLevelRelations (P, G, E^+)
      // P: program; G: call graph;
      // E^+: extended variable use graphs.
1     for each function f ∈ G do
2         Q^f = F^f = get_updated_var_use_graph(P, f);
3         vars^f = get_nodes_in(F^f);
4     do
5         for each function f in the post-order traversal of G do
6             for each pair of variables src, dst ∈ vars^f do
7                 Q^f(src, dst) =
                      compute_transitive_summary(E^+, src, dst);
8     while at least one Q^f is updated
9     Q = ⋃_{f∈G} Q^f;
10    return Q;
```

Figure 4: Function-level relation analysis algorithm

fore computing program-level relation summary is critical for the LvDetector framework to efficiently analyze large and complex extensions; otherwise, directly analyzing transitive relations on a program-level graph consisting of many extended variable use graphs with complex cycles and paths will be very inefficient. This is the key reason for us to explicitly divide the transitive variable relation analysis into two steps at the function-level and program-level.

The value of a variable in a function may be passed to another variable in another function through global variables or JavaScript events. We use an abstract operator $GLOBAL$ to represent the type of value passing through global variables, and define C_{global} in Formula 11 as the set of all such global value flows extracted from the whole program P; we use an abstract operator $MESSAGE$ to represent the type of value passing through JavaScript events, and define $C_{message}$ in Formula 12 as the set of all such message value flows extracted from the whole program P. In Formula 13, E' defines the further-extended variable use graphs, and it is the union of E^+, C_{global}, and $C_{message}$; in Formula 14, $Operator'$ defines the further-extended set of abstract operators, and it is the union of $Operator^+$ and $\{GLOBAL, MESSAGE\}$.

Formula 15 defines the transitive relation summary, T^{f_i,f_j}, which is the set of transitive relations from any variable x in function f_i to any variable y in function f_j. Formula 16 defines the ***program-level relation summary***, T, which is the output of the program-level relation analysis component.

$$C_{global} = \{v^{f_i} \xrightarrow{GLOBAL} v^{f_j} \mid \textit{global variable } v \textit{ is defined in } f_i \textit{ and used in } f_j\} \quad (11)$$

$$C_{message} = \{arg^{f_i} \xrightarrow{MESSAGE} para^{f_j} \mid \textit{an event is dispatched in } f_i\textit{, and processed in } f_j\textit{, } arg^{f_i} \textit{ is the argument to this event, } para^{f_j} \textit{ is the corresponding parameter of } f_j\} \quad (12)$$

$$E' = E^+ \cup C_{global} \cup C_{message} \quad (13)$$

$$Operator' = Operator^+ \cup \{GLOBAL, MESSAGE\} \quad (14)$$

$$T^{f_i,f_j}(x,y) = \{(x^{f_i}, op_1, v_1^{f_1}, \cdots, v_{k-1}^{f_{k-1}}, op_k, y^{f_j}) \mid f_i, f_1, \ldots, f_{k-1}, f_j \in G,\ op_1, \ldots op_k \in Operator',\ x^{f_i} \xrightarrow{op_1} v_1^{f_1}, \ldots, v_{k-1}^{f_{k-1}} \xrightarrow{op_k} y^{f_j} \in E'\} \quad (15)$$

$$\textit{Program level relation summary } T = \bigcup_{f_i, f_j \in G} T^{f_i,f_j} \quad (16)$$

```
Compute-ProgramLevelRelations (P, Q, E^+, sVars, dVars)
      // P: program; Q: function-level relation summaries;
      // E^+: extended variable use graph;
      // sVars: a set of source variables;
      // dVars: a set of destination (sink) variables.
