Memoirs of a Browser: A Cross-browser Detection Model for Privacy-breaching Extensions

Cristiano Giuffrida Stefano Ortolani
Vrije Universiteit Amsterdam
De Boelelaan 1081, Amsterdam, The Netherlands
{giuffrida,ortolani}@cs.vu.nl

Bruno Crispo
University of Trento
Via Sommarive 15, Trento, Italy
crispo@disi.unitn.it

Abstract

Web browsers are becoming an increasingly important part of our everyday life. Many users spend most of their time surfing the web, and browser-only operating systems are gaining growing attention. To enhance the user experience, many new browser extensions (or add-ons) are continuously released to the public. Unfortunately, with their constant access to a large pool of private information, extensions are also an increasingly important attack vector. Existing approaches that detect privacy-breaching browser extensions fail to provide a generic cross-browser mechanism that can effectively keep up with the ever-growing number of browser implementations and versions available nowadays.

In this paper, we present a novel cross-browser detection model solely based on supervised learning of browser memory profiles. We show how to carefully select relevant features for the model, which are derived directly from the memory activity of the browser in response to privacy-sensitive events. Next, we use support vector machines to automatically detect privacy-breaching extensions that react to these events. To verify the effectiveness of our model, we consider its application to extensions exhibiting keylogging behavior and discuss an end-to-end implementation of our detection technique. Finally, we evaluate our prototype with the 3 most popular web browsers (Firefox, Chrome, and IE) and against real-world browser extensions. Our experiments confirm that our approach achieves good accuracy and can seamlessly support a variety of browsers with little effort.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection—Invasive software (e.g., viruses, worms, Trojan horses), Information flow controls

General Terms Security

Keywords Browser Security, Privacy, Memory Profiling, Keylogger

ASIACCS '12, May 2–4, 2012, Seoul, Korea.

1. Introduction

Web browsers are undoubtedly one of the most popular user applications. This is even more evident in recent times, with Google introducing a platform (Chromebook [13]) where the browser is the only application provided to the user. With their modular and extensible architecture, modern browsers are also an appealing platforms for third-party software developers, who can easily publish new extensions to extend any standard web browser functionality.

Extendability is a crucial feature that makes web browsers a very attractive service platform. From a security perspective, however, extensions opened up new opportunities for attacks. Most extensions do not require any special privilege to be installed, despite their ability to access all the user private data. Delegating the decision about extension's security to trusted parties is not a conclusive solution, given that privacy-breaching behavior has been found even in store-approved extensions [6]. Furthermore, extensions are becoming the perfect drop for trojans that deploy a malicious extension as part of their infecting procedure [3].

Recent solutions specifically designed to detect privacy-breaching extensions [9, 31] require significant changes to the browser and are typically specific to a particular browser implementation and release. Besides requiring access to the source-code, porting these solutions to all the major browsers requires a significant effort. In addition, maintaining such infrastructures over time is likely to be ill-affordable, given the increasing number of new browser versions released every year, as Figure 1 demonstrates. To deal with these challenges effectively, we advocate the need for more general, cross-browser (i.e., version- and implementation-independent) approaches to detect different classes of privacy-breaching extensions.

In this paper, we present a novel cross-browser detection model for extensions that eavesdrop privacy-sensitive events, and consider, without loss of generality, its appli-

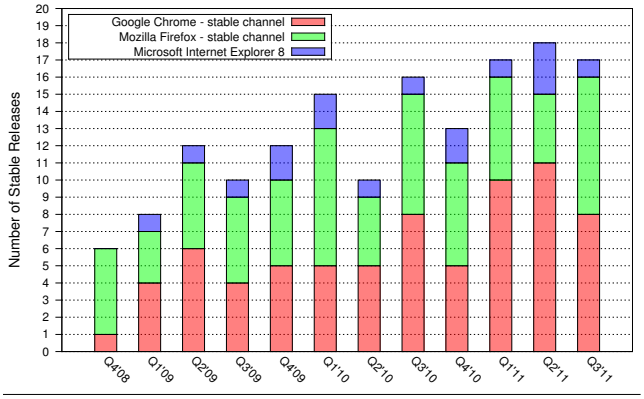

Figure 1. Rate of stable (major and minor) releases [12, 23, 26].

