# Behavior Decomposition: Aspect-level Browser Extension Clustering and Its Security Implications

Bin Zhao and Peng Liu

The Pennsylvania State University-University Park, PA, USA
{biz5027,pliu}@ist.psu.edu

**Abstract.** Browser extensions are widely used by millions of users. However, large amount of extensions can be downloaded from webstores without sufficient trust or safety scrutiny, which keeps users from differentiating benign extensions from malicious ones. In this paper, we propose an aspect-level behavior clustering approach to enhancing the safety management of extensions. We decompose an extension's runtime behavior into several pieces, denoted as AEBs (Aspects of Extension Behavior). Similar AEBs of different extensions are grouped into an "AEB cluster" based on subgraph isomorphism. We then build profiles of AEB clusters for both extensions and categories (of extensions) to detect suspicious extensions. To the best of our knowledge, this is the first study to do aspect-level extension clustering based on runtime behaviors. We evaluate our approach with more than 1,000 extensions and demonstrate that it can effectively and efficiently detect suspicious extensions.

**Keywords:** Behavior Clustering, Graph Isomorphism, Browser Security

## 1 Introduction

Extensions are pervasively supported by commodity web browsers, such as Firefox, Chrome, and Internet Explorer. With thousands of extensions in webstores, Firefox add-ons are the most heavily used extensions. It is reported that 85% of Firefox 4 users have installed an extension, with "more than 2.5 billion downloads and 580 million extensions in use every day in Firefox 4 alone" [23].

However, as we will shortly discuss in Section 2, there are three major security issues associated with those extensions. First, to support the enhanced functionality, web browsers usually grant the "guest" extensions from third-party with full or similar privileges as granted to the "host" browsers themselves [8]. This entails that they can breach the sandboxing policy and the same origin policy. Second, extensions can hide themselves or even masquerade other legitimate ones to conduct malicious actions. Third, there lacks a sufficient security management for extensions among developers, browser webstores, and users.

**Protection Requirements.** To address these issues, a variety of techniques have been proposed in the literature; however, existing techniques are still limited in meeting the following real-world protection requirements: (R1) User data confidentiality and integrity [20]; (R2) Simplicity and practicality in deployment and use, which means the approach should not require one to modify the browser

code; (R3) Resilience to code obfuscation/polymorphism and runtime actions of JavaScripts; (R4) Acceptable overhead to the browser and OS.

**Limitations of Prior Approaches.** To see the limitations of existing defenses with respect to these four requirements, let us break down prior approaches into three classes which we will review shortly in Section 8: (C1) Sandboxing policy; (C2) Using static information flow analysis to identify potential security vulnerabilities in extensions [2, 29]; (C3) Using dynamic information flow to monitor the execution of extensions [8, 20, 21].

We briefly summarize their limitations as follows. (a) Classes C1 and C2 cannot meet R1, as they often have a high false negative rate. (b) Classes C1 and C3 cannot satisfy R2 because they often require browser code modification or are difficult to deploy in practice. (c) Classes C1 and C2 cannot satisfy R3, as many obfuscation/polymorphism techniques can evade them. Particularly, static information flow analysis cannot properly handle dynamic scripting languages like JavaScript as many runtime actions cannot be determined statically [21, 25]. (d) Finally, Class C3 cannot satisfy R4 as they usually pose big overhead.

**Key Insights and Our Approach.** Motivated by the limitations of existing defenses and to satisfy the protection requirements, we propose *aspect-level browser extension behavior clustering*.

We aim to generate alerts for suspicious extensions based on behavior characteristics. In this paper, System Call Dependence Graphs (SCDGs) are used as a representation of behaviors for extensions. We then decompose an extension's runtime behavior into several pieces, denoted as Aspects of Extension Behavior (AEBs). Basically, each AEB corresponds to a unique (sub)SCDG. We aim to group similar AEBs of different extensions into an "AEB cluster". As a result, each extension is mapped to a vector of AEB clusters, which we call the *extension profile*. On a commodity browser's webstore, the extensions are organized by categories; so each category can also be mapped to a vector of AEB clusters, which we call the *category profile*.

A key observation is that extensions in the same category have similar behaviors as they implement similar functionality. Hence, the detection of suspicious extensions is based on the following rationales. First, uniqueness. Each category in the webstore has a unique functionality. A category's functionality correlates to a unique category profile. Second, inclusiveness and exclusiveness. Using a large set of training extensions, we can build a representative profile for each category, meaning that most of the legitimate AEB clusters will be included in each category's profile. However, a suspicious extension bearing different functionality will generate its unique vector of AEB clusters, and thus lead to a unique extension profile, which is not a subset of its category's profile.

Based on this insight, we aim to help (augment) the human review process, as a "safety checker", as follows: whenever a new extension (which might be malicious) is submitted for adoption by Category $C$, the reviewers or the end-users can firstly use our system to map the extension to a particular vector of AEB clusters and generate this extension's profile. If this extension's profile is not a subset of $C$'s profile, an alert may be raised. The users or reviewers can then look into it and decide whether or not to install this extension.

**Main Use Cases of Our Approach.** In general, there are two primary concern holders for the usages of detecting suspicious extensions, end-users and

webstores. Our approach can be both used by these two concern holders. The two main use cases of our approach should be as follows. (a) Webstores can use our approach to do cost-effective safety check of uncertified extensions submitted by third party developers; (b) A trustworthy web portal, e.g., one operated by governments or authoritative organizations, can be set up to allow end-users to upload and check the safety of any extensions through simply a couple of clicks.

Though this work is not the first to apply behavior clustering in the security field [4, 16], this is still the first attempt to employ it into detecting suspicious browser extensions, which is a rather different story with others. Overall, this work makes the following contributions:

- To the best of our knowledge, this is the first study to cluster web browser extensions based on Operating System level runtime behaviors.
- This is the first attempt to apply symbolic execution into the study of web browser extensions. By increasing the input space coverage, the detection rate of suspicious extensions is greatly improved.
- We introduced new methods to address the differentiating of system call traces between the "host" browser and extensions. This greatly improves the accuracy of clustering and detection results.
- We dramatically increased the **scale** of dynamic analysis of browser extensions from around 20 (extensions per study) in the literature [3, 8, 20, 21] to more than 1,000 extensions in our study. Although static analysis [2, 29] of over 1,000 extensions can be done in a rather efficient way, dynamic analysis of over 1,000 extensions is a totally different "story".
- We evaluate our approach atop the Mozilla Firefox browser. The experiment results using large amount of training and testing dataset extensions show that our approach can effectively and efficiently cluster the existing extensions and detect suspicious ones.

## 2 Issues Associated with Browser Extensions

In this section, we discuss two major security issues with extensions, the breach of sandboxing policy for extensions and the hidden/masquerading extensions.

**Breach of Sandboxing Policy.** Due to the functionality, some extensions may contain native libraries and call corresponding APIs so that they can access browser resources while other scripts are usually restrained [12]. This feature may expose users to the threat of information leaks. Scripts that run on web pages conform to certain constraints, e.g., Same Origin Policy (SOP); however, extensions can read and alter web pages, and execute with full or similar privileges as the browser, meaning that they are not restricted by SOP. With these privileges, extensions, if malicious, can put users under security risks. For example, a common practice found in many extensions is using *XMLHttpRequest* to download JavaScript or JSON from a remote web site [24]. Once downloaded, extension authors proceed to "use *eval*() to decode the string content into JavaScript objects". This is dangerous because the decoded JavaScript has full chrome privileges and can perform unpredictable malicious actions [8, 24].

**Hidden and Masquerading Extensions.** An extension can hide itself from the browser's extension manager via the *install manifest* or CSS [5]. Thus, the extension can steal the user's credentials, create sockets, and even delete user's files though this is rarely seen. Extensions can also hide their behaviors

by pretending to be legitimate ones. One example is FormSpy (2006), which is actually a downloader-AXM Trojan, but masquerades as the legitimate NumberedLinks 0.9 extension. It can steal passwords and e-banking login details, forwarding them to a third party web site [22].

## 3 Problem Statement and Behavior Representation

### 3.1 Problem Statement

Currently, neither webstores nor users can distinguish benign extensions from malicious ones. There misses a bridge among developers, users, and webstores. Users need a reliable checker to know what exactly an extension has done and how it deals with the data and personal information. We aim to let users know this before they install a specific extension through our approach.

Specifically, the problem statement is as follows. First, how to provide detailed behavior indicators to the users? Second, how to generate alerts based on behavior characteristics of extensions? Third, how to represent behavior so that meaningful analysis can be done? This representation should also reflect the functionalities and features of those extensions. Fourth, how to do the above things in an automatic way, so that human involvement can be minimized?

### 3.2 Behavior Representation

A proper representation of behaviors for extensions should be determined first. We represent behavior using a particular graph called SCDG. In our model, the behavior (of an extension) is represented by a set of disconnected SCDGs. Each SCDG is a graph in which "system calls are denoted as vertices, and dependencies between them are denoted as edges" [30]. A SCDG essentially shows the interaction between a program and its operating system. This interaction is an essential behavior characteristic of the program in concern [30, 31]. We formally define SCDG as follows [30, 31].

**Definition 1.** *System Call Dependence Graph. Let $p$ be the running program (say extension). Let $I$ be the input to $p$. $f(p, I)$ is the obtained system call traces. $f(p, I)$ can be represented by a set of System Call Dependence Graphs (SCDGs) $\bigcup_{i=0}^{n} G_i$: $G_i = \langle N, E, F, \alpha, \beta \rangle$, where*

- *$N$ is a set of vertices, $n \in N$ representing a system call*
- *$E$ is a set of dependence edges, $E \subseteq V \times V$*
- *$F$ is the set of functions $\bigcup f : x_1, x_2, ..., x_n \rightarrow y$, where each $x_i$ is a return value of system call, $y$ is the dependence derived by $x_i$*
- *$\alpha$ assigns the function $f$ to an argument $a_i \in A$ of a system call*
- *$\beta$ is another function assigning attributes to node value*

In our model, the behavior of an extension has several aspects. We define Aspect of Extension Behavior (AEB) as follows.

**Definition 2.** *Aspect of Extension Behavior. Let $p$ be the running extension. $G = \langle N, E, F, \alpha, \beta \rangle$ is one SCDG for $p$. If $\exists\ G' \subseteq G$ such that $G'$ can represent what $p$ has done and accessed, we say that $G'$ is an Aspect of Extension Behavior (AEB) for $p$.*

An AEB is a subgraph of a SCDG. Each AEB corresponds to a unique (sub)SCDG. Consequently, the behavior of an extension can be decomposed into a set of AEBs. Representative AEBs include "bookmark accessing", "DOM storage accessing", "form submitting", "Cookies reading", and "Downloading", etc.

### 3.3 Why Use SCDG and AEB as the Representation of Behavior?

**Why System Calls?** We perform system call tracing on browser extensions for several reasons. First, system calls are the only interface between OS and a program, providing the only way for a program to access the OS services. Second, almost every attack goal is bundled with OS resources. Hence, for malicious extensions, it is usually not possible for them to conduct malicious actions without triggering system calls, even if they use obfuscation or polymorphism techniques [18, 30]. Third, though the attacker can use compiler optimization techniques to camouflage an extension, these tricks usually do not change dependencies between system calls [30]. In addition, system calls can be practically tracked and analyzed, while giving little overhead to the browser and OS.

**Why SCDGs and AEBs?** SCDGs are employed based on the following observation and insight. A single system call trace tells little information about the overall behavior of an extension directly, as system calls are low level reflection about the behavior characteristics of a program. A problem occurs how to map the low level system call traces with application level behavior. An intermediate representation is required to correlate them. SCDGs can appropriately reflect the dependencies between system calls. They are the abstraction of a sequential system calls. To connect SCDGs with application level behavior, we then introduce AEBs in this paper. Based on the definition, every AEB is associated with a unique (sub)SCDG, while AEBs are the decomposed runtime behavior of an extension. Hence, SCDGs and AEBs can be employed as an intermediate representation of behavior for an extension. AEBs thus can act as a difference between benign and suspicious extensions.

## 4 System Design

### 4.1 Approach Rationale

First, given that most webstores already have a human review process in place for adoption of new extensions (though it is not sufficient enough), our goal is to augment this process and off-shoulder the human reviewer's workload as much as possible. Second, we aim to build a system that can differentiate benign extensions from suspicious ones based on behaviors. An appropriate and accurate representation of extension's behavior can reflect the difference of behaviors between benign and suspicious extensions. Specifically, SCDGs are used to represent the behavior of extensions in system level. They can act as a distinguishing characteristic between extensions. Third, extensions are classified into several categories by extension webstores, such as Bookmarks, Tabs, and Shopping, etc. A basic observation is that extensions in the same category have similar behaviors as they implement similar functionality. SCDGs and AEBs act as the intermediate representation to correlate the system level behavior tracking and application level behavior. If an extension has one outlier AEB that all other extensions (in the same category) do not have, this should be considered as abnormal and suspicious.

### 4.2 System Overview

Fig. 1 shows the architecture of our system. It consists of four components: Dynamic Tracer, SCDG Extractor, SCDG Clustering, and Alert Generator.

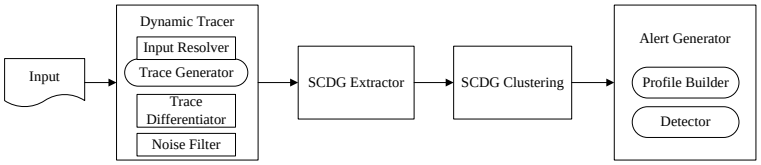


**Fig. 1.** Architecture of our system, which consists of four components: Dynamic Tracer, SCDG Extractor, SCDG Clustering, and Alert Generator.

**Dynamic Tracer.** The dynamic tracer is mainly composed of an input resolver and a trace differentiator. The dynamic tracer tracks the behaviors of both benign and suspicious extensions in the form of system calls, using the input resolver to address the input space issue. The trace differentiator is a component resolving the system call traces of extensions from the host browser.

**SCDG Extractor.** The SCDG Extractor takes the trimmed system call traces of each extension as the input, and aim to generate SCDGs for each extension. It first explores the dependencies between system calls. Then, it identifies objects and encodes them for the use of the following component.

**SCDG Clustering.** SCDG Clustering is used to generate AEB clusters. Specifically, we compare the SCDGs using subgraph isomorphism under the restriction of $\gamma$-isomorphism. To increase the efficiency, we also perform several pruning techniques to reduce the search space and computational complexity.

**Alert Generator.** The alert generator aims to raise alerts for suspicious extensions. This component has two primary functionalities. It first builds profiles for each extension and thereafter the categories. Then, we use the profiles of categories instead of extensions to detect suspicious extensions.

**Challenges.** This system faces several key challenges. The first is the input space issue. We use an input resolver to overcome this challenge. The second hurdle is the differentiating of system call traces between the browser and extensions. As the tracing is conducted per process, we need our tracing to know whether a system call is invoked by a specific extension or the browser. The trace differentiator is employed to handle this. The third one is to identify the relevant objects and encode them when extracting SCDGs. Though exploring dependencies between system calls is not new, for browser extensions, we have to identify relevant objects and encode them to formalize the nodes in SCDGs so that we can do additional pruning techniques in SCDG clustering. A fourth challenge is how to identify suspicious extensions and raise alerts for them. The profile builder acts as the key factor to serve the detection of suspicious extensions.

### 4.3 Dynamic Tracing

Dynamic Tracing is a key challenge in our system. The dynamic tracer takes the browser and extensions as the input, and eventually generates the trimmed system call traces for each extension. It consists of four smaller components: trace generator, input resolver, trace differentiator, and the noise filter. In a nutshell, the trace generator takes the browser and running extensions, and inputs to obtain the system calls. The inputs associated with the trace generator are generated by the input resolver to address the input space issue. Trace differentiator is used to identify whether a system call is invoked by a specific extension in concern. Finally, the noise filter can remove the noises to reduce the workload

of SCDG extraction in the following work. In this subsection, we primarily focus on two key challenges when perform dynamic tracing. We then give a brief introduction to the noise filter.

**Input Resolving.** A first key challenge for dynamic tracing is known as input space issue. An input used by a program (value and event, e.g. data read from disk, a network packet, mouse movement, etc.) cannot always be guaranteed to reoccur during a re-execution. As a result, an extension will result in a set of execution paths due to different inputs, while these execution paths cannot be guaranteed the same during the dynamic tracing. It is very likely that certain malicious actions can only be triggered under specific inputs (i.e., conditional expressions are satisfied, or when a certain command is received). If these specific inputs are not included in the test input space, it is possible that malicious actions can be triggered in a particular execution path.

However, almost none of the prior approaches related to browser extensions have taken input space coverage issue into account [2, 18, 20, 21, 29]. There is a need to automatically explore the input space of client-side JavaScript extensions. Generally, the input space of a JavaScript extension can be divided into two categories: the event space and the value space [28]. Rich browser extensions typically define many JavaScript event handlers, which may execute in any order as a result of user actions such as clicking buttons or submitting forms. The value range of an input includes user data such as form field and text areas, URL and HTTP channels.

To address the input space issue, an input resolver (IR) is used based on dynamic symbolic execution in our paper. The IR can be used to "hit" as many execution paths as possible for an extension. In the IR, symbolic variables are tracked instead of the actual values. Values of other variables which depend on symbolic inputs are represented by symbolic formulas over the symbolic inputs. When a symbolic value propagates to the condition of a branch, it can use a constraint solver to generate inputs to the program that would cause the branch to satisfy some new paths [28].

As our IR is primarily designed based on symbolic execution, we first introduce how symbolic execution works. Suppose that a list of symbols $\{\xi_1, \xi_2 ...\}$ are supplied for a new input value of a program each time [17]. Symbolic execution maintains a symbolic state, which maps variables to symbolic expressions, a symbolic path constraint *pc*, and a Boolean expression over the symbolic inputs $\{\xi_i\}$. *pc* accumulates constraints on the inputs that trigger the execution to follow the associated path. For a conditional `if (e)` $S_1$ `else` $S_2$, *pc* is updated with assumptions on the inputs to choose between alternative paths [6, 33]. If the new control branch is chosen to be $S_1$, *pc* is updated to $pc \wedge \mu(e) = 0$; otherwise for $S_2$, *pc* is then updated to $pc \wedge \mu(e) \neq 0$. $\mu(e)$ denotes the symbolic predicate obtained by evaluating $e$ in symbolic state $\mu$. In symbolic state, both branches can be taken, resulting in two different execution paths. Symbolic execution terminates when *pc* is not satisfied. Satisfiability is checked with a constraint solver. For each execution path, every satisfying assignment to *pc* gives values to the input variables that guarantee the concrete execution proceeds along this path. For code containing loops or recursion, one needs to give a limit on the iteration, i.e., a timeout or a limit on the number of paths [6, 17, 33].

Specifically, the IR includes a dynamic symbolic interpreter that performs symbolic execution of JavaScript, a path constraint extractor that builds queries based on the results of symbolic execution, a constraint solver that finds satisfying assignments to those queries, and an input feedback component that uses the results from the constraint solver as new program inputs [28]. They are used to generate values to "hit" as many paths as possible.

On the other hand, a unique challenge for extensions is the event space issue. Our IR can address the issue of detecting all events causing JavaScript code execution as follows. First, a GUI explorer will search the space of all events using a random exploration strategy. Second, an instrumentation of browser functions can process HTML elements to record the time of the creation and destroy of an event handler [28]. Ordering of user events registered by the web page is randomly selected and automatically executed. The same ordering of events can be replayed by using random seed. The explorer also generates random test strings to fill text fields when handlers are invoked [28].

**System Call Differentiating.** The other big challenge is the differentiating of system call traces between the browser and extensions. Different browsers have adopted various extension system mechanisms, posing great challenge to the tracing of system calls. For Firefox, all extensions and the browser itself are wrapped into a single process. This poses great challenge to differentiate all the running extensions from the browser: First, how does one differentiate system calls between the browser and extensions? Second, how does one differentiate system calls among various extensions?

To address this, we introduce fine-grained system call tracing. When executing, extension and browser JavaScript are interpreted by JavaScript Engine and connect XPCOM through XPConnect. An important issue is extension JavaScript can access to the resources through browser APIs. Therefore, a possible way is to track or intercept the functions to distinguish the real callers of system calls. Prior approaches have been proposed to track those functions [1, 2]. Functions can give cues with respect to when a function is entered and exited, and where the function is called from. Through these runtime call tree we can differentiate the system calls between web browser and extensions.

Specifically, we use Callgrind, which is based on Valgrind [14, 15]. Callgrind uses runtime instrumentation via the Valgrind framework for its cache simulation and call-graph generation [26]. It can collect the caller/callee relationship between functions. It maps a subroutine to the component library which the subroutine belongs to. Hence, if a subroutine in the execution stack is called from the component library during the execution of an extension and the browser, it will be marked [31]. Therefore, it can dynamically build the call graph generated by web browser and extensions. To increase the accuracy of system call differentiating, we also add a *timestamp* for each call. The delay between the time of system call trace and the timestamp is too small to be counted. The timestamp can help quickly locate the system call traces of extensions and remove unnecessary system call traces.