1     C_global = C_message = ∅;   T = Q;
2     for each global variable v in P do
3         defs = get_definitions(v); uses = get_uses(v);
4         for each pair of v^{f_i} ∈ defs and v^{f_j} ∈ uses do
5             C_global = C_global ∪ (v^{f_i} --GLOBAL--> v^{f_j});
6     for each event evt dispatched in f_i and processed in f_j do
7         arg^{f_i} = argument_of_event(evt);
8         para^{f_j} = parameter_of_function(f_j);
9         C_message = C_message ∪ (arg^{f_i} --MESSAGE--> para^{f_j});
10    E' = E^+ ∪ C_global ∪ C_message;
11    for each pair of src ∈ sVars and dst ∈ dVars do
12        T^{f_i,f_j}(src, dst) =          // src is in f_i, dst is in f_j
              compute_transitive_summary(E', src, dst);
13    return T;
```

Figure 5: Program-level relation analysis algorithm

Figure 5 illustrates the overall program-level relation analysis algorithm for computing T. It constructs C_{global} (Formula 11) from line 2 to line 5, constructs $C_{message}$ (Formula 12) from line 6 to line 9, builds the further-extended variable use graphs E' (Formula 13) at line 10, updates T with the newly computed transitive relations from line 11 to line 12 for each pair of variables constructed from the input sets $sVars$ and $dVars$, and finally returns T. The *compute_transitive_summary* sub-procedure at line 12 is the same one that is used in the function-level relation analysis algorithm (Figure 4). It is worth mentioning that in the program-level relation analysis, the number of edges ***will not increase exponentially*** because paths and cycles were summarized in the *compute_transitive_summary* sub-procedure, and the transitive relations computed in the function-level analysis will not be computed again in the program-level analysis.

Figure 3(c) illustrates the *program-level relation analysis* result for the code excerpt in Figure 1. The source variables are v_3 and v_6, and the sink variable is v_{14}. Two new transitive relations are computed from v_3 to v_{14} and from v_6 to v_{14}, respectively; both of them are labeled with "CALL,JS_GLOBAL,BINARY_OP,RETURN,CALL".

3.4 Vulnerability Analysis

The purpose of this phase is to analyze vulnerabilities based on the program-level relation summary and the source-destination (sink) variable pairs as shown in Figure 6. For all the relations from a source variable to a destination (sink) variable, currently LvDetector reports vulnerabilities based on two rules. One is that the ENCRYPT abstract operator does not appear in a relation (line 3); the other is that both the ENCRYPT and DECRYPT abstract operators appear in a relation, but no ENCRYPT abstract operator appears after the last DECRYPT abstract operator (line 5). Otherwise, LvDetector simply records a relation as a non-vulnerable information flow (line 8). Application developers may misuse cryptographic primitives as demonstrated by Egele et al. [13]. The current version of LvDetector does not further examine cryptographic misuses such as using constant keys or non-random initialization vectors in browser extensions, thus its vulnerability detection is more like a lower-bound analysis.

```
Analyze-Vulnerability (T, sVars, dVars)
    // T: program-level relation summary;
    // sVars: a set of source variables;
    // dVars: a set of destination (sink) variables.
1   for each pair of src ∈ sVars and dst ∈ dVars do
2       for each relation r ∈ T(src, dst) do
3           if the ENCRYPT operator does not appear in r then
4               report_vulnerability(r);
5           else if the DECRYPT operator appears in r but no
            ENCRYPT appears after the last DECRYPT then
6               report_vulnerability(r);
7           else
8               record_non_vulnerable_flow(r);
```

Figure 6: Vulnerability analysis algorithm

The report_vulnerability sub-procedure automatically classifies the source variables into two categories. All the source variables that accept sensitive information (e.g., the *password* type inputs, cookies, and bookmarks) either from user inputs or webpages are in the *sensitive* category, and the rest are in the *other* category. This sub-procedure also groups the sink variables into the *network* category and the *local disk* category, with their values sent across the network or saved to the local disk, respectively. It further classifies the reported vulnerabilities as *high-severity*, *medium-severity*, and *unranked* ones as shown in Table 1. Those vulnerabilities that leak information from the sensitive source variables to the network sink variables are classified as high-severity. Those vulnerabilities that leak information from the sensitive source variables to the local disk sink variables are classified as medium-severity because unprotected sensitive information on a user's local disk can also lead to security breaches due to, for example, bots [33]. The rest are classified as unranked because their source variables are not automatically classified as sensitive; an analyst can further classify these unranked ones based on whether the source variables can be considered as sensitive.