cation to extensions with keylogging behavior. Extensions in this category intercept all the user-issued keystrokes and leak them to third parties. Keylogging extensions are particularly dangerous because they can be easily used in large-scale attacks (i.e., they do not depend on the DOM of the visited page), with the ability to capture all the user sensitive data, including passwords and credit card numbers. For their ease of implementation, they are generally hard to detect and no countermeasure exists for all the browser implementations available. Their simplicity also makes them the ideal privacy-breaching candidate for code injection attacks in vulnerable legitimate extensions. Figure 2 shows how to use a simple and compact payload to inject a full-fledged keylogger in *Feed Sidebar* version < 3.2 for Firefox affected by a typical privileged code injection vulnerability [27].

```
<title>Apparently Legitimate Website</title>
<link>http://www.legitimate.com</link>
<description>
  Legitimate encoded image follows: &lt;iframe src=&q
  uot;data:text/html;base64,PHNjcmlwdD5kb2N1bWVudC5hZ
  GRFdmVudExpc3RlbmVyKCJrZXlwcmVzcyIsZnVuY3Rpb24oZS17
  dmFyIHg9bmV3IFhNTEh0dHBSZXF1ZXN0KCk7eC5vcGVuKCJHRVQ
  iLCJodHRwOi8vbm90LmxlZ2l0aW1hdGUuY29tLz9rPSIrZS53aG
  ljaCxmYWxzZSk7eC5zZW5kKG51bGwpO30sZmFsc2UpOzwvc2Nya
  XB0Pg==&quot;&gt;&lt;/iframe&gt;
</description>
```

RSS item with a malicious Base64 encoded payload.

```
<script>
  document.addEventListener("keypress", function(e) {
    var x = new XMLHttpRequest();
    x.open("GET","http://not.legitimate.com/?k=" +
        e.which, false);
    x.send(null); }, false);
</script>
```

Decoded payload.

Figure 2. Deploying a keylogger via *Feed Sidebar* exploit.

The contributions of this paper are threefold. First, to the best of our knowledge, we are the first to introduce a cross-browser detection model for privacy-breaching extensions designed to completely ignore the browser internals. To fulfill this requirement, our model analyzes only the memory activity of the browser to discriminate between legitimate and privacy-breaching extension behavior. An SVM (support vector machine) classifier is used to learn the properties of a number of available memory profiles and automatically identify new privacy-breaching profiles obtained from unclassified extensions. Second, we discuss a fine-grained memory profiling infrastructure able to faithfully monitor the behavior of modern browsers and derive accurate memory profiles for our model. Our infrastructure can be enabled and disabled on demand, thus allowing convenient user- or policy-initiated detection runs. Finally, we have implemented our detection technique in a production-ready solution and evaluated it with the latest versions of the 3 most popular web browsers: Firefox, Chrome, and IE (as of September 2011 [36]). To test the effectiveness of our solution, we have selected all the extensions with keylogging behavior from a dataset of 30 malicious samples, and considered the most common legitimate extensions for all the browsers analyzed. Our experimental analysis reported no false negatives and a very limited number of false positives.

2. Our Approach

Browsers are becoming increasingly complicated objects that accomplish several different tasks. Despite their implementation complexity, the basic model adopted is still fairly simple, given their event-driven nature. Browser events are typically triggered by user or network input. In response to a particular event, the browser performs well-defined activities that distinctly characterize its reaction. If we consider all the possible components that define the browser behavior (e.g., processes, libraries, functions), we expect independent components to react very differently to the given event.

Browser extensions follow the same event-driven model of the browser. When an extension registers an handler for a particular event, the browser will still react to the event as usual, but will, in addition, give control to the extension to perform additional activities. Since the presence of the extension triggers a different end-to-end reaction to the event, we expect new behavioral patterns to emerge in the activities performed by all the possible components of the browser.