To completely remove the interference from other extensions, we tend to run just one extension while disabling all other irrelevant installed extensions. This definitely reduces the possibility of parallel processing. However, two reasons can support this practice. First, each system call tracing occupies very little time,

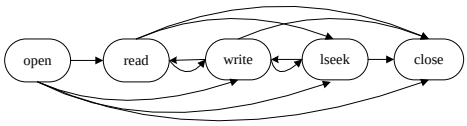


**Fig. 2.** Possible dependencies among system calls of file management.

which we will see it in the evaluation section. Running one extension exclusively will not reduce much of the speed in our approach. Second, this practice will greatly improve the accuracy of the system call trace differentiating, serving better in detecting suspicious extensions in later components.

*Noise Filtering Rules.* First, we neglect system calls that do not represent the behavior characteristics we want, e.g., system calls related to memory management, page faults, and hardware interrupts [7, 30]. We will discuss why we neglect them in details in the evaluation section. Second, system calls with very similar functionality are considered the same. For example, *fstat(int fd, struct stat *sb)* system call is very much the same as *stat(const char *path, struct stat *sb)* [30]. Third, failed system calls are ignored [30, 31].

### 4.4 SCDG Extracting

A SCDG is determined by two parts, nodes which are system calls and edges which are dependencies, respectively. We mainly focus on how to derive dependencies between system calls and how to do object encoding on nodes.

**Dependencies between System Calls.** An entry in the system call trace is composed of a system call name, arguments, return value and time, etc. Obviously, arguments of a system call are dependent on previous system calls. There are two types of data dependence between system calls. First, there will be a data dependence if a system call's argument is derived from the return value(s) of previous system calls. Second, a system call can also be dependent on the arguments of previous system calls [18]. Fig. 2 shows an example of the possible dependencies among system calls of file management [9]. System call *read* is dependent on *open* as the input argument of *read* is derived from the return value of *open* - the file descriptor.

In the definition of SCDG, we mention that $\alpha$ assigns function $f$ to $a_i$ to a system call. Here, $f$ is a function to derive dependencies between system calls. Specifically, for an argument $a_i$, $f_{a_i}$ is defined as $f_{a_i} : x_1, x_2, ..., x_n \rightarrow y$, where $x_i$ denotes the return value or arguments of a previous system call , $y$ represents the dependence between $a_i$ and these return values. If $a_i$ of a system call depends on the return value or arguments of previous system call, an edge is built between these two system calls.

**Objects Identifying and Encoding.** A challenge related to node derivation function $\beta$ in the definition of SCDG is to identify related *objects*. In this paper, *objects* include related OS resources and services, browser resources, network related services, and files, etc. In Linux, we divide those related *objects* into an *object tree* as shown in Fig. 3. Under a particular parent node, each child node represents an *object*. From left to right sibling node, each is represented by a natural number as in Fig. 3. Thus, each node can be represented by the numbers from root to its parent node and to this node. Hence, each node corresponds to a unique code, which we call *object code*. This process is called *objects encoding*.

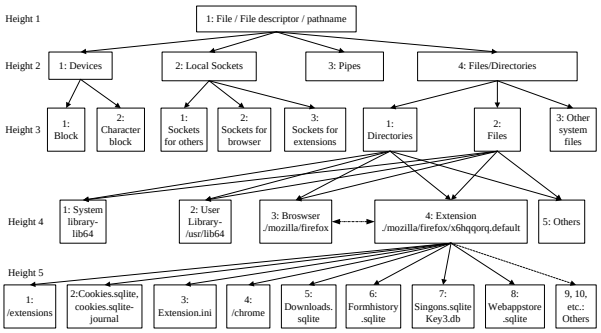


**Fig. 3.** Object tree shows related objects and object encoding.

For each particular argument $a_i$ of a system call, we search it by traversing the object tree using depth-first-search algorithm. If found, retrieve the *object code* for $a_i$ by backtracking to the root. Take "Files" in height 3 for example. It will be denoted as 1.4.2, where 1 represents the root, 4 represents the parent object, and 2 represents the object itself.

We build an object tree and assign each node with an object code primarily for three reasons. First, each argument of a system call trace usually contains a long string of characters. Using object code, we can formalize and simplify each node. Second, simplifying node value can improve the efficiency when doing subgraph isomorphism analysis. Compared with raw node values, checking each node with simple object code will reduce the time consumption. Fig. 3 lists most of the related objects under the browser profile and the extension. Due to space limit, we place some sensitive objects into others including $XUL.m, xpti.dat, urlclassifierkey3.txt$, etc. Besides node derivation, another important application is using the object tree to **identify AEBs**. Through the object tree, AEBs can be identified by (sub)SCDGs with real-world meaning related to browser extensions, such as "form submitting" and "Cookies accessing".

### 4.5 SCDG Clustering

We use subgraph isomorphism to compare SCDGs and group them into AEB clusters. We first define some terminology regarding graph isomorphism [30, 31].

**Definition 3.** *Graph/Subgraph/$\gamma-$Isomorphism. Suppose there are two SCDGs $G = \langle N, E, F, \alpha, \beta \rangle$ and $H = \langle N^{'}, E^{'}, F^{'}, \alpha^{'}, \beta^{'} \rangle$, where dependence edge $e \in E$ is derived from $(F, \alpha)$. A graph isomorphism of $G$ and $H$ exists if and only if there is a bijection between the vertex sets of $G$ and $H$: $f : N \to N^{'}$ such that any two vertices $u$ and $v$ of $G$ are adjacent in $G$ if and only if $f(u)$ and $f(v)$ are adjacent in $H$, which is represented as $G \simeq H$. Specifically,*

- $\forall n \in N, \beta(n) = \beta(f(n))$,
- $\forall e = (u, v) \in E, \exists e^{'} = (f(u), f(v)) \in E^{'}$, *and on the contrary,*
- $\forall e^{'} = (u^{'}, v^{'}) \in E^{'}, \exists e = (f^{-1}(u^{'}), f^{-1}(v^{'})) \in E$

*Particularly, if*

- $\exists\ H_1 \subset H$ *such that* $G \simeq H_1$, *we say that a subgraph isomorphism exists between $G$ and $H$.*

- $\exists\ H_1 \subset H$ *such that* $G \simeq H_1$ *and* $|H_1| \geq \gamma|H|$*, where* $\gamma \in (0,1]$*, we say that* $H$ *is* $\gamma-$*isomorphic to* $G$*.*

In principle, a large amount of pairs of subgraph isomorphism testing are required. However, we can perform some pruning techniques to reduce the search space and computational complexity. First, based on the definition of $\gamma$-isomorphism, a SCDG pair $(g, g')$ can be excluded if $|g'| < \gamma|g|$, where $g'$ and $g$ are SCDGs from different extensions. Second, although subgraph isomorphism is an NP-complete problem, it has shown that some algorithms are fast in practice, which are based on backtracking and look-ahead algorithm [30], e.g., the VF algorithm which is suitable for graphs with a large number of nodes. In this paper, we use an optimized VF algorithm called $VF2$ subgraph isomorphism algorithm to compare SCDGs [10]. Third, SCDGs obtained and optimized are not ordinary graphs. They bear special characteristics which can help reduce the computational complexity. We have encoded the nodes to make it more efficient to perform backtrack-based isomorphism.