Each vulnerability report contains a list of high-severity, medium-severity, and unranked vulnerability records for each scenario-driven execution. Each record includes the complete information flow, and highlights the original variables and operations to provide more intuitive information. For example, for the code excerpt in Figure 1, the information flow from the variable "pwd" in the save() function to the

Table 1: Vulnerability classification

Source Vars \ Sink Vars	Network	Local disk
Sensitive	High-severity	Medium-severity
Other	Unranked	Unranked

variable "content" in the post() function is identified as a high-severity vulnerability, and the corresponding record is: $v_3(\boldsymbol{pwd}) \xrightarrow{CALL(\textbf{encode()})} v_9(\boldsymbol{pwd}) \xrightarrow{JS_GLOBAL(\textbf{encodeURI()})} v_{10}() \xrightarrow{BINARY_OP(+)} v_{13}() \xrightarrow{RETURN(\textbf{encode()})} v_7(\boldsymbol{str}) \xrightarrow{CALL(\textbf{post()})} v_{14}(\boldsymbol{content})$. Note that the contents in the parentheses such as "pwd", "+", and "encode()" are the original variables, operations, and function calls in the source code. In addition, the locations (i.e., file names, function names, and line numbers) of the original variables, operations, and function calls are also provided in each record. This intuitive information can help analysts easily locate the reported vulnerabilities in the extensions.

4. EVALUATION

We implemented LvDetector in Java. We also integrated two popular compilers into the LvDetector framework. In the instrumentation component, we used Closure Compiler [7] to identify all the functions and callsites. We chose WALA Compiler [8] as the SSA builder to generate SSA IRs. We evaluated LvDetector on 28 most popular or top rated extensions that belong to six categories as shown in Table 2; 17 of them were selected from the Firefox extension store [14], and 11 of them were selected from the Google Chrome extension store [5]. The main criteria for choosing these extensions are: they must use cryptographic functions; they must have sensitive source variables and network sink variables so that high-severity vulnerabilities may exist (Section 3.4). In the following subsections, we detail one case study, the overall analysis results for 28 extensions, the responsible disclosure and feedback, and the performance results; we also further discuss the false positives and false negatives.

4.1 Case Study of RoboForm

RoboForm (Lite) is a Firefox extension that can help users remember and auto-fill their website passwords [30]. It provides a master password mechanism to further protect users' website passwords. We used LvDetector to analyze RoboForm on six use scenarios.

Scenario 1: A user provides the master password in a 'password' type input field to RoboForm to protect the saved website passwords. The master password is automatically classified as sensitive; it should only be known by the user and should not be sent out even to the remote server of RoboForm. LvDetector identified one high-severity vulnerable information flow, in which the master password is leaked out through one sink statement, the send() method call of an XMLHttpRequest object, without the protection of any cryptographic function. We verified that this information flow is indeed vulnerable.

Scenario 2: A user allows RoboForm to save a website password to its remote server. The website password is automatically classified as sensitive; it should only be known by the user and the corresponding website. LvDetector identified two high-severity vulnerable information flows, in which the website password is leaked out through the same sink statement as in scenario 1 without the protection of any

cryptographic function. They are two flows because they take different code branches. We verified that these two information flows are indeed vulnerable.

Scenario 3: A user allows RoboForm to save a website password to the local disk without using a master password. LvDetector identified one medium-severity vulnerable information flow, in which the website password is leaked out through one sink statement, the write() method call of a FileOutputStream object, without the protection of any cryptographic function. We verified that this information flow is indeed vulnerable.

Scenario 4: A user allows RoboForm to save a website password to the local disk with the protection of a master password. LvDetector identified two information flows and simply recorded them as non-vulnerable: one saves the website password to the local disk after performing an AES encryption, the other saves the master password to the local disk after performing a DES encryption. We verified that these two information flows are indeed non-vulnerable.

Scenarios 5 and 6: A user creates (scenario 5) and types (scenario 6) a RoboForm login account in a dialog box. The RoboForm login password is automatically classified as sensitive; it should only be known by the user and RoboForm. LvDetector identified one high-severity vulnerable information flow in each of the two scenarios. The RoboForm login password is leaked out through the same sink statement as in scenario 1 without the protection of any cryptographic function. However, these two information flows should not be identified as vulnerable because the RoboForm login password is sent only to the remote server of RoboForm.

4.2 Overall Results

Table 2 summarizes the overall analysis results on the 28 extensions. The second column lists the number of the use scenarios chosen in each extension. The third column lists the number of analyzed statements over the total number of statements in each extension. The fourth column lists the number of different cryptographic functions identified in each extension. The fifth column lists the number of the source variables in each extension for all the chosen scenarios. The sixth column lists the number of the sink variables in each extension for all the chosen scenarios. The seventh column lists the number of ***true positives (TP)*** that are vulnerable information flows correctly identified by LvDetector; correspondingly, the eighth column lists the number of ***false positives (FP)*** that include nonexistent flows and non-vulnerable existent flows. The ninth column lists the number of ***true negatives (TN)*** that are non-vulnerable information flows correctly identified by LvDetector; correspondingly, the last column lists the number of ***false negatives (FN)*** that are vulnerable information flows incorrectly identified by LvDetector as non-vulnerable. These TP/FP/TN/FN numbers come from our examination of the information flows reported/recorded by LvDetector (Figure 6).

For example, we chose six scenarios in the RoboForm case study (Section 4.1). LvDetector analyzed 6880 out of the total 26120 lines of code. LvDetector automatically identified six different cryptographic functions, and automatically identified seven source variables and 19 sink variables, LvDetector detected six vulnerable information flows with four true positives and two false positives, and recorded two non-vulnerable information flows with two true negatives and zero false negative.

The following five formulas present the precision, recall, F-measure, accuracy, and false positive rate calculations for the results in Table 2.

$$Precision(Pre) = \frac{TP(18)}{TP(18) + FP(6)} = 75\% \tag{17}$$

$$Recall(Rec) = \frac{TP(18)}{TP(18) + FN(0)} = 100\% \tag{18}$$

$$F-measure = \frac{2 \times Rec(100\%) \times Pre(75\%)}{Rec(100\%) + Pre(75\%)} = 86\% \tag{19}$$

$$Accuracy = \frac{TP(18) + TN(23)}{TP(18) + TN(23) + FP(6) + FN(0)} = 87\% \tag{20}$$

$$FalsePositiveRate = \frac{FP(6)}{FP(6) + TN(23)} = 21\% \tag{21}$$

A good analysis framework should achieve high precision and high recall. However, a tradeoff often exists between high precision and high recall because achieving one may compromise the other and vice versa. To combine precision and recall, the harmonic mean of them, F-measure, is often used. Accuracy is the overall success rate of the analysis.

From these calculations, we can conclude that LvDetector is an effective framework. It achieves a high precision rate (75%), indicating that the majority of the identified vulnerable flows are indeed vulnerable. It achieves a high recall rate (100%), indicating that LvDetector can identify the majority of the actually vulnerable flows for the executed scenarios. It also achieves a high F-measure rate (86%) and a high accuracy rate(87%). The false positive rate is 21%; however, the detection results of LvDetector will be used by analysts to more easily identify information flow vulnerabilities. This usage is different from that of other systems such as intrusion detection or online malware detection systems, in which the detection results will be used to make immediate decisions such as dropping network packets or removing malicious programs. Therefore, a 21% false positive rate will not cause too much inconvenience to the analysts.

Overall, LvDetector identified 18 true information leakage vulnerabilities in 13 extensions. These vulnerabilities are previously unknown, and they exist in 46% of the analyzed extensions. Nine of them are high-severity vulnerabilities, and seven of them are medium-severity vulnerabilities. The remaining two are unranked because their source variables that accept users' notes are not automatically classified as sensitive; they can be further classified as one high-severity and one medium-severity vulnerabilities, respectively, since users' notes and tasks may contain sensitive information. We examined that three main reasons account for those 18 vulnerabilities: developers did not realize the importance of protecting sensitive data before sending or saving them, protection was not applied to all the code branches for sensitive information flows, code had bugs such as sending or saving plaintext rather than ciphertext. These information leakage problems deserve serious attention from extension developers, browser vendors, researchers, and users.

4.3 Responsible Disclosure and Feedback

Among the 13 extensions that have vulnerabilities, 12 of them contain contact information on their websites or extension store webpages. We emailed those 12 developers asking if they would like to know the details about the vulnerabilities in their extensions, and received eight replies.

Table 2: Analysis results on 28 Firefox * and Google Chrome + extensions

Extensions	Num of Use Scenarios	Num of Statements (Analyzed / Total)	Num of Different Crypto Functions	Num of Variables		Num of Positives		Num of Negatives	
				Source	Sink	True (TP)	False (FP)	True (TN)	False (FN)
Feeds, News & Blogging									
1. Gmail Manager NG *	1	455 / 1482	4	1	3	0	0	1	0
2. Email Notifier *	1	212 / 1680	1	1	11	1	0	0	0
Shopping									
3. Shoptimate *	1	287 / 10612	1	1	2	2	0	0	0
4. EFT Pass +	1	735 / 4466	4	1	5	0	0	0	0
Privacy & Security									
5. Autofill Forms *	1	3490 / 3524	1	1	1	1	4	1	0
6. Cookies Manager *	1	419 / 3753	2	1	1	1	0	3	0
7. Secure Bookmarks +	1	2028 / 2569	2	1	5	0	0	6	0
8. Lazarus *	2	3525 / 7610	2	2	65	0	0	3	0
9. RoboForm *	6	6880 / 26120	6	7	19	4	2	2	0
10. QuickPasswords *	1	2082 / 2170	2	1	8	0	0	0	0
11. Link Password *	1	984 / 984	3	2	3	0	0	3	0
12. uPassword *	1	1467 / 3803	1	2	2	0	0	0	0
13. MD5 Reborned Hasher *	1	504 / 504	2	1	1	0	0	0	0
14. Encrypted Communication *	1	404 / 404	2	2	1	0	0	2	0
15. EverSync *	1	3868 / 6273	2	1	7	1	0	0	0
Productivity									
16. Add Tasks to Do It +	1	351 / 467	1	1	6	1	0	0	0
17. Tab Wrangler +	1	228 / 3659	1	1	1	1	0	0	0
18. Any.do +	3	442 / 12980	1	3	14	2	0	0	0
19. 123 Password +	1	128 / 418	2	1	1	0	0	0	0
20. ChromePW +	1	307 / 787	1	1	2	0	0	1	0
Social & Communication									
21. X-notifier +	1	846 / 5220	4	1	4	1	0	0	0
22. Simple Mail *	1	5013 / 9832	2	1	13	0	0	1	0
23. Inbox Ace +	1	15682 / 20251	1	1	19	0	0	0	0
24. Google Plus Follower Stats *	1	600 / 5532	1	1	1	0	0	0	0
25. FoxToPhone *	1	1342 / 1771	1	1	15	1	0	0	0
26. FB Chat History Manager *	1	102 / 560	3	1	1	0	0	0	0
Accessibility									
27. Smart Bookmarks +	1	1038 / 2818	1	1	1	1	0	0	0
28. AnnoPad +	1	566 / 10533	1	1	6	1	0	0	0
Total	36	53965 / 150783	18	40	218	18	6	23	0

We further provided the detailed vulnerabilities to the eight developers respectively. Two of them patched their extensions in the online stores; one of them removed his extension from the Google extension store; four of them did not further respond to us; one of them disagreed with our analysis result, and mentioned that tons of extensions persist much more sensitive data all over the place and his extension does not encrypt data because the browser's storage APIs do not provide encryption options.