Our approach builds on the intuition that the differences in the reaction to a particular event can reveal fundamental properties of the extension behavior, even with no prior knowledge (e.g., variables used or API functions called) of the exact operations performed in response to the event. More importantly, if we can model the behavior of how particular extensions react to certain events, we can then also identify different classes of extensions automatically. Our detection strategy leverages this idea to discriminate between legitimate and privacy-breaching extension behavior.

Similarly to prior approaches [16, 28], we artificially inject bogus events into the system to trigger the reaction of the

browser to a particular event of interest. Concurrent to the injection phase, the monitoring phase records all the activities performed by the different components of the browser in response to the events injected. The reaction of the browser is measured in terms of the memory activities performed when processing each individual event. Our analysis is completely quantitative, resulting in a black-box model: we only consider the memory access distribution, not the individual data being processed in memory. The reason for using a monitoring infrastructure at this level of abstraction is to ignore browser and extension internals allowing for a cross-browser detection strategy. At the same time, memory profiling allows us to build a very fine-grained behavioral model and achieve better detection accuracy. Furthermore, we can turn on the detection process only when needed, thus limiting the performance impact to short and predictable periods of time.

To model and detect privacy-breaching behavior, our injection phase simulates a number of user-generated events. This is possible by using common automated testing frameworks that simulate the user input. Unlike prior approaches that artificially injected bogus events in the background [16, 28, 29], we need to simulate foreground user activity to trigger the reaction of the browser. In addition, we cannot assume every browser reaction correlated with the input to be a strong indication of privacy-breaching behavior. Browsers normally react to foreground events even if no extension is installed. To address this challenge, our detection model is based on supervised learning.

The idea is to allow for an initial training phase and learn the memory behavior of the browser and of a set of representative extensions in response to the injected events. The training set contains both legitimate and privacy-breaching extensions. The memory profiles gathered in the training phase serve as a basis for our detection technique, which aims to automatically identify previously unseen privacy-breaching extensions. The next sections introduce our memory profiling infrastructure and our detection model, highlighting the role of memory profiles in our detection strategy.

3. Browser Memory Profiling

To gather memory profiles that describe the browser behavior, we need the ability to monitor any memory activity as we artificially inject events into the browser. Naturally, we favor a non-intrusive monitoring infrastructure with minimal impact on the user experience. If slowdowns may be acceptable for a short period of time, it is undesirable to lower the quality of the entire browsing experience. For this reason, we advocate the need for an *online* solution, with no runtime overhead during normal use and the ability to initiate and terminate memory profiling on demand, without changing the browser or requiring the user to restart it.

To overcome these challenges, our solution comprises an in-kernel driver able to profile all the memory accesses by forcefully protecting the address space of the profiled application. This strategy generates memory access violations—i.e., page faults (PFs)—for each memory operation, allowing a custom PF handler in a kernel driver to intercept and record the event. The driver uses shadow page tables to temporarily grant access to the target memory regions and allow the program to resume execution. When the memory operation completes, the driver restores the protection for the target regions to intercept subsequent accesses.

Our profiling strategy is explicitly tuned to address programs as sophisticated as modern browsers, which are well known for their intense memory activity. Instead of intercepting every memory access, we use write protection to intercept and record only memory write operations, while avoiding unnecessary PFs in the other cases. In addition, we introduce a number of optimizations to eliminate other irrelevant PFs (for example on transient stack regions). Filtering out unnecessary PFs is crucial to eliminate potential sources of noise from our browser analysis. Note that intercepting memory writes is sufficient for our purposes, since we are only interested in privacy-breaching extensions that actually harvest (and potentially leak at a later time) sensitive data.

In addition, our kernel driver collects fine-grained statistics on each memory write performed. We record details on the execution context (i.e., the process) that performed the memory write, the program instruction executed, and the memory region accessed. Rather than keeping a journal detailing every single memory operation, we introduce a number of memory performance counters (MPCs from now on) to gather global statistics suitable for our quantitative analysis. Each MPC reflects the total number of bytes written by a particular process' component in a particular memory region in the monitoring window. This is intended to quantify the intensity of the memory activity of a particular process executing a specific code path to write data to a particular memory region. Our driver maintains a single MPC for each available combination of process, code region, code range, and data region. To characterize the memory activity in a fine-grained manner and identify individual code paths more accurately, we break down every code region into a number of independent code ranges of predefined size.

While other approaches have focused on memory profiling at the granularity of individual code regions [29], our experiments revealed this was insufficient to accurately model the behavior of modern browsers. To achieve greater discrimination power, our strategy is to identify key code paths at the level of individual functions being executed. While it is not possible to automatically identify functions in the general case (symbols may not be available), we approximate this strategy by maintaining r different code ranges for each code region.

4. The Model

In this section, we introduce our model and discuss the design choices we made to maximize the detection accuracy.

Our analysis starts by formalizing the injection and monitoring phase of our detection technique.

Definition 1. *An injection vector is a vector* $\boldsymbol{e} = [e_1, \ldots, e_n]$ *where each element* e_i *represents the number of events injected at the time instant* t_i, $1 \leq i \leq n$, *and* n *is the number of time intervals considered.*

The injection phase is responsible to feed the target program with the event distribution given by the vector **e** for a total of $n \times t$ seconds, t being the duration of the time interval considered. In response to every event injected, we expect a well-defined reaction from the browser in terms of memory activity. To quantify this reaction, the monitoring phase samples all the predefined MPCs at the end of each time interval. All the data collected is then stored in a memory snapshot for further analysis.

Definition 2. *A memory snapshot is a vector* $\boldsymbol{c} = [c_1, \ldots, c_m]$ *where each element* c_j *represents the* j*-th MPC,* $1 \leq j \leq m$, *and* m *is the total number of MPCs considered.*

At the end of the monitoring phase, the resulting n memory snapshots are then combined together to form a memory write distribution.

Definition 3. *A memory write distribution is a* $n \times m$ *matrix* $C = [\boldsymbol{c}_1, \ldots, \boldsymbol{c}_n]^T = [c_{i,j}]_{n \times m}$ *whose rows represent the* n *memory snapshots and the columns represent the* m *MPC distributions considered.*

In our model, the memory write distribution is a comprehensive analytical representation of the behavior of the target browser in response to a predetermined injection vector **e**. Once the injection vector has been fixed, this property allows us to repeat the experiment under different conditions and compare the resulting memory write distributions to analyze and model any behavioral differences. In particular, we are interested in capturing the properties of the baseline behavior of the browser and compare it against the behavior of the browser when a given legitimate or privacy-breaching extension is installed.

Our ultimate goal is to analyze and model the properties of a set of memory write distributions obtained by monitoring legitimate browser behavior and a corresponding set of memory write distributions that represent privacy-breaching browser behavior. Given a sufficient number of known memory write distributions, a new previously unseen distribution can then be automatically classified by our detection technique. This strategy reflects a two-class classification problem, where positive and negative examples are given by memory write distributions that reflect privacy-breaching and legitimate browser behavior, respectively.

4.1 Support Vector Machine

To address the two-class classification problem and automatically discriminate between legitimate and privacy-breaching browser behavior, we select support vector machine (SVM) [8] as our binary classification method. SVMs have been largely used to address the two-class classification problem and offer state-of-the-art accuracy in many different application scenarios [22]. An SVM-based binary classifier maps each training example as a data point into a high-dimensional feature space and constructs the hyperplane that maximally separates positive from negative examples. The resulting maximum-margin hyperplane is used to minimize the error when automatically classifying future unknown examples. Each example is represented by a feature vector $\mathbf{x}_i \in \mathbb{R}^d$ and mapped into the feature space using a kernel function $K(\mathbf{x}_i, \mathbf{x}_h)$, which defines an inner product in the target space. To ensure the effectiveness of SVM, one must first carefully select the features that make up the feature vector, and then adopt an appropriate kernel function, kernel's parameters, and soft margin parameter [5]. In our particular setting, the feature vectors must be directly derived from the corresponding memory write distributions. This process applies to any positive, negