# Private Browsing Mode Not Really That Private: Dealing with Privacy Breaches Caused by Browser Extensions

Bin Zhao, Peng Liu
College of Information Sciences and Technology
The Pennsylvania State University
University Park, Pennsylvania 16802
Email: {biz5027, pliu}@ist.psu.edu

*Abstract*—**Private Browsing Mode (PBM) is widely supported by all major commodity web browsers. However, browser extensions can greatly undermine PBM. In this paper, we propose an approach to comprehensively identify and stop privacy breaches under PBM caused by browser extensions. Our approach is primarily based on run-time behavior tracking. We combine dynamic analysis and symbolic execution to represent extensions' behavior to identify privacy breaches in PBM caused by extensions. Our analysis shows that many extensions have not fulfilled PBM's guidelines on handling private browsing data. To the best of our knowledge, our approach also provides the first work to stop privacy breaches through instrumentation. We implemented a prototype SoPB on top of Firefox and evaluated it with 1,912 extensions. The results show that our approach can effectively identify and stop privacy breaches under PBM caused by extensions, with almost negligible performance impact.**

## I. Introduction

Modern commodity web browsers provide a customizable environment to run web apps, access personal information, and manage login credentials, etc. Historically, web browsers store information such as form entries, passwords (if authorized by the user), cookies, web cache, etc. The stored information is always a privacy concern. Recently, web browsers have provided a feature denoted "*Private Browsing Mode*" (PBM) to partially address this concern. Mozilla Firefox, Microsoft IE, and Google Chrome all supports this mode, though they denote it in different terms, i.e., "Private Browsing", "InPrivate Browsing" and "incognito mode", respectively. Under PBM, web browsers do not store some private browsing data in the disk, including but not limited to browsing history, cookies, cache web, form entries, and passwords [25]. Essentially, PBM has two goals. First, no history should be stored on a user's computer during PBM [2]. For example, a family member should not know the sites one visited under PBM using the desktop shared by his/her family. Second, hide one's identity during PBM. This could have several benefits. For example, it will be more difficult for a remote attacker to steal the user's personal credentials thus reducing the security threat. Another surprising benefit is that users could save more money under PBM. Many web sites learn a user's interests by storing one's browsing session information. For example, through a user's search history, an airline may rise up the price next time he/she searches it to persuade him/her to make the purchase now. While under PBM, one might see a surprising lower price! Hence, it is reported that a significant number of people (20%) use PBM [12].

However, there is a major privacy issue with PBM. As the most popular customization, browser extensions pose a great threat to PBM. It is reported that 85% of Firefox users have installed at least one extension, "with more than 2.5 billion downloads and 580 million extensions in use every day in Firefox 4 alone" [22]. Nonetheless, all major browsers do not control how extensions handle personal data, nor do they provide sufficient support or mandatory development kit for extensions under PBM. When the PBM window is activated, the privacy protection is weak in the sense that browser extensions may do something inappropriately. First, some private browsing data generated under PBM are not properly handled. Second, extension related data under PBM are ignored by the browser. We will discuss details on the privacy breaches caused by extensions later in Section II.

**Prior Works.** Prior works primarily focus on how to identify violations of PBM. There are just few studies on it [2], [19]. G. Aggarwal *et al.* proposed a tool ExtensionBlocker to let users run extensions safely during PBM [2]. However, this tool simply disables all unsafe extensions from running. B. S. Lernera *et al.* proposed a static type system to verify extensions' compliance with PBM. This static analysis unfortunately leaves some security holes that need to be filled. It cannot ensure that all privacy breaches are identified and let alone stop privacy breaches of browser extensions.

**Key Insights and Our Approach.** Motivated by the limitations of existing works dealing with browser extensions, we propose an approach to *identify and stop privacy breaches under PBM for browser extensions*. Our approach is based on the following three insights. (1), Commodity browsers provide PBM guidelines for developers. The usual practice is that browser programmers follow these PBM guidelines. But many if not most extension developers forget or do not care that much about the PBM programming guidelines (e.g., using PBM flag) [19]. This is the primary reason for privacy breaches caused by extensions even when they are running in PBM mode. (2), A privacy breach is technically caused by two root causes: one is privacy-harming disk operations; second is forgetting to wipe out the in-memory private data. (3), The two root causes can be identified through a combination of dynamic analysis and symbolic execution. In particular, *System*

*Call Dependence Graphs* (SCDGs) can be used to identify the two root causes. But there are several challenges, such as input space issue and system call traces differentiation between browser and extensions.

Based on these insights, we proposed an approach to identify and stop privacy breaches caused by browser extensions. In a nutshell, our approach has two steps. Step 1, we obtain extensions' behavior via a combination of system level tracking and symbolic execution. This step will result in some *privacy breach patterns* for any extension that causes private data leakage. Step 2, the extensions will then be instrumented to stop privacy leakage using the identified patterns. In step 1, SCDGs are generated as a representation of behaviors for extensions. SCDGs can clearly describe disk operations and in-memory private data activities. State transition diagrams are used to identify potential privacy breaches. In a combination of state transition diagrams and SCDGs, we can generate the privacy breach patterns caused by some extensions, which are essentially sub-graphs of SCDGs. These privacy breach patterns will then be employed in step 2 to instrument those extensions so as to stop the corresponding privacy leakage. Two alternatives are introduced for extension instrumentation: disabling storing private browsing data under PBM and/or clearing temporarily-stored private browsing data when the last PBM window is to be closed. After instrumentation, extensions can then be installed and used by users.

**Main Use Cases of Our Approach.** Our approach can be used in two ways. First, extension repositories can use it to do safety check of uncertified extensions submitted by third-party developers. Second, a trustworthy web portal can be set up to allow users to upload and check the safety of any extensions.

**Contributions.** Our approach is not to replace the current private browsing module used by web browsers. Instead, we aim to complement the module and enhance the privacy protection against problematic browser extensions. Overall, this work makes the following contributions:

- We identify two root causes of privacy breaches caused by extensions: privacy-harming disk operations and forgetting to wipe out the in-memory private data.
- We systematically employ system level behavior tracking on extensions. Their behavior are dynamically represented by SCDGs, through which we can identify privacy breach patterns caused by extensions.
- This is the first attempt to stop privacy breaches caused by extensions under PBM through instrumentation. Two alternatives are proposed to instrument extensions which have caused privacy breaches.
- We implemented a prototype called SoPB. We evaluated our prototype on top of the Firefox browser with 1,912 extensions. The experimental results show that we can effectively identify and stop privacy breaches caused by browser extensions, with no false negatives and almost negligible performance impact.

The rest of the paper is organized as follows. Section II presents the issues with extensions under PBM. In Section III, we propose our approach to identify privacy breaches in PBM caused by extensions. We briefly introduce the implementation details in Section IV, followed by a comprehensive evaluation of SoPB in Section V. We then discuss the scalability and some limitations of our approach in Section VI. Finally, we summarize the related work and draw a conclusion in Section VII and Section VIII, respectively.