4.4 Performance Results

We measured the running time of LvDetector in analyzing the vulnerabilities in each extension on a desktop computer with 2.83GHz CPU, 2.96GB memory, 32-bit Windows 7 operating system, and Java Runtime Environment 1.7. For the vulnerability analysis of the 36 use scenarios (Table 2), the maximum running time is about 48 minutes (corresponding to the scenario 1 of the RoboForm case study described in Section 4.1), the minimum running time is less than one minute, the median running time is six minutes, and the average running time is 12 minutes with a standard deviation of 13. Because LvDetector is an offline analysis framework, such a running time performance is quite acceptable. Note that the running time is not linear to the Lines of Code, and it is often related to the code complexity.

4.5 Discussion

False positives may come from a few sources. First, in the *variable use analysis*, the conditions in the control flow of the SSA IR are not currently considered; therefore, extra (i.e., nonexistent) information flows could be later included in the *function-level relation analysis*. Second, in the *variable use analysis*, the individual elements in a collection type of object such as array or linkedlist are not further differentiated from each other - the analysis granularity is only at the object level; therefore, extra information flows could be later included in the *function-level relation analysis*. Third, in the *program-level relation analysis*, all possible edges are created between global variable definitions and their uses; therefore, extra information flows could be included. Fourth, if the value of a sensitive variable is leaked to an intended remote server for further processing, this type of leakage should not be identified as vulnerable. Among the 6 false positives reported in our evaluation results (Section 4.2), four of them

come from the first source, and the remaining two come from the fourth source as explained in the scenarios 5 and 6 of the RoboForm case study (Section 4.1). Although in our evaluation we did not observe any false positive coming from the second and the third sources, analysts should still pay attention to these sources when they use LvDetector.

False negatives may occur due to reasons such as the misuses of cryptographic primitives [13], which are not further examined by the current version of LvDetector (Section 3.4). In addition, because LvDetector builds call graphs from the scenario-driven execution traces, vulnerabilities will not be identified for the scenarios that are not executed by analysts.

In the future, we plan to consider the conditions in the control flow of the SSA IR, differentiate the elements in an Array from each other, and refine the mappings between global variable definitions and their uses; with these enhancements, we expect that false positives can be reduced. We also plan to investigate potential cryptographic misuses [13] in browser extensions and other path exploration techniques such as [23, 32] to see if LvDetector can identify more vulnerabilities.

5. RELATED WORK

Existing research on analyzing the security of JavaScript-based extensions mostly focused on identifying privilege escalation related vulnerabilities that could lead to web-based attacks or malware installation. Researchers applied static information flow analysis techniques [1] and dynamic taint analysis techniques [12] to identify privilege escalation related vulnerabilities in buggy browser extensions. Guha et al. proposed a new model for secure development, verification, and deployment of browser extensions to limit potential over-privilege problems [18]. Barth et al. designed a new extension system for Google Chrome that uses least privilege, privilege separation, and strong isolation mechanisms [2]. Carlini et al. manually reviewed and evaluated the effectiveness of those three mechanisms in a set of Google Chrome extensions [3]. Liu et al. revealed that malicious attacks can still violate the least privilege and privilege separation mechanisms of the Google Chrome extension system, and proposed some countermeasures [24].

Only a handful of existing solutions [4, 11, 22] aimed to address the same problem targeted by our LvDetector, but they took either pure dynamic approaches or pure static approaches. In [4], Chang and Chen proposed a framework, *iObfus*, to dynamically protect against the potential sensitive information leakage through browser extensions. iObfus marks sensitive web elements, obfuscates the sensitive information before performing any I/O operation, and de-obfuscates the information only for trusted domains. In [11], Dhawan and Ganapathy proposed a framework, *Sabre*, to dynamically track information flows in JavaScript-based Firefox extensions. Sabre associates security labels with JavaScript objects, tracks the propagation of those labels at runtime in the SpiderMonkey JavaScript engine of Firefox, and raises an alert if an object with a sensitive label is written to a low-sensitivity sink point. These frameworks are not publicly available for comparison. However, generally speaking, only using online dynamic techniques without performing a static analysis in advance suffers from ***three main drawbacks***: (1) asking users to respond to runtime alerts may not be wise, while using default response options may become too restrictive [1]; (2) it is not possible to detect all information flows dynamically [31, 36]; (3) performance and memory overhead can often be incurred to the system [1]. In addition, dynamic approaches are often browser-specific and require high instrumentation effort [11]. In [22], Kashyap and Hardekopf proposed an abstract interpretation approach to validate the pre-defined security signatures for browser extensions; however, pure static analysis of JavaScript-based extensions can often incur high false positives as we discussed in Section 3.1. Our LvDetector combines both static and dynamic program analysis techniques, and aims to automatically identify information leakage vulnerabilities in browser extensions before they are released to users.

Static and dynamic program analysis techniques have also been used to address other JavaScript security problems in web applications. On the one hand, static program analysis techniques have been used to detect JavaScript malware [6, 9], detect web application vulnerabilities such as injection and cross-site scripting [17], and examine a restricted version of JavaScript that enables the API confinement verification [34]. Static techniques can provide a comprehensive code coverage, but may over-estimate the actual execution paths and incur false positives. On the other hand, dynamic program analysis techniques have been used to enforce information flow security for a set of core features in JavaScript [19], detect privacy-violating information flows such as cookie stealing and history sniffing [20], and identify client-side code injection vulnerabilities [32]. Dynamic techniques can capture the precise program execution information, but may overlook certain potential execution paths and incur false negatives. Static and dynamic program analysis techniques have also been combined to prevent cross-site scripting attacks [35, 36], track information flow in JavaScript code injection attacks [21], and extract the dynamically generated code for analyzing script injection attacks [37]. Our LvDetector uses both static and dynamic program analysis techniques but focuses on addressing a different problem than those addressed by this body of work.

6. CONCLUSION

In this paper, we present a framework, LvDetector, that combines static and dynamic program analysis techniques for automatic detection of information leakage vulnerabilities in legitimate browser extensions. Extension developers can use LvDetector to locate and fix the vulnerabilities in their code; browser vendors can use LvDetector to decide whether the corresponding extensions can be hosted in their online stores; advanced users can also use LvDetector to determine if certain extensions are safe to use. LvDetector is not bound to specific web browsers or JavaScript engines; it follows a modular design principle, and can adopt other program analysis techniques. We implemented LvDetector in Java and evaluated it on 28 popular Firefox and Google Chrome extensions. The evaluation results and the feedback to our responsible disclosure demonstrate that LvDetector is useful and effective.

7. ACKNOWLEDGMENTS

The authors sincerely thank anonymous reviewers for their valuable suggestions and comments. The first two authors were supported in part by NSF grants CNS-1359542 and DGE-1438935. The third author was supported in part by NSF grant CCF-1261811.

8. REFERENCES

[1] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. Vex: Vetting browser extensions for security vulnerabilities. In *Proc. of USENIX Security Symposium*, pages 339–354, 2010.

[2] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *Proc. of NDSS*, 2010.

[3] N. Carlini, A. P. Felt, and D. Wagner. An evaluation of the google chrome extension security architecture. In *Proc. of USENIX Security Symposium*, 2012.

[4] W. Chang and S. Chen. Defeat information leakage from browser extensions via data obfuscation. In *Proc. of ICICS*, 2013.

[5] Google Chrome Extensions. `https://chrome.google.com/extensions/`.

[6] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *Proc. of ACM PLDI*, pages 50–62. ACM, 2009.

[7] Closure Compiler. `https://developers.google.com/closure/compiler/`.

[8] WALA Compiler. `http://wala.sourceforge.net/wiki/index.php`.

[9] C. Curtsinger, B. Livshits, B. G. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *Proc. of USENIX Security Symp.*, 2011.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4), 10 1991.

[11] M. Dhawan and V. Ganapathy. Analyzing information flow in javascript-based browser extensions. In *Proc. of ACSAC*, pages 382–391, 2009.

[12] V. Djeric and A. Goel. Securing script-based extensibility in web browsers. In *Proc. of USENIX Security Symposium*, 2010.

[13] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In *Proc. of CCS*, 2013.

[14] Firefox Extensions. `https://addons.mozilla.org/`.

[15] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Proc. of ACM OOPSLA*, pages 108–124, 1997.

[16] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *Proc. of USENIX Security Symposium*, pages 151–168, 2009.

[17] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable javascript. In *Proc. of ISSTA*, 2011.

[18] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *Proc. of IEEE S&P Symposium*, pages 115–130, 2011.

[19] D. Hedin and A. Sabelfeld. Information-flow security for a core of javascript. In *Proc. of IEEE CSF*, 2012.

[20] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in javascript web applications. In *Proc. of CCS*, 2010.

[21] S. Just, A. Cleary, B. Shirley, and C. Hammer. Information flow analysis for javascript. In *Proc. of ACM PLASTIC Workshop*, pages 9–18, 2011.

[22] V. Kashyap and B. Hardekopf. Security signature inference for javascript-based browser addons. In *Proc. of IEEE/ACM CGO Symposium*, pages 219–229, 2014.

[23] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Proc. of IEEE S&P Symposium*, pages 443–457, 2012.

[24] L. Liu, X. Zhang, G. Yan, and S. Chen. Chrome extensions: Threat analysis and countermeasures. In *Proc. of NDSS*, 2012.

[25] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, 2015.

[26] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of javascript applications in the presence of frameworks and libraries. In *Proc. of ESEC/FSE*, pages 499–509, 2013.

[27] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: Large-scale evaluation of remote javascript inclusions. In *Proc. of CCS*, pages 736–747, 2012.

[28] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do - a large-scale study of the use of eval in javascript applications. In *Proc. of ECOOP*, 2011.

[29] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *Proc. of ACM PLDI*, 2010.

[30] RoboForm. `http://www.roboform.com/`.

[31] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE JSAC*, 21(1), 2003.

[32] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *IEEE S&P Symp.*, 2010.

[33] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. A. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: analysis of a botnet takeover. In *Proc. of CCS*, 2009.

[34] A. Taly, U. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated Analysis of Security-Critical JavaScript APIs. In *Proc. of IEEE S&P Symp.*, 2011.

[35] O. Tripp, P. Ferrara, and M. Pistoia. Hybrid security analysis of web javascript code via dynamic partial evaluation. In *Proc. of ISSTA*, pages 49–59, 2014.

[36] 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. of NDSS*, 2007.

[37] S. Wei and B. G. Ryder. Practical blended taint analysis for javascript. In *Proc. of ISSTA*, 2013.

[38] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Proc. of ACM POPL*, 1980.

[39] M. Weiser. Program slicing. In *Proc. of ICSE*, 1981.

[40] Q. Yi, V. Adve, and K. Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In *Proc. of ACM PLDI*, pages 169–181, 2000.

[41] C. Yue and H. Wang. A measurement study of insecure javascript practices on the web. *ACM Transactions on the Web*, 7(2):7:1–7:39, 2013.