After performing the VF2 algorithm, SCDGs will be grouped into different clusters. Each cluster is called AEB cluster. They are defined as follows.

**Definition 4.** *AEB Cluster. Let* $P$ *be the training set extensions,* $G_i$ *be a vector of SCDGs derived from* $p_i \in P$*, where* $(i = 0, 1, 2, ...)$*. If* $\exists g_j \in G_i \& g_j^{'} \subset g_j \& |g_j^{'}| \geq \gamma|g_j|$ *such that* $g_0^{'} \simeq g_1^{'} \simeq ... \simeq g_m^{'}$*, where* $\gamma \in (0,1]$*, we say that an AEB Cluster is constructed and represented by* $\langle g_0^{'}, g_1^{'}, ..., g_m^{'} \rangle$*.*

Each AEB cluster is actually a set of (sub)SCDGs, corresponding to one particular AEB. As a result, ***each extension should fall into multiple AEB clusters***.

### 4.6 Alert Generating

So far, we can get the AEB clusters for each extension. However, how these AEB clusters serve security purposes, namely, detecting suspicious extensions is not presented yet. Alert generator acts as the last component in connecting those AEB clusters with extensions and their categories in detecting suspicious extensions. Specifically, we compare the profile of a to-be-examined extension with the profile of the category that this extension belongs to. The rationale is that the extension's profile should be a subset of its category's profile.

We define the profile of an extension as follows.

**Definition 5.** *Extension Profile. For an extension* $p$*,* $AC$ *is the corresponding vector of AEB clusters derived from behavior clustering. Then, the profile of* $p$ *can be represented as* $\langle p, AC \rangle$*.*

Following the same spirit, we define the profile of a category as follows.

**Definition 6.** *Category Profile. For a category* $C$ *in the extension webstore, its profile is the union of the extensions' (in category* $C$*) profiles, represented as* $\bigcup_{i=0}^{n} \langle p_i, AC_i \rangle$*, where* $p_i \in C$*.*

In this paper, we use the profiles of categories instead of extensions to detect suspicious extensions. It does not make much sense to directly compare the

profiles of two extensions, even if they are in the same category. None of the extensions can represent the overall functionality of this category, and thus their profiles can vary much to some degree.

Therefore, based on the detection rationales mentioned in the introduction (uniqueness and inclusiveness/exclusiveness), we use profiles of categories correlates existing categories and AEB clusters to detect suspicious extensions as follows. For a to-be-examined extension belonging to category $C$, if its profile is not a subset of $C$'s profile, we consider this extension as a suspicious one, and those outlier AEB clusters are called *suspicious AEB clusters*. An alert will be raised and those suspicious AEB clusters will be presented to the users. The users can then look into these AEB clusters and decide whether to install it.

## 5 Implementation

We implemented a system call tracing tool strace++ based on strace [11, 19]. Strace++ can track the system calls with a given time and filter off the unnecessary system calls. Our input resolver is primarily based on Kudzu [28]. We modified it to employ it on the web browser and generate inputs for strace++. Our trace differentiator employs Callgrind under Valgrind. We also implemented the SCDG extractor under Valgrind. The SCDG extractor constructs SCDGs based on the following functionality. When a system call of an extension is invoked, it can construct a new node and dependencies between system calls. The SCDG extractor then formalizes the node by identifying the objects and encoding them. Thus, SCDGs can be extracted [30, 31]. We adopted the subgraph isomorphism and $\gamma$-isomorphism based on $VF2$ algorithm of NetworkX [27].

## 6 Evaluation

Regarding the 4 protection requirements raised in Section 1, R2 has already been satisfied due to the design of our system. So we evaluate our system in this section with respect to R1, R3 and R4. Basically, we have three evaluation goals: (G1) What is the effect of the input resolving on input space issue? (G2) Whether our approach can identify suspicious extensions effectively? (G3) Can our approach perform efficiently and scalably?

### 6.1 Evaluation Environment

Our experiments were performed on a workstation with a 2.40 GHz Quad-core Intel(R) Xeon(R) CPU and 4GB memory, under Fedora 12. $\gamma$ is set to be 0.8. We use Firefox 3.6 as the host browser, as it is one of the most stable versions among various Firefox versions. We have examined 1,293 extensions in total for training and testing extensions (including malicious and new extensions).

### 6.2 What is the Effect of Input Resolving on Input Space Issue?

Two questions need to be answered to evaluate the effectiveness of our input resolver (IR). First, will there be a significant increase in execution paths and input after using the IR? Second, will there be any outliers for execution paths and input without the IR? If so, is the percentage of outliers acceptable? With the IR, we can get the times of execution, input, and system call traces. However, without the IR, we can only get the system call traces. Hence, it is hardly possible for us to directly compare the times of execution and input. Thus, we

**Table 1.** Comparison on Input Space with and without IR

| Category | # of ext. | # of SCDGs w/o IR | # of SCDGs w/ IR | outlier |
|---|---|---|---|---|
| alert | 15 | 454 | 670 | 3 |
| bookmark | 19 | 720 | 1064 | 5 |
| download | 18 | 623 | 1085 | 2 |
| shopping | 20 | 640 | 956 | 0 |

can compare the system call traces as they can directly reflect the times of execution and input. However, it is still difficult and impractical to compare them among thousands of them. Therefore, we evaluated our IR by comparing SCDGs as they can also reflect execution paths and the input to a large degree.

Specifically, we have evaluated our IR from two perspectives based on SCDGs. First, is there a considerable increase in the total number of SCDGs after employing the IR? Second, are there any outliers of SCDGs after employing the IR? Table 1 shows the results without and with applying the IR on the browser. We have selected four categories and 72 extensions in total as the representatives. The third and fourth columns show the total numbers of SCDGs for extensions in the same category with and without the IR. On average, there is a significant 54.8% increase in the total number of SCDGs after using the IR. On the other hand, if a SCDG before using the IR is not included in the set of SCDGs after using the IR, we call it an outlier. The last column shows the total number of outliers for each category. On average, 0.4% of previous SCDGs are outliers, which we think is a very small amount of percentage. Outliers are most likely caused by the different parameters of graphs. This basically does not impact much on the follow-up clustering as we use $\gamma$-isomorphism. Not only can our IR increase the total number of SCDGs substantially, but it can also control the outliers in a very small range.

### 6.3 Can Our System Identify Suspicious Extensions Effectively?

To evaluate the effectiveness of our system in detecting suspicious extensions, we first present the training extensions dataset and the clustering results. We then use the testing extensions to evaluate our system.

**What does the Training Dataset Look Like?** In total, we extract SCDGs for 1,107 training set extensions. Table 2 shows the training set statistics for each category we examined. There are more than ten categories for Firefox extensions; however, we choose 8 categories from them based on the following criteria: downloads and representative categories for malicious extensions.

The unfiltered system call traces (SCTs) we obtained vary from 70,000 to 200,000. Based on our filtering rules, the average percentage of filtered SCTs is 32.4%. Here, we find that up to 98.4% of the filtered system calls related to memory management belong to the browser other than extensions. So it is impractical and makes little sense to include the memory management system calls in our dynamic tracing. The training dataset clearly shows that our trace differentiator can greatly decrease the SCTs for an extension, which is only 17.2% of the filtered SCTs. In the training set, each SCDG usually has hundreds of nodes and edges. Fig. 4 is a subgraph of the SCDGs from one famous Firefox extension FoxTab. It clearly shows the attributes of each node and dependencies between nodes. Take the first node $N1(stat; 1.4.2.4.4)$ as example. The system call stat

**Table 2.** Training Set Extensions Statistics

| Category | alert | bookmark | download | feed | privacy | social | shop | search |
|---|---|---|---|---|---|---|---|---|
| # of extensions | 135 | 154 | 103 | 150 | 130 | 150 | 145 | 140 |
| # of avg. raw SCT | 132,146 | 130,545 | 172,208 | 112,062 | 102,865 | 143,066 | 154,971 | 146,053 |
| # of avg. SCT filt. | 89,205 | 90,416 | 112,782 | 71,628 | 72,386 | 93,052 | 110,495 | 98,821 |
| # of avg. ext. SCT | 15,220 | 17,832 | 21,435 | 14,451 | 11,890 | 13,547 | 16,155 | 16,072 |
| # of avg. SCDG | 44 | 58 | 56 | 61 | 45 | 42 | 51 | 52 |
| # of AEB clusters | 46 | 53 | 44 | 55 | 48 | 43 | 47 | 56 |

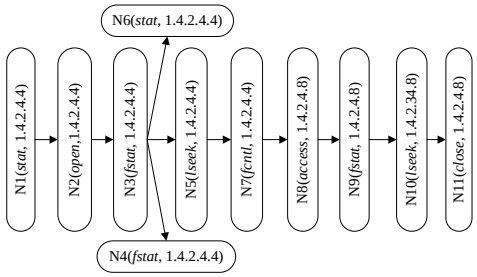


**Fig. 4.** One sub-SCDG extracted from the extension FoxTab, showing the dependence graph of the system calls. Each node consists of two parameters, system call name and the code for this system call. It is also one member subgraph of the "DOM Storage Accessing" AEB cluster.

with the code 1.4.2.4.4 means *Chrome accessing*. Usually, for each particular extension, there are 30 to 80 SCDGs if excluding repetitions.

**What do the Clustering Results and Category Profiles Look Like?** We then compare SCDGs using subgraph/$\gamma$-isomorphism. We finally aggregated SCDGs into AEB clusters. Fig. 4 also shows a member subgraph of the "DOM Storage Accessing" AEB cluster for Foxtab. This AEB cluster includes hundreds of SCDGs, one from each extension, as many extensions need this AEB to access the DOM storage. If one SCDG or sub-SCDG is the only one in this category after clustering, we will manually check whether it is a malicious one to guarantee the ground truth of the training set.

Based on the definition of category profile, each category can be mapped to a vector of AEB clusters. Table 2 shows that each category usually has a number of AEB clusters from 30 to 60. For example, for "Download" category profile, the AEB clusters are as follows: "chrome context accessing", "language pack retrieving", "file system checking", "webappstore.sqlite accessing", "webappstore.sqlite modifying", "nsIXMLHttpRequest", "nsIHttpChannel", "socket opening", "nsIDownloader accessing", "DOM Storage accessing", "nsIInputStream", "download.sqlite opening", and "download.sqlite modifying", etc.

**What does the Testing Dataset Look Like?** There are 186 extensions in our testing set, including 8 existing malicious extensions and 1 malicious extension written by us. Table 3 shows the statistics. There is a slight difference in the number of AEB clusters between training set and testing set. So are there any suspicious AEB clusters that deviate from the category profiles?

**What are the Resulting Suspicious AEB Clusters?** To answer this question, we use our detection rules to examine the AEB clusters of those ex-

**Table 3.** Testing Set Extensions Statistics and Results

| Categories | alert | bookmark | download | feed | privacy | social | shop | search |
|---|---|---|---|---|---|---|---|---|
| # of ext. | 20 | 25 | 24 | 25 | 25 | 22 | 25 | 20 |
| # of average ext. SCT | 16,925 | 17,946 | 22,531 | 16,013 | 10,462 | 9,952 | 13,674 | 11,895 |
| # of average SCDG | 42 | 53 | 54 | 65 | 44 | 47 | 58 | 50 |
| # of AEB clusters | 50 | 55 | 48 | 54 | 52 | 45 | 46 | 51 |

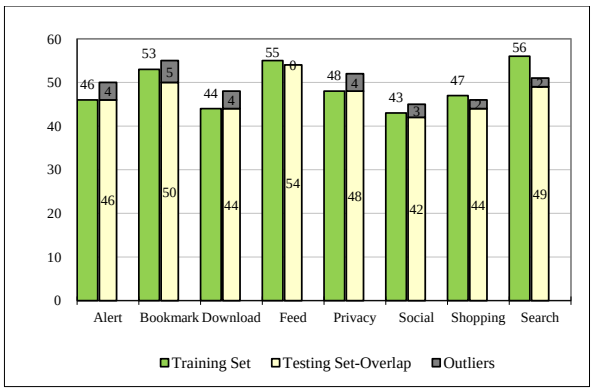


**Fig. 5.** The number of AEB clusters for training set and testing set including outliers.

tensions. Fig. 5 clearly shows a comparison between the training set and testing set in the number of AEB clusters corresponding to each category. Most AEB clusters of the testing set belong to the category profiles. However, 7 of 8 categories have outliers, namely suspicious AEB clusters. On average, there are 6.0% of suspicious AEB clusters in the testing set.

Table 5 presents the detailed information for 5 extensions, including 4 existing malicious extensions and 1 malicious extension written by us. Note that the extensions in Table 5 do not represent all the detection results. They are just 5 of 10 extensions which are detected as suspicious. Facebooker is said to provide status updates to users; however, in the back end, it can download files stealthily. Let us analyze the results shown in Table 5. The column of "suspicious AEB clusters" shows the suspicious AEB clusters presented to the users. The suspicious AEB clusters of FormSpy and FFsniFF are "form action", "form submission", "formhistory.sqlite accessing", and "nsIHttpChannel". Particularly, for FormSpy, "ID masquerading" is detected as suspicious by the system. As mentioned before, FormSpy would forward sensitive information the user submitted to a third party web site. Similarly, FFsniFF can find form and send it to a specified email. The suspicious AEB clusters for FireStarterFox are "data submission" and "unknown URL injection". FreeCF is posted as a shopping coupon, but actually it can cause Facebook scams. Its suspicious AEB clusters are "script loading" and "unknown server accessing". For the extension written by us, Facebooker is successfully detected as a suspicious one with suspicious AEB clusters "unknown downloads", "downloads.sqlite opening" and "nsIDownloader Accessing".