TABLE I: How private browsing data handled by PBM?

| Information | Internet Explorer | Chrome | Firefox | Safari |
|---|---|---|---|---|
| Extension related data | Disabled by default, can be enabled individually | | Enabled by default | |
| Cookies | Kept in memory in PBM, cleared when exits | | | |
| Temp Internet files | Stored on disk/memory in PBM, deleted when exits | | | |
| Webpage history | Not stored | | | |
| Passwords | Not stored | | | |
| Download List entry | Kept in memory in PBM, cleared when exits | | | |
| DOM storage | Kept in disk in PBM, cleared when exits | | | |
| Form/search bar entry | Not stored | | | |

## II. Issues with Browser Extensions under PBM

### A. Private Browsing Data

Before we discuss the possible privacy breaches of PBM caused by browser extensions, let us first define what is private browsing data. Based on the PBM guidelines [23], the meaning of "private browsing" is informally defined through some data not being stored on local disk or any local storage. Based on this, we define *Private Browsing Data* as follows. During a PBM browsing session, certain private data (Family I) is not supposed to be generated. In addition, although the other private data (Family II) can be generated during a PBM session, such data should not be put onto the local disk. Putting Family I and Family II together, we get the definition of *Private Browsing Data* [31], [25], [8], [3]. Table I lists the pre-specified *private browsing data* types [31], [25], [8], [3]. For example, cookies and browsing history are considered as private browsing data. However, as stated in [23], [25], a file downloaded from a remote server or a bookmark added during a PBM session are not defined as private browsing data.

### B. Privacy Breaches of PBM Caused by Browser Extensions

Since developers of the browser code will follow the PBM guidelines, we focus on the private browsing data accessed by the extensions (typically extensions conduct read and write operations on these data; the data are typically not generated by extensions). In the current PBM system model, the PBM window and the public window execute the same piece of code. A particular value returned by a global function tells whether the current window is PBM or normal session. Let us take Firefox as an example.

A module `PrivateBrowsingUtils.jsm` is imported to handle private browsing data under PBM. The function `isWindowPrivate(window)` (or `isContentWindowPrivate()` in latest Firefox versions) in this module is used to determine if a given DOM window is private [24]. The following sample code shows how a developer can use this function. The return value of the function `isWindowPrivate(arg)` is then used to separate private mode from public mode [24].

```
if (PrivateBrowsingUtils.isWindowPrivate(window))
     { ... }
else { ... }
```

**PBM Guidelines.** Overall, the browser code is carefully implemented to support PBM. A basic **PBM guideline** is quoted as follows: "*It is not acceptable for an extension that records things like URLs or domains visited to even offer the option to opt out of private browsing mode*" [23]. This can be elaborated into two primary PBM guidelines. **Guideline I**: Disk operations involving private data should be strictly controlled. In addition, the generation of certain private data should also be strictly controlled. **Guideline II**: Extensions should check whether the current window is under PBM (flag checking) whenever private browsing data is accessed. However, neither of these two guidelines are fulfilled by many existing extensions.

**Privacy breaches we are and are not dealing with.** There are two primary types of privacy breaches associated with extensions, (I) putting private browsing data onto local disk/storage, and (II) transferring private browsing data to a remote computer. Based on the two PBM guidelines, a privacy breach concerned by PBM is actually type I only. PBM guidelines do not explicitly require any protection of type II. PBM is specifically designed to disable the storing of some private browsing data "on the local computing device" [8]. Therefore, in this paper we only focus on type I. Type II is out of the scope of this work. Hence, we define a privacy breach as follows.

**Definition 1.** *Privacy Breach. Let $p$ be a running extension. Let $D$ be the private browsing data accessed by $p$ during a PBM session. If $\exists d \in D$ do not fulfill Guideline I and Guideline II, we say that there is a Privacy Breach caused by this extension $p$.*

For example, when browsing inside a PBM window, the browser code will disable putting any cookie on the disk. Even if the window is closed, the cookie will not be put onto the disk. During the whole PBM session, the disk is not supposed to hold any cookie data. It is possible that in-memory cookie data could be sent by a malicious extension to a remote server; however, this is not a concern of PBM. Any unauthorized transfer or retrieval of data from a computer or server over the Internet or other network is regarded as data exfiltration, not PBM related privacy leakage [6].

A primary reason that extensions cause privacy breaches is that "PBM does not magically handle what your extension does in saving browsing history data; that is the job of each extension" [11]. All major browsers do not control how extensions handle private browsing data. From Table I, we can see that most private browsing data are properly handled (deleted or not stored) during PBM. However, extension related data are not properly dealt with whether extension are disabled by default or not. There are two major ways for an extension to violate PBM guidelines. First, extensions do many disk harming operations, e.g. storing cookies and passwords, generating new files, etc. Second, extensions can maintain their own profiles (store data and files an extension needs to run), which are not properly handled when the PBM session is ended.

Another reason is that extension developers do not follow the PBM guidelines. Some extension developers even do not know changes are needed to comply with PBM guidelines [19]. Even if they know, do they know how to make changes to extensions? An Add-on SDK maintained by Jetpack Project for Firefox extension developers does offer some PBM related APIs [26]. However, as shown in our evaluation (Section V), only a small percentage of extensions use this SDK. Besides, the results also show that even the SDK cannot ensure full compliance with PBM, let alone without the SDK.

## III. Our Approach to Deal with Privacy Breaches Caused by Extensions

Prior static analysis approaches leave some security holes when checking whether PBM guidelines are fulfilled. Inspired by this, can a combination of dynamic analysis and symbolic execution fill these security holes? This can be elaborated as follows. (1) Given an extension, can we find out the potential privacy breaches of this extension? (2) If there is a privacy breach caused by this extension, can we instrument this extension based on what we learned from the analysis to stop the privacy breaches?

### *A. Behavior Representation of Browser Extensions*

As aforementioned, an extension's behavior can be represented by using a particular graph called SCDG, specifically, 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 describes the interaction between a running program and the operating system. This interaction is an essential behavior characteristic of the program [30]. In this paper, SCDG is defined as follows [30], [33].

**Definition 2.** *System Call Dependence Graph. Let $p$ be a running extension. Let $I$ be the input to $p$. $f(p, I)$ is the generated 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, $n \in N$ is a system call*
- *$E$ is a set of data dependence edges, $E \subseteq N \times N$*
- *$F$ 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 a function assigning attributes to node value*

### *B. Why Use System Calls and SCDGs?*

We use lower-level system calls as a representation of browser extensions's behavior for three primary reasons. First, system calls are the only interface between the operating system and a running program, providing the only way for a program to access the OS services [30]. Second, system calls can clearly show the interactions with filesystem and disk operations. Third, private browsing data are closely related with specific files and folders. History, bookmarks, etc., are all stored in specific files. When tracing private browsing data, it is easier to track from source.

However, from a single system call trace, we know little information about the overall behavior of an extension, as system calls are low level reflection about the behavior characteristics of a program. How can we map the low level system call traces with application level behavior? We need an intermediate representation to connect them together. This is one of the primary reasons for using SCDGs, as SCDGs can appropriately reflect the dependencies between system calls. They are the abstraction of a sequential system call traces. They can clearly describe the interactions among all the private browsing data and disk operations.

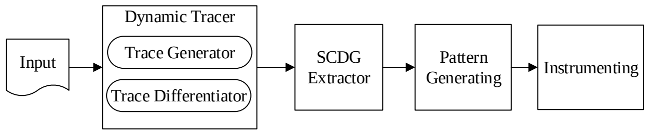

Fig. 1: Overview of the approach, including 4 components.

## *C. Approach Overview*

Our approach consists of four components: Dynamic Tracer, SCDG Extractor, Pattern Generating and Instrumenting, as shown in Fig. 1.

**Dynamic Tracer.** The dynamic tracer is mainly composed of a trace generator and a trace differentiator. The dynamic tracer tracks the behaviors of extensions in the form of system calls, using a symbolic execution based input resolver to address the input space issue. The trace differentiator is a component distinguishing 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 aims to generate SCDGs for each extension. It first explores the data dependencies between system calls. Then, it identifies objects (e.g., cookies, cache, etc.) and encodes them for the use of the following component.

**Pattern Generating.** This component is to generate privacy breach patterns caused by extensions based on SCDGs and the definition of privacy breaches. We narrow down the files/folders that are constrained with browser extensions and potential private data leakage. A state transition diagram is created for each file/folder to identify privacy breaches. Privacy breach patterns are generated from a combination of state transition diagram and SCDGs.

**Instrumenting.** If there is a privacy breach caused by an extension, this extension will be instrumented to ensure the privacy protection. A primary reason that a privacy breach occurs is that extension developers forget or even ignore to check whether the current window is under PBM. In fact, whenever private browsing data is accessed, the extension should be coded to determine if a given DOM window is private. Instrumenting is done in two steps. Step 1, evaluating the private browsing data causing privacy breaches by using the private browsing module. Step 2, based on the generated privacy breach patterns, instrument the code to properly handle the private browsing data.

**Challenges.** Our approach 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 narrow down and define all the privacy breaches caused by extensions. We define this based on privacy-harming disk operations and in-memory private data.

## *D. System Level Behavior Tracking*

System level behavior tracking is done by the dynamic tracer. The dynamic tracer takes the browser and extensions as the input, and eventually generates the trimmed system call traces for each extension. Two primary components are contained in the dynamic tracer: trace generator and trace differentiator. There are two key challenges when perform dynamic tracing.

**Input Space Issue.** Input space issue, also known as execution path issue, is to trigger or elicit a program's behavior as much as possible. This is a key challenge when employing dynamic tracing on extensions. An input used by a program (value and event, e.g. data read from disk, a network packet, keyboard input, etc.) cannot always be guaranteed to reoccur during a re-execution [33]. As a result, an extension may result in a set of execution paths with different inputs, while these execution paths may be different. It is very likely that certain 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 these actions cannot be triggered, reducing the coverage of an extension's behavior. As a result, this may reduce the extracted SCDGs and finally cause some false negatives in the following privacy breach pattern generating.

There, we need to automatically explore the input space for 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 [29]. 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 filled by a user, text areas, and URLs [33].

We proposed an input resolver (IR) based on dynamic symbolic execution in our paper. The IR is used to "hit" as many inputs and finally execution paths as possible for an extension. The IR tracks the symbolic variables instead of the actual values. Values of other variables depending 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 [29], [33].

Let us 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 running 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 $\{\xi_i\}$ [33]. There is a counter *pc* accumulating constraints on the inputs that trigger the execution to follow the associated path. For a conditional expression `if (e)` $\texttt{S}_1$ `else` $\texttt{S}_2$, *pc* is updated with assumptions on the inputs to choose between alternative paths [7], [32]. 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 $pc \wedge \mu(e) \neq 0$, where $\mu(e)$ denotes the symbolic predicate obtained by evaluating *e* in symbolic state $\mu$. Hence, both branches can be taken under symbolic state, resulting in two different execution paths. When *pc* is not satisfied, symbolic execution is terminated. The satisfiability is checked with a constraint solver. For each execution path, every satisfying assignment to *pc* gives values to the input variables that ensure the specific execution proceeds along this path. If there are loops or recursion, a limit is given on the iteration, i.e., a timeout or a limit on the number of paths [7], [17], [32].

The IR works as follow. The IR includes a dynamic symbolic interpreter performing symbolic execution of JavaScript, a path constraint extractor building queries based on the results of symbolic execution, a constraint solver finding satisfying assignments to those queries, and an input feedback component using the results from the constraint solver as new program inputs [29], [33].

There is a unique challenge for extensions which is the event space issue. To detect all events resulting in JavaScript code execution, we propose the following approach. First, a GUI explorer searches the space of all events using a random exploration strategy. Second, browser functions are instrumented to process HTML elements so as to record the time of the creation and destroying of an event handler [29], [33]. Ordering of user events registered by the web page is randomly selected and automatically executed. Random seed is used to replay the same ordering of events. The GUI explorer can also generate random test strings to fill text fields when handlers are invoked [29].

**Differentiating System Calls between an Extension and Browser.** Differentiating of system call traces between the browser and extensions poses a great challenge to our approach. Different browsers have adopted various extension system mechanisms. For Chrome, each extension maintains its own process, which is also separated from Chrome browser process. Hence, the differentiating problem does not exist for Chrome. However, 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: How can we differentiate system call traces between the browser and extensions? Second, how can we differentiate system call traces among various extensions?

A fine-grained system call tracing approach is introduced to address this challenge. Let us first explore how extensions interact with the Firefox browser. When an extension is running, the extension and browser JavaScript are first interpreted by JavaScript Engine. Then they are connected to XPCOM through XPConnect. An key point in this process is that extension JavaScript can access to the resources through Firefox APIs. Therefore, one possible way of locating the real caller of a system call is to track or intercept the functions. Several previous approaches have been proposed to track those API functions [4], [5]. From these functions, we can obtain cues about when a function is entered and exited, and where the function is called from. Based on this runtime call tree, we can differentiate the system calls between web browser and extensions.

During the implementation, `Callgrind` is used to implement this approach, which is based on `Valgrind` [14], [15]. `Callgrind` uses runtime instrumentation via the Valgrind framework for its cache simulation and call-graph generation [27]. Caller/callee relationships between functions are collected by `Callgrind`. A subroutine is mapped to the component library which the subroutine belongs to [33]. 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 [30]. Therefore, it can dynamically build a call graph generated by the web browser and extensions. We added a *timestamp* for each system call to increase the accuracy of system call differentiation. The timestamp can help quickly locate the system call traces of extensions and remove unnecessary system call traces.

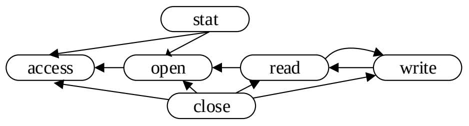

Fig. 2: Example dependencies among syscalls of file operations.

However, how can differentiate system call traces among various extensions? To remove the interference, we run just one extension while disabling all other irrelevant installed extensions. This definitely reduces the possibility of parallel processing. However, three reasons support this practice. First, each system call trace occupies very little time. Running one extension exclusively will not reduce much of the speed in our approach. Second, it is not necessary to do parallel processing for extensions, as usually the running extensions do not affect each other's behavior. Third, this practice will greatly improve the accuracy of the system call trace differentiating, reducing possible false positives and negatives.

**Noise Filtering Rules.** System call traces usually contain a significant amount of noise that can clutter the trace and influence analysis of system call dependencies. Removing the noise from traces can improve the quality of the following SCDG extraction. In this paper, we employed three basic filtering rules. Filtering rule 1, system calls that do not represent the behavior characteristics we want are ignored, e.g., system calls related to page faults and hardware interrupts [9], [30]. Filtering rule 2, system calls with very similar functionality are considered the same. For example, *fstat(int fd, struct stat *sb)* system call is regarded as the same with *stat(const char *path, struct stat *sb)*. Filtering rule 3, we ignore failed system calls, as they do not affect the dependencies and furthermore the SCDGs[30].

## *E. SCDG Extracting*

A SCDG is essentially determined nodes which are system calls and edges which are dependencies. In this section, we primarily focus on how to derive dependencies between system calls and how to do object encoding on nodes.

**Dependencies between System Calls.** A system call trace consists of the system call name, some arguments, a return value and time, etc. Usually arguments of a system call are dependent on previous system calls. For example, the file descriptor argument `fd` in one system call is usually derived from previous system call. There are two types of data dependencies 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], [33]. Fig. 2 shows an example of some dependencies among system calls of file management [13]. 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 where $f$ is a function to derive dependencies between system calls. Specifically, for an argument $a_i$, $f_{a_i}$ is defined as $f_{a_i} : x_1, x_2, ..., x_n \to y$, 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 to system calls.

**Objects Identifying and Encoding.** Here we formalize node derivation function $\beta$ in the definition of SCDG for each system call. A primary challenge for $\beta$ 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 several categories. First, files and folders related to browser extensions, as shown in Table II, which we will discuss specifically in the following subsection. Second, files and folders related to the host browser. Most of them are stored under the browser profile folder. Third, system and user libraries related to browser and extensions in operating system. Fourth, local sockets for browser and extensions. We represent each file/folder using a natural number. For example, `localstore.rdf` can be encoded with 1.4 meaning this file is coded in the fourth position of the first category.

We identify the objects and assign each node with an object code primarily for two 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 shows several SCDGs from the extension Foxtab. For simplicity, we just include a system call name and a sequence number for each node, while excluding the code for each node. These SCDGs are extracted when starting Foxtab extension. Usually, each extension can get a large amount of system call traces leading to many SCDGs. We show the statistics in the evaluation section.

## *F. Generating Privacy Breach Pattern*

Based on the extracted SCDGs, to identify if there is a privacy breach caused by extensions, we generate the privacy breach patterns from those SCDGs. Before generating the patterns, we first narrow down and define all the privacy breaches caused by extensions. Then we explore the SCDGs to identify if they correspond to a privacy breach.

Privacy breaches caused by browser extensions are constrained with private data leakage. Privacy-harming disk operations are essentially related to operations on files and folders. Hence, there are two things we need to clarify here. What files and folders are privacy concerns? What operations done to those files/folders can be privacy harming?

**Files/folders Related to Extensions.** In this paper, we classify those files and folders into three categories. (C1) Files and folders *directly* related to extensions. “Directly” means that extensions usually need these files and folders to install, remove, start or run normally. Another important feature of C1 is that they are not handled by the existing Private Browsing Module. Table II describes files and folders in C1, taking Firefox as the platform. Most of these files and folders are specifically designed for Firefox extensions, without which extensions cannot run normally. Many extensions maintain their own profile folders under Firefox profile folder. These folders usually save some useful data required by extensions' functionality, such as browsing history, and screenshots, etc.

TABLE II: Files/folders directly relate to Firefox extensions

| File/Folder Name | Description |
|---|---|
| Firefox profile folder | Folder for most data |
| extensions | Folder for all installed extensions code |
| extension profile folders | Folders for corresponding installed extensions |
| addons.json | Stores AddonRepository data |
| addons.sqlite | Database storing AMO data for installed add-ons, e.g. screenshots, ratings, homepage, etc. |
| extensions.ini | Lists folders of installed extensions and themes |
| extensions.json | Stores XPIProvider data for installed extensions |
| extensions.sqlite | Installed extension information |
| localstore.rdf | Toolbar and window size/position settings |
| permissions.sqlite | Permission database for cookies, pop-up blocking, image loading and add-ons installation |

(C2) Files and folders *indirectly* relate to Firefox extensions. “Indirectly” means that these files and folders are not necessary for all extensions, but some extensions may need them based on their functionality and implementation. This category includes `cookies.sqlite` (stores Cookies), `formhistory.sqlite` (stores form data), `places.sqlite` (stores bookmarks, browsing history, favorite icons, etc.), `pref.js` (stores all preferences), and `key3.db/signons.sqlite` (key database/encrypted saved passwords), etc. A second feature of C2 is that most of them will be handled by the existing Private Browsing Module during PBM as shown in Fig. 1, except `places.sqlite`. Hence, `places.sqlite` in C2 needs to be monitored.

(C3) Files/folders primarily relate to Firefox and do not impact extensions' running. It includes `bookmarkbackups`, `minidumps`, `cert8.db`, `compatibility.ini`, and `healthreport.sqilte`. Files and folders in C3 are not necessary to monitor.

**Privacy-harming Operations and In-memory Private Data.** After narrowing down private browsing data, a *State Transition Diagram* is created for each file/folder to determine whether operations on this file/folder could be privacy harming. Fig. 4 describes the State Transition Diagram for files/folders. There are three states for the files/folders, an initial state, an active state, and a final state. Active state represents that the monitored file/folder is present. Final state represents that the monitored file/folder is dead or disappears from the monitored directory. Second, we explore the operations and transitions. We select 7 significant operations that can trigger a state transition and can lead to privacy concern. When a file/folder is moved to a directory or renamed, events `IN_MOVE` or `IN_MOVE_SELF` may happen and trigger a transition to the state active. When a file/folder is created, events `IN_CREATE` can also trigger the transition to the state “Active”. Self-transitions for state active can be triggered by events `IN_MODIFY` when modifying a file or `IN_CLOSE` when close a file (not) for writing. The final state can be triggered by events `IN_DELETE` or `IN_DELETE_SELF` when deleting a file/folder.

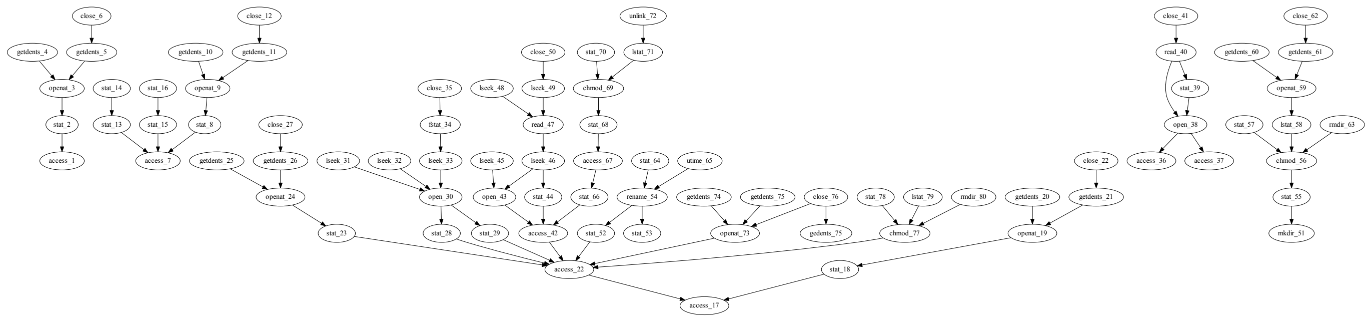

Fig. 3: Some SCDGs extracted from the extension FoxTab, showing the dependence graph of the system calls. Each node consists of two parameters, system call name and the sequence for this system call.

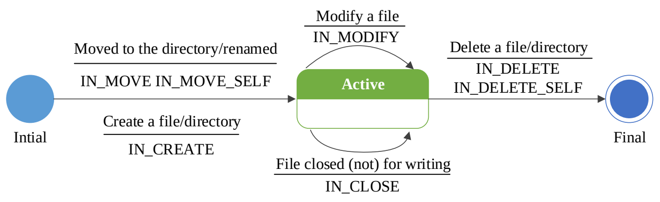

Fig. 4: State Transition Diagram for files/folders

How can we identify privacy breaches based on State Transition Diagrams? If both the two prerequisites are satisfied, we define it as a privacy breach for disk operations: (1) *There is a transition from the initial state to the active state triggered by events `IN_CREATE`, `IN_MOVE` or `IN_MOVE_SELF`*; and (2) *the file/folder is still in state "active" when PBM exits*.

**Privacy Breach Pattern Generating.** A privacy breach pattern is essentially a sub-graph of SCDG that corresponds to the previously defined privacy breach. To find out the pattern, we need to map the defined privacy breaches into SCDGs. This can be done in two steps. Step 1 is to map the operations to SCDGs. In SCDGs, operations are represented as the arguments in system call traces. Step 2 is to map the files/folders into SCDGs, which is represented as return value and arguments (e.g. fd) in SCDG node. The state transitions are mapped to the dependencies in SCDGs. If a defined privacy breach is located in one SCDG, we say this (sub)SCDG is a privacy breach pattern of this extension.

Each time when we get one privacy breach pattern, we use *subgraph isomorphism* to compare SCDGs of other extensions to check if this pattern appears in other extensions. Subgraph isomorphism is defined as follows in our paper [30], [33].

**Definition 3.** *Subgraph 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 subgraph isomorphism of $G$ and $H$ exists if and only if there is a bijection between the vertex sets of $G_1$ and $H_1$ where $G_1 \subset G$ and $H_1 \subset H$: $f : N \to N'$ such that any two vertices $u$ and $v$ of $G_1$ are adjacent in $G_1$ if and only if $f(u)$ and $f(v)$ are adjacent in $H_1$, 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,*
- $\forall e' = (u', v') \in E', \exists e = (f^{-1}(u'), f^{-1}(v')) \in E$

Subgraph isomorphism is employed in generating privacy breach patterns for the following reasons. First, a privacy breach pattern is actually a (sub)SCDG. Using subgraph isomorphism can greatly increase the time performance when generating privacy breach patterns. Second, using subgraph isomorphism can know what privacy breach patterns are most popular. Third, subgraph isomorphism can also help reduce the workload when doing instrumentation, as similar privacy breach patterns usually use the same instrumentation methods.

## G. Extension Instrumentation

Extension instrumenting is to instrument any extension that has caused privacy breaches in our dynamic analysis. Before instrumenting extensions, we first explore the essential reason that cause the privacy breaches. As we mentioned in the introduction, a primary reason that a privacy breach occurs is that extension developers forget or even ignore to check whether the current window is under PBM. However, whenever private browsing data is accessed, the extension should be coded to determine if a given DOM window is private. Therefore, given that we already know what privacy breaches occur on extensions, instrumentation is done in several steps. Step 1, evaluating the privacy-harming disk operations and in-memory private data that causing privacy breaches by using the private browsing module. Step 2, based on the generated privacy breach patterns, instrument the code to properly handle the private browsing data. The private browsing data can be dealt with in two ways. First, directly disable storing private browsing data in PBM by default. Second, as some private browsing data is permitted to be stored during a PBM session, clear these temporarily-stored private browsing data when the last PBM window is to be closed. Step 3, add code to ensure that private browsing data is properly handled under PBM session. Step 4, zip the code to the right format and the code instrumentation is completed.

**Disabling storing private browsing data in PBM.** A basic idea is that when the extension accesses the private browsing data, such as browsing history and cache, it needs to first check if the current session is under PBM.

**Clearing any temporarily-stored private data.** "As it is permissable to store private browsing data in non-persistent ways for the duration of a private browsing session" [24]. To be notified when such a session ends (i.e., when the last PBM window is closed), observe the `last-pb-context-exited` notification. After receiving this notification, the instrumented code can then clear the specified temporarily-stored private data like browsing station history and cookies.

A case study on instrumentation is conducted in Section V-I showing some details in implementing these two methods.

## IV. Implementation

We implemented a prototype called SoPB (Shepherd of Private Browsing) for our approach. Corresponding to the approach architecture, the prototype is divided into Dynamic Tracer, SCDG Extractor and Pattern Generating and Extension Instrumentation. Overall, the core code are approximately 2,040 lines (C++), with another about 1,100 lines of code implementing the UI in JavaScript.

The trace generator is implemented based on `strace` [20]. It can track the system calls and filter off the unnecessary system calls. Our input resolver is primarily based on `Kudzu` [29]. We modified it to employ it on the web browser and generate inputs for the trace generator. 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 [30].

A primary implementation of the pattern generating is to find out privacy breaches based on state transition diagrams. We use `inotify` to monitor filesystem and events, as identified in Section III-F. We then determine privacy breaches based on all generated state transition diagrams and the privacy breaches. The privacy breach patterns (sub-SCDGs) are generated based on a comparison among privacy breaches and SCDGs. We implemented the subgraph isomorphism based on $VF2$ algorithm of NetworkX [28].

For instrumentation, we obtain the private browsing data generated by an extension based on the privacy breach patterns. If the private browsing data is used to maintain the functionality of this extension, these data are permitted to be retained during the PBM session, but will be cleared when the last PBM window is to be closed. Otherwise, we disable the storing of private browsing data under PBM.

## V. Evaluation

We propose 8 primary evaluation goals to SoPB. (G1) Evaluation on the input space issue. (G2) Do extensions comply with PBM guidelines? (G3) How many times does an extension violate PBM guidelines? (G4) What privacy breach patterns are generated? (G5) How many patterns are shared by extensions? (G6) Can instrumentation effectively stop the privacy breaches? (G7) Performance evaluation. (G8) A case study on extension instrumentation.

### A. 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 Ubuntu 12.04.4 LTS with Linux 3.8.0-35-generic. We use Firefox 26.0 as the host browser, which uses per window PBM. We examined 1912 extensions in total based on two criteria. (1) Popularity, top 2100 extensions based on average daily users. 1903 extensions are actually chosen, while others are either incompatible with Linux or Firefox 26.0. (2) Recommendations, also known as featured extensions in AMO. 9 additional extensions are added to our extension pool contributing to the final number of 1912.

TABLE III: Comparison on Input Space with and without `IR`

| # of ext. | # of SCDGs w/o `IR` | # of SCDGs w/ `IR` | # of outliers |
|---|---|---|---|
| 87 | 3014 | 4601 | 10 |

TABLE IV: Results on privacy-harming disk operations. Note: PB means privacy breaches, PF means profile folder.

| Total # of ext. | # of ext. have PF | # of ext. cause PB in PF | # of ext. cause PB other than PF | Total # on privacy-harming disk operations |
|---|---|---|---|---|
| 1912 | 472 | 285 | 26 | 311 |

### B. Evaluation on Input Space Issue

The input space issue is evaluated by comparing SCDGs as they can reflect execution paths and the input to a large degree. Specifically, two metrics are evaluated on our IR, increase and outliers of SCDGs after employing the IR.

87 extensions are randomly chosen in this evaluation based on their categories in the extension repositories. The experimental results are shown in Table III without and with applying the IR on the browser. There is a significant 52.7% increase in the total number of SCDGs after using the IR. On the other hand, an outlier occurs if a SCDG before using the IR is not included in the set of SCDGs after using the IR. On average, there is only a very small percentage (0.3%) of previous SCDGs are outliers. Outliers are most likely caused by the different parameters of graphs. Our IR can increase the total number of SCDGs substantially, but also control the outliers in a very small range [33].

### C. Do Extensions Comply with PBM Guidelines?

Based on the PBM guidelines in Section II-B, three primary criteria are evaluated: privacy-harming disk operations, in-memory private data handling, and flag checking.

**Privacy-harming Disk Operations.** Table IV shows that many extensions create their own profile folders under the Firefox profile folder (472/1912). Our prototype recursively monitors these extension profile folders and finds that many of them violate PBM guidelines. For example, in the profile folder of Foxtab extension, there are several folders including `data`, `thumbs`, `thumbsRCT`, and `ThumbsTS`. Screenshots and browsing sites under PBM will be stored in these folders even when the last PBM window exits. This is regarded as a privacy breach. Overall, we find that 472 of 1912 tested extensions (24.7%) maintain their profile folders. 285 of them have caused privacy breaches. On the other hand, some extensions (26) create files other than in profile folders and do not remove them even when the last PBM window is closed. This is a privacy breach that apparently violates PBM guidelines. As shown in Table IV, 26 extensions have such kind of privacy breach. In total, there are 311 (16.3%) extensions have privacy-harming disk operations.

**In-memory Private Data Handling.** We mainly evaluate whether in-memory private data of extensions are removed when the last PBM window is to be closed. To check memory content, we implemented an extension to periodically obtain and update the memory cache list, including the key value of the cache and the last modified date. As described in Section III-G, we observe the `last-pb-context-exited` notification to get notified when the last PBM window is closed. Then we compare the memory cache to check if there is any private data related to extensions that are not removed. Our

TABLE V: Experimental results on the times of extensions' privacy breaches. Note: PBP means privacy breach patterns, and PB means privacy breaches.

| Total # of ext. | Avg. # of SCDGs | Total # of Uniq PBP | Avg. testing time (sec) | Avg. # of PBP | Avg. times of a PB |
|---|---|---|---|---|---|
| 32 | 48 | 28 | 155.2 | 4.2 | 6.2 |

experimental results show that extensions (actually browser) are doing better in handling in-memory private data than disk operations. There are only 9 extensions that have caused privacy breaches in this category, meaning that the in-memory private data generated by these 9 extensions under PBM are retained when the last PBM window is closed. This result is not surprising as usually memory management is done by the host browser. After an investigation of these 9 extensions,

**Flag Checking.** Only 19.0% (363/1912) of the tested extensions have checked the flag when private browsing data is accessed. Of those extensions which do flag checking, **31.9%** (116 extensions) use Jetpack Add-on SDK to handle extensions' behavior during PBM, compared with the other **68.1%** (247) extensions' PBM handling is still done by developers's own implementation. Overall, given that only less than a quarter of tested extensions do flag checking and an even smaller amount of **6.1%** use Add-on SDK, we consider that Firefox's requirements on extensions under PBM are poorly responded by extension developers. The results demonstrate that most extension developers have not followed Firefox's guidelines which requires extensions to respect PBM [1].

## D. How Many Times Does an Extension Violate PBM?

In this section, we discuss how many times an extension may violate PBM guidelines. Actually, the times of an extension's violation of PBM guidelines are dependent on two factors: the length of an extension's running time and the frequency of this extension's violation. Therefore, given a privacy breach found in an extension, can we know how many times this privacy breach happens in this extension? Overall, there are 320 extensions causing privacy breaches. We randomly chose 32 extensions (10%) to conduct this evaluation. Table V shows the experimental results on these extensions. On average, there are around 4 unique privacy breach patterns for each extension in the testing set. During the testing time (the average is 155.2 seconds), the average times of a privacy breach happening in one extension is 6.2, while the median times is 6. The number of times varies much for each privacy breach. There are 4 privacy breaches happening only once during the testing time. In comparison, the greatest times a privacy breaches happening in an extension is 14.

## E. What Privacy Breach Patterns are Generated?

Based on the privacy breaches found above, we can generate the privacy breach patterns. As mentioned in the above subsection, 311 extensions (out of 947) are found to have privacy-harming disk operations. 9 extensions forget to remove the in-memory private data when the last PBM window is to be closed. Table VI shows the statistical result for the tested extensions. For an extension that violates the PBM guidelines, on average there are 4.3 privacy breach patterns (denoted as PBP in the Table VI). Through subgraph isomorphism, 219 unique privacy breach patterns (sub-SCDGs) are generated for all the 320 extensions that have caused privacy breaches.

TABLE VI: Experimental statistics for tested extensions.

| Total # of ext. | Avg. # of SCDGs | # of ext. violate PBM guidelines | Avg. # of PBP for each ext. | Unique PBP # |
|---|---|---|---|---|
| 1912 | 46 | 320 | 4.3 | 219 |

TABLE VII: Number of extensions fall in each privacy breach category.

| # of ext. have privacy breaches | Ext. in C1 | Ext. in C2 | Ext. in C3 | Ext. in C4 | Ext. in C5 |
|---|---|---|---|---|---|
| 320 | 272 | 10 | 26 | 17 | 9 |

To simplify, we summarize these unique privacy breach patterns into the following categories.

- C1, Creates and finally stores files in the extension's profile folder,
- C2, Moves and finally stores files into the extension's profile folder,
- C3, Creates and finally stores files in other places,
- C4, Renames newly generated files/ in extension's profile folder,
- C5, Generates and finally stores the in-memory data.

Table VII shows the number of extensions that fall into each privacy breach category. All 9 extensions having privacy breaches related to in-memory data fall into C5. Most extensions causing privacy breaches belong to C1. This is reasonable as many extensions maintain a profile folder, so it is very likely that some could forget or ignore to remove the files generated under PBM. The total number in the four categories are greater than 320 because there are a few extensions that belong to more than one category.

We have all the statistics so far for the privacy breach pattern. How do the privacy breach patterns look like? Fig. 5 shows two privacy breach patterns from the Foxtab extension, which are actually two sub-SCDGs drawn from Fig. 3. The left figure is a privacy breach pattern named "trash folder creating and checking". Foxtab creates a trash folder; however it will not remove this folder when the PBM session ends. The right figure shows a privacy breach pattern named "renaming a file". Foxtab renames a file during a PBM session without removing the new renamed file when the PBM session ends.

## F. How Many Patterns Are Shared by Extensions?

In this section, we want to know how many privacy breach patterns are shared by a specific number of extensions (e.g., 2, 3, 4,...). There are two questions: which privacy breach

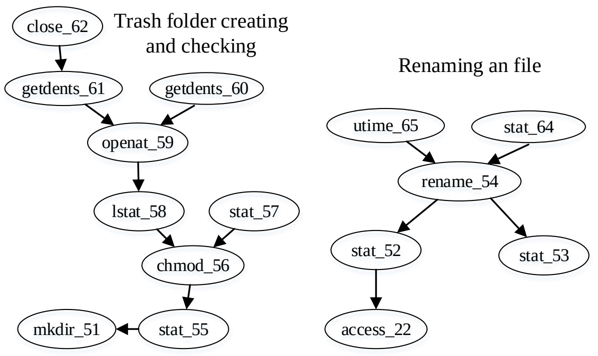

Fig. 5: Two privacy breach patterns from Foxtab.

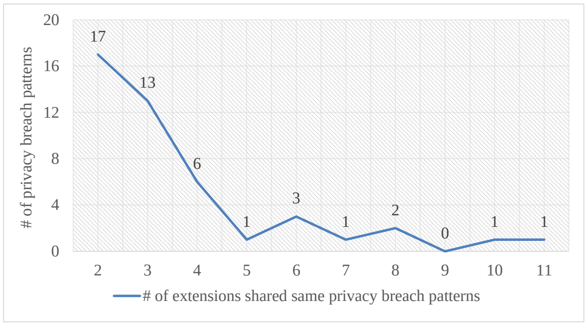

Fig. 6: Number of extensions sharing same privacy breach patterns

pattern is shared by most extensions? and which privacy breach patterns are not shared by other extensions (a.k.a, only one extension has this privacy breach pattern)? Fig. 6 shows the number of privacy breach patterns are shared by 2 and more extensions. All the remaining 174 privacy breach patterns are only found in one extension, not shared by others. In total, there are 45 privacy breach patterns are shared by 2 and more extensions. Most of the these privacy breach patterns (36 out of 45) are shared by 2, 3 and 4 extensions. The greatest number of extensions sharing the same privacy breach pattern is 11. This privacy breach pattern belongs to category C1. These results are reasonable as extensions usually will generate different SCDGs.

### *G. Does Instrumenting Effectively Stop Privacy Breaches?*

We instrumented 15 extensions (out of the 320 extensions causing privacy breaches) to evaluate the effectiveness of stopping privacy breaches. Based on Section III-G, there are two methods to instrument these 15 extensions: M1, disabling storing private browsing data in PBM, and M2, clearing temporarily-stored private browsing data when the last PBM window is to be closed. If an extension needs to temporarily store some private browsing data to maintain its functionality, M2 will be used to instrument this extension. It is also possible that an extension may store some private browsing data which are not necessary for this extension. In this case, M1 is used to instrument this extension. A few extensions can use both M1 and M2 to do the instrumentation. Table VIII shows the results for the 15 instrumented extensions. Let us take Foxtab extension as the example. Foxtab's privacy breach patterns belong to three categories: C1, C2, and C4. We use M1 to disable the storing of browsing sites. M2 is used to clear other temporarily-stored data such as screenshots.

**Effects of Instrumentation Analysis.** All these instrumented extensions are wrapped and then installed on Firefox. We then evaluate these extensions again using SoPB to check if there is still a privacy breach. Using dynamic tracer, SCDG extractor and privacy breach pattern generating, **no privacy breach** is found for these 15 instrumented extensions. This demonstrates that the instrumentation can effectively stop the privacy breaches caused by browser extensions.

### *H. Performance Evaluation*

We tend to provide a cost-effective service to enhance the privacy protection for browser extensions. Hence, we discuss the performance of SoPB in two metrics: memory usage and startup time. We made a Firefox extension for SoPB. The reason we made it an extension is that this can better reflect SoPB's impact on Firefox and their relations.

TABLE VIII: Results of 15 instrumented extensions.

| Instrumented Extensions | Version | SCDGs | PBP categories | Instrumenting method |
|---|---|---|---|---|
| Foxtab | 1.4.9 | 46 | C1,C2,C4 | M1:disable storing data, M2:clear temp data |
| Fire.pm | 1.4.15 | 39 | C1 | M1:disable storing data |
| The Camelizer | 2.4.9 | 51 | C1 | M2:clear temp data |
| DownloadHelper | 4.9.24 | 53 | C3 | M2:clear temp data |
| Youtube Downloader | 2.1.1 | 39 | C3 | M2:clear temp data |
| Firebug | 1.12.8 | 32 | C1 | M1:disable storing data |
| FlashGot | 1.5.6.8 | 48 | C1,C3 | M1:disable storing data, M2:clear temp data |
| Adblock Plus | 2.6.6 | 43 | C1,C4 | M2:clear temp data |
| DownThemAll! | 2.0.17 | 36 | C3 | M2:clear temp data |
| WOT | N/A | 31 | C1 | M2:clear temp data |
| Blur (DoNotTrackMe) | 4.5.1334 | 47 | C1 | M1:disable storing data, M2:clear temp data |
| Youtube MP3 Podcaster | 3.5.0 | 40 | C3 | M2:clear temp data |
| Super Start | 2.0.2 | 61 | C1 | M2:clear temp data |
| Ant Video Downloader | 2.4.7.26 | 55 | C1,C3 | M2:clear temp data |
| ScrapBook | 1.5.11 | 49 | C1 | M2:clear temp data |

TABLE IX: Average startup time for Firefox w/ and w/o SoPB in milliseconds

| | main | start | selectProfile | afterProfileLocked |
|---|---|---|---|---|
| Avg. time w/o SoPB | 16.20 | 3.00 | 91.27 | 98.56 |
| Avg. time w SoPB | 16.50 | 3.06 | 90.75 | 98.42 |

**Memory Usage.** We use Firefox's `about:memory` to measure the memory usage of Firefox. 10 tests are done to measure the usage of SoPB extension. The average memory usage for SoPB is **3.4%** of Firefox's total memory usage. This memory usage can be considered very small. The memory usage of SoPB on Firefox can almost be neglected. CPU usage measurement is not performed, as the CPU usage for Firefox varies much depends on what actions a user conduct.

**Startup Time.** We use an extension `About Startup 0.1.12` to test the startup time for Firefox with and without our SoPB [21], each with 10 tests. We calculate the average startup time based on these tests. Table IX describes four metrics when start Firefox. "main" means the Gecko main function is entered. "start" means when it starts to load Firefox. "selectProfile" means Firefox start to choose a profile. "afterProfileLocked" means when Firefox is done with profile selection. The former 3 metrics should not be affected by SoPB. The last one may be affected because SoPB is stored in the profile. The results show that there is almost no difference between two rows in terms of average startup time, considering that they are in milliseconds.

Overall, our evaluation on memory usage and startup time demonstrate that SoPB has very little impact on Firefox's performance. The overhead brought by SoPB should not be a concern for users, developers, and webstores.

### *I. A Case Study on Extension Instrumentation*

We use an extension "Fire.fm" to demonstrate how we did our instrumentation. Fire.fm lets you have direct access on the extensive music library on Last.fm. There are two key challenges in instrumentation. First, locate the privacy breaches in fire.fm. Based on the structure of Firefox extensions, handling privacy browsing data is usually done under "resources" or "components". For fire.fm, this is done under "resources". Second, determine how to handle the private browsing data as mentioned in Section III-G. To demonstrate the two methods, we instrumented fire.fm in both of these two ways.

**Disabling Storing Private Browsing Data in PBM.** The following code shows how to instrument Fire.fm to disable storing the private browsing history (the radio station history). In the function of storing recent station history, `if(!FireFM.Private.isPrivate)` is added to check if the current session is under PBM. When the current session is under PBM, the evaluation in the `if` statement is `false`, so storing station history is disabled. On the other hand, only if the current session is in normal mode (not PBM), should the station history be stored using `this._stationHistory.unshift(aStation)`.

```
 _storeRecentStation : function(aStation) {
    //Check if under PBM session
    if (!FireFM.Private.isPrivate) {
    ... // Do some initialization and checking
    // Add the station at the top of the list.
    this._stationHistory.unshift(aStation);
    }
 },
```

It is considered good practice for the extension to enable respecting PBM based on a preference (choice) specific to that extension, and set that preference to `true` by default [23]. For example, a preference `_historyPref` called "disabling station history" can be added into Fire.fm. So the above `if` statement can be revised as follows:

```
if(this._historyPref.value&&!FireFM.Private.isPrivate)
```

**Clearing any temporarily-stored private data.** The following function `clerRecentHistory` in the code snippet shows one example in clearing the temporarily-stored private data when the last PBM window is closed.

```
function pbObsvr() {/* clear private data */}
var os=Components.classes["@mozilla.org/observer
    -service;1"].getService
    (Components.interfaces.nsIObserverService);
os.addObserver(pbObsvr,"last-pb-context-exited",false);

//Clears the recent station history.
clearRecentHistory : function() {
this._logger.debug("clearRecentHistory");
// clear the list.
this._stationHistory.splice(0,
                    this._stationHistory.length);
 ...
},
```

We then wrap these code, zip the extension, and install it on Firefox. Our testing shows that both of these two methods can stop the privacy breaches.

## VI. Discussion and Limitations

We first discuss the scalability of our approach. Although we use Firefox during implementation and evaluation, our approach can be easily scaled to other browsers. Let us take Chrome as an example. It is much easier for system call tracking in Chrome as each Chrome extension maintains its own process. We do not need the trace differentiator for Chrome extensions, greatly increasing the accuracy. On the other hand, our approach can also be transplanted to Windows platform. Most extensions usually constrain their system calls under 50 common ones. Besides, there is also a "strace for Windows" called `drstrace` to track all system calls executed by a target application [10].

An extension can also control or incorporate another extension to manipulate the behaviors. For example, an extension can include the download manager to display the downloading entries, etc. This is easier to address as each extension has a unique ID. We also track all the child process(es) forked by the parent extension. SCDGs can describe their interactions and dependencies.

Our approach may have two limitations. 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 browser may be treated as the running extension. However, when extracting SCDGs for this extension, most of the mistaken ones will be excluded. Therefore, this possibility affects little on SCDG extraction. The other possibility is that system call traces of the running extension may be treated as the browser's. This may eliminate some system call traces or even SCDGs for this extension. As a result, it is possible that this could lead to false negatives, although the possibility is very small.

Second, for some extensions, it is hard to decide whether the data retained by the extension is used to maintain the functionality of extensions in PBM or not. Hence, when doing instrumentation, if some private browsing data belong to category, we use M2 (allowing extensions retain private data during PBM while clearing them when the last PBM window is closed) to deal with them to avoid an extension functioning abnormally or even crashing.

## VII. Related Work

Although PBM has been in commodity use for just a few years, still there is some work in this area, and a larger area of extension security.

**Static Analysis.** G. Aggarwal *et al.* did a preliminary yet important study on PBM in modern browsers [2]. They defined some goals of PBM in different browsers. A preliminary study was carried out to test whether the current implementations of PBM could defend against the defined threat model [2]. Then they did a manual review on some popular extensions to find out if they violated PBM. This work gives some fundamental study on extensions under PBM. However, what they proposed is to disable all unsafe extensions under PBM, which is not practical. Besides, no effective measures are proposed to enhance privacy for browsers.

B. S. Lerner *et al.* used static analysis to analyze JavaScript extensions for PBM [19]. They built a static type system to verify whether an extension violated PBM policies or not. In evaluation, they retrofitted type annotations to Firefox's API and to some extensions [19]. This work identifies possible violations of PBM for extensions. However, the type system is not an automated process, only a small number of samples have been studied. This work does not propose effective methods to prevent privacy breaches under PBM for extensions. It also does not handle bookmarks and downloaded files under PBM.

**Dynamic Analysis.** B. Zhao and P. Liu proposed an approach to dynamically analyze browser extensions' behavior [33]. They used an aspect-level behavior clustering to detect suspicious extensions. SCDGs are used to represent extensions' behavior. Although we also use system level behavior tracking, our approach is different from theirs in the following perspectives. First, our approach is not intended to detect malicious or suspicious extensions. Our approach is to address the privacy issues associated with browser extensions. Second, their work groups different extensions based on aspect-level clustering, while our approach analyzes extensions only based on itself. Third, we also proposed an approach to stop the privacy breaches via instrumentation.

We used symbolic execution to address the input space issue. Recently, A. Kapravelos *et al.* proposed a system called Hulk to "monitor extension actions and create a dynamic environment that adapts to extension needs in order to trigger the intended behavior of extensions" [16]. Hulk employed a fuzzer to drive the event handlers that modern extensions heavily rely upon. Hulk indeed elicited tons of behaviors even many malicious behaviors. However, Hulk is heavily relying on Chrome. Besides, Hulk is claimed to have limitations on observed behavior that depends on specific targets [16]. Therefore, though Hulk is effective in eliciting extension behaviors, symbolic execution is still employed in our approach.

## VIII. Conclusion

We have proposed an approach to identify and stop privacy breaches under PBM caused by browser extensions. Dynamic analysis and symbolic execution are combined to identify the privacy breach patterns. Based on the privacy breach patterns and state transition diagrams, extensions are instrumented to ensure the privacy protection. Our prototype SoPB based on the approach demonstrates good effectiveness and acceptable performance impact in identifying and stopping privacy breaches caused by extensions.

## Acknowledgment

This work was supported by ARO W911NF-09-1-0525 (MURI), NSF CNS-1223710, NSF CNS-1422594, NSF SBE-1422215, and ARO W911NF-13-1-0421 (MURI). We would like to thank our shepherd Dr. Gilles Muller for his insightful comments.

## References

[1] Add-ons Blog. Private browsing support to be required for add-ons, Feb 2010. https://blog.mozilla.org/addons/2010/02/23/private-browsing-support-required-for-add-ons/.

[2] Gaurav Aggrawal, Elie Bursztein, Collin Jackson, and Dan Boneh. An analysis of private browsing modes in modern browsers. In *Proceedings of 19th Usenix Security Symposium*, 2010.

[3] Apple Support. Safari 5.1 (os x lion): Browse privately. http://support.apple.com/kb/ph5000.

[4] Aurelian Melinte. Monitoring function calls, June 2008. http://linuxgazette.net/151/melinte.html.

[5] Sruthi Bandhakavi, Samuel T. King, P. Madhusudan, and M. Winslett. Vex: Vetting browser extensions for security vulnerabilities. In *Proceedings of 19th USENIX Security Symposium*, pages 339–354, 2010.

[6] Kenton Born. Browser-based covert data exfiltration. *arXiv preprint arXiv:1004.4357*, 2010.

[7] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In *International Conference on Software Engineering*, pages 1066–1071, 2011.

[8] Chrome Help. Incognito mode (browse in private). https://support.google.com/chrome/answer/95464?hl=en.

[9] Mario Couture, R. Charpentier, M. Dagenais, A. Hamou-Lhadj, and A. Gherbi. Self-defence of information systems in cyber-space – A critical overview. In *NATO IST-091 Symposium*, April 2010.

[10] Dr. Memory. System call tracer ("strace") for windows. http://www.drmemory.org/strace_for_windows.html.

[11] Ehsan Akhgari. Prepare your add-on for PBM, 2008. http://ehsanakhgari.org/blog/2008-11-08/prepare-your-add-private-browsing.

[12] Elie Bursztein. 19% of users use their PBM, 2012. http://www.elie.net/blog/privacy/19-of-users-use-their-browser-private-mode.

[13] Waseem 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.

[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] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*

[17] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[18] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *Proceedings of 18th USENIX Security Symposium*, pages 351–366, 2009.

[19] Benjamin S. Lerner, Liam Elberty, Neal Poole, and Shriram Krishnamurthi. Verifying web browser extensions' compliance with private-browsing mode. In *European Symposium on Research in Computer Security*, pages 57–74, 2013.

[20] Linux Man Page. strace. http://linux.die.net/man/1/strace.

[21] Mike Hommey. About startup 0.1.12, Jun 2013. https://addons.mozilla.org/en-US/firefox/addon/about-startup/.

[22] Mozilla. How many Firefox users have add-ons installed? 85%. http://blog.mozilla.com/addons/2011/06/21/firefox-4-add-on-users/.

[23] Mozilla Developer Network. Supporting PBM, 2013. https://developer.mozilla.org/en-US/docs/Supporting_private_browsing_mode.

[24] Mozilla Developer Network. Supporting per-window PBM, 2013. https://developer.mozilla.org/EN/docs/Supporting_per-window_private_browsing.

[25] Mozilla Support. Private browsing - browse the web without saving information about the sites you visit. https://support.mozilla.org/en-US/kb/private-browsing-browse-web-without-saving-info.

[26] Mozilla Wiki. Jetpack. https://wiki.mozilla.org/Labs/Jetpack.

[27] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of Programming Language Design and Implementation*, pages 89–100, 2007.

[28] NetworkX. Advanced interface to VF2 algorithm. http://networkx.lanl.gov/preview/reference/algorithms.isomorphism.html.

[29] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for JavaScript. In *IEEE Symposium on Security and Privacy*, 2010.

[30] Xinran Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Behavior based software theft detection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, New York, NY, USA, 2009.

[31] Windows 7 Support. What is inprivate browsing? http://windows.microsoft.com/en-us/windows/what-is-inprivate-browsing#1TC=windows-7.

[32] Ru-Gang Xu. *Symbolic Execution Algorithms for Test Generation*. PhD thesis, University of California-Los Angeles, 2009.

[33] Bin Zhao and Peng Liu. Behavior decomposition: Aspect-level browser extension clustering and its security implications. In *International Symposium on Research in Attacks, Intrusions and Defenses*, pages 244–264, 2013.