**False Negative and False Positive Analysis.** In the testing set, 10 extensions are detected as suspicious ones, while the other 176 extensions are regarded as benign with no suspicious AEB clusters. Among the 10 suspicious extensions,

**Table 4.** Results of 5 Example Extensions Drawn from Testing Set.

| Testing set | version | SCDG | category | suspicious AEB clusters |
|---|---|---|---|---|
| FormSpy | N/A | 24 | bookmarks | ID masquerading, form submission, nsIHttpChannel, form action, formhistory.sqlite accessing |
| FFsniFF | 0.3 | 14 | privacy | form action, form submission, nsIHttpChannel, formhistory.sqlite accessing |
| FireStaterFox | 1.0.2 | 17 | search | data submission, unknown URL injection |
| FreeCF | 0.1 | 12 | shop | script loading, unknown server accessing |
| Facebooker | 1.0 | 19 | social | downloads, nsIDownloader accessing, downloads.sqlite opening |

8 are the malicious extensions we provided, 1 is the malicious extension we wrote. To thoroughly evaluate the false negatives, we manually examined the remaining 176 extensions. Basically, as most of them are small programs, we examine the source code and compare them with the functionalities they claim. So far, we find them benign with no malicious actions. This means all the 9 malicious extensions are detected without any false negatives, meaning **the false negative rate is 0%** using the test set, demonstrating the effectiveness of our system on detecting suspicious extensions. This is reasonable, as a large pool of training extensions enable more accurate clustering results.

However, in the results, 1 of the suspicious-regarded extensions is actually a false positive after we manually check the source code. We examine it on the webstores, and find that it bear a distinct feature. It belongs to more than one category with a larger range of functionalities (this is possible, but not many). Specifically, the extension named MailAlert belongs to both Alert and Feed categories. It provides mail account alert and news feed. Consequently, this extension's profile may not be the subset of the Alert category profile. In this case, a false positive may occur as our detection rules are restrictive when MailAlert is regarded as only belonging to the Alert category during detecting. Though the false positive rate seems a little higher (10%) in our testing, it is not the real case for the webstores, as only less than 1% of extensions belong to more than one categories. In fact, we provide two alternatives to address this issue. Before examining an extension, we first check whether it belongs to more than one categories or not. If so, we examine it combining all the category profiles it belongs to. Basically, this is the primary and regular approach as done with other extensions, which can eliminate most of the false positives. In addition, manually checking its code can reduce the false positive sharply while this practice is not recommended as the first alternative can satisfy the basic requirements.

### 6.4 Efficiency and Scalability

We tend to provide a cost-effective online service for both helping the certification of webstores and the safety check of any extensions submitted by public users. Hence, we discuss the efficiency of our approach, specifically, the time consumed when employing each component. Fig. 6 shows the average time consumed by eight categories during dynamic tracing, SCDG extracting, and SCDG clustering. Each category corresponds to one in Table 2 from left to right. Dynamic tracing takes up to 48.4% of the total average time, which is 56.5 seconds on

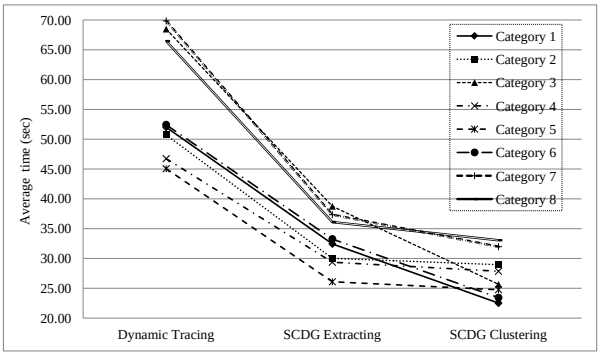


**Fig. 6.** Avg. time consumed by the eight categories during the first three components.

average for all the categories. SCDG extracting and SCDG clustering cost 32.9 seconds and 27.3 seconds on average, respectively. The total consumed time for the three evaluated components is 116.7 seconds on average, which is a reasonable time performance. For the profile building and detecting, it is usually very natural and easy once we have completed the previous work. As a result, the time consumed by them can be neglected. Hence, our approach can be efficiently used to detect suspicious extensions for the use of both end-users and webstores.

Scalability is another important factor to evaluate the detection approach. Unlike other dynamic analysis approaches, our approach can scale from 20 extensions to over 1,000 extensions, due to several reasons: (a) the symbolic executions of multiple extensions are independent of each other, so they can be done in a parallel manner; (b) the system call tracking of multiple extensions are to a large extent independent of each other; (c) although the VF2 algorithm has an exponential complexity, which does not directly indicate superb scalability, our experiments show SCDG clustering consumes the least amount of absolute time.

## 7 Discussion and Limitations

There are several limitations and counterattacks while employing behavior clustering into detecting suspicious extensions. First, although we have a fine-grained technique to differentiate system call traces between the browser and running extensions, it is still possible that we mix system call traces between them. Let us take a clear look at the two possible mistakes. The first possibility is that system call traces of the running extension may be treated as the browser's. This may eliminate some SCDGs for this single extension. We use a large number of extensions in the same category to build the category profile instead of each extension; hence, the first possibility can rarely affect the detection results. The other possibility is that system call traces of the browser may be treated as the running extension. However, when extracting SCDGs from the set of system call traces for this extension, most of the mistaken ones will be excluded. Therefore, this possibility also affects little on SCDG extraction.

Second, as a common limitation for system call tracing, it is not applicable if the running program invokes no system calls. This is possible for some simple

extensions such as some arithmetic operations [30]. However, this rarely happens on malicious extensions, as most malicious actions would invoke system calls.

Third, one may consider developing a malicious extension that implements its behaviors in a different way to evade the system. However, such kind of mimicry attack is very difficult to implement. In our SCDGs, each node separates itself from other nodes through two things: system call name and object code. To make a successful mimicry attack, the attacker needs to mimic not only the system call name, which is sometimes quite easy [32], but also the object code. Most malicious extensions have to access objects that are different from those accessed by others in the same category. Hence, some object codes must be different and so are some nodes in some SCDGs. On the other hand, the attacker can always let extensions do more, i.e., accessing more objects than needed. In this way, a malicious extension can access the objects accessed by the others in the same category. However, this kind of "object mimicry attack" usually cannot satisfy the attackers requirements. In addition, to successfully mimic an attack, the attacker also needs to consider the dependencies besides nodes. Even if several system calls are reordered, it cannot change the results of SCDGs and subgraph isomorphism as we use $\gamma$-isomorphism to cluster extensions.

Finally, our system has a limitation when malicious extensions inject JavaScript into pages rather than carrying out malicious actions directly. Currently we do not track those injected JavaScript pages, so we do not know whether they have done some malicious actions or not. However, in future work, our system can be modified to first identify possible injections and then track both the injections and extensions. As many of the injections relate to "alerting" a new window, an injection of "url" or "image", accessing the cookies, etc, the system should pay particular attention to them to detect possible malicious actions.

## 8 Related Work

**Static Analysis.** Static analysis is used to identify malicious extensions via analyzing JavaScript code statically including objects and functions without executing the programs [2, 29]. Bandhakavi *et al.* [2] proposes VEX to exploit the extension vulnerabilities using static analysis. They describe several flow patterns as well as unsafe programming practices, particularly regarding some crucial APIs, which may lead to privilege escalation in JavaScript extensions. VEX analyzes extensions for these flow patterns using context-sensitive and flow-sensitive static analysis. This approach can address some crucial security issues. However, it is very difficult to employ this on dynamic scripting languages like JavaScript in extensions. A well-known example is the *eval*() statement in JavaScript that allows a string to be evaluated as executable code. Without knowing the runtime values of the arguments to the *eval*() expressions, it is very difficult to determine runtime actions of the scripts [21, 25]. On the other hand, static analysis may not work if obfuscation techniques are used by attackers.

**Dynamic Analysis.** Consequently, recent efforts have been employed using runtime monitoring and tracking as these techniques can avoid the static analysis pitfalls [8, 13, 21]. Several methods have been proposed using runtime monitoring, including tainting XPCOM calls, and monitoring sensitive APIs and resources.

Dhawan *et al.* [8] implement a system called Sabre to monitor the JavaScript execution. They enumerate all the sensitive resources and low-sensitivity sinks.

Sabre associates one label with each JavaScript object in the browser and extension. Objects that contain sensitive data will be labeled differently with those containing low-sensitive data. The system will propagate labels as objects are executed and modified by extensions. An alert will be raised if an object containing sensitive data is accessed in an untrusted way or by a suspectable object.

Ter Louw *et al.* [21] implement a new tool called BROWERSPY to monitor XPCOM calls so that every time an extension accessing XPCOM is monitored and controlled by policies defined in the execution monitor. However, the overhead caused by the runtime monitoring sometimes can become a headache to the browser. In addition, XPCOM level monitoring is too restrictive and can disable some useful and normal XPCOM calls [8].

## 9 Conclusion

We propose a new approach of aspect-level behavior clustering in detecting suspicious extensions. We use SCDGs and AEBs derived from system level tracking to represent behavior characteristics of extensions. We then create profiles for both extensions and categories in the use of identifying suspicious extensions and raising alerts. We evaluate our system atop a real-world web browser with a large set of extensions including malicious ones. The experimental results show the effectiveness and efficiency of our system in detecting suspicious extensions.

**Acknowledgments.** This work was supported by ARO W911NF-09-1-0525 (MURI), NSF CNS-0905131, NSF CNS-1223710, and ARO MURI project "Adversarial and Uncertain Reasoning for Adaptive Cyber Defense: Building the Scientific Foundation".

## References

1. A. Melinte. Monitoring function calls, June 2008. `http://linuxgazette.net/151/melinte.html`.
2. S. Bandhakavi, S. King, P. Madhusudan, and M. Winslett. Vex: Vetting browser extensions for security vulnerabilities. In *USENIX Security Symposium*, pages 339–354, 2010.
3. A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *NDSS*, 2010.
4. U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krügel, and E. Kirda. Scalable, behavior-based malware clustering. In *NDSS*, 2009.
5. P. Beaucamps and D. Reynaud. Malicious Firefox extensions. In *SSTIC '08 Symposium sur la sécurité des technologies de l'information et des communications*, Rennes, France, June 2008.
6. C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In *ICSE*, pages 1066–1071, 2011.
7. M. Couture, R. Charpentier, M. Dagenais, and A. Hamou-Lhadj. Self-defence of information systems in cyber-space – A critical overview. In *NATO IST-091 Symposium*, Apr. 2010.
8. M. Dhawan and V. Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *Proceedings of the 25th ACSAC*, pages 382–391, Hawaii, USA, December 2009.

9. W. Fadel. Techniques for the abstraction of system call traces to facilitate the understanding of the behavioural aspects of the Linux kernel. Master's thesis, Concordia University, Nov 2010.
10. P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *15th Workshop on Graph-based Representations in Pattern Recognition*, pages 188–199, 2001.
11. Google Code. straceplus. `http://code.google.com/p/strace-plus/`.
12. A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *IEEE SOSP*, pages 115–130, 2011.
13. O. Hallaraker and G. Vigna. Detecting malicious JavaScript code in Mozilla. In *ICECCS*, pages 85–94, 2005.
14. J. Seward and N. Nethercote and T. Hughes. Valgrind documentation, August 2012. `http://valgrind.org/docs/manual/index.html`.
15. J. Weidendorfer. Kcachegrind, September 2005. `http://kcachegrind.sourceforge.net/cgi-bin/show.cgi/KcacheGrindIndex`.
16. G. Jacob, R. Hund, C. Kruegel, and T. Holz. Jackstraws: Picking command and control connections from bot traffic. In *USENIX Security Symposium*, 2011.
17. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.
18. C. Kolbitsch, P. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *USENIX Security Symposium*, pages 351–366, 2009.
19. Linux Man Page. strace. `http://linux.die.net/man/1/strace`.
20. M. Louw, J. Lim, and V. Venkatakrishnan. Extensible web browser security. In *DIMVA*, pages 1–19, 2007.
21. M. Louw, J. Lim, and V. Venkatakrishnan. Enhancing web browser security against malware extensions. *Journal in Computer Virology*, 4(3):179–195, 2008.
22. McAfee Labs. FormSpy. `http://www.mcafee.com/threat-intelligence/malware/default.aspx?id=140256`.
23. Mozilla. How many Firefox users have add-ons installed? 85%. `http://blog.mozilla.com/addons/2011/06/21/firefox-4-add-on-users/`.
24. Mozilla Developer Network. Downloading JSON and JavaScript in extensions. `https://developer.mozilla.org/en/Downloading_JSON_and_JavaScript_in_extensions`.
25. Mozilla Developer Network. Eval, Jun 2011. `https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/eval`.
26. N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100, 2007.
27. NetworkX. Advanced interface to VF2 algorithm. `http://networkx.lanl.gov/preview/reference/algorithms.isomorphism.html`.
28. P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *IEEE SOSP*, pages 513–528, 2010.
29. P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, and G. Vigna. Cross Site Scripting prevention with dynamic data tainting and static analysis. In *NDSS*, 2007.
30. X. Wang, Y. C. Jhi, S. Zhu, and P. Liu. Behavior based software theft detection. In *Proceedings of the 16th ACM CCS*, New York, NY, USA, 2009.
31. X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Detecting software theft via system call based birthmarks. In *Proceedings of the 2009 ACSAC*, pages 149–158, Washington, DC, USA, 2009. IEEE Computer Society.
32. Z. Xin, H. Chen, X. Wang, P. Liu, S. Zhu, B. Mao, and L. Xie. Replacement attacks: automatically evading behavior-based software birthmark. *Int. J. Inf. Sec.*, 11(5):293–304, 2012.
33. R. G. Xu. *Symbolic Execution Algorithms for Test Generation.* PhD thesis, University of California-Los Angeles, 2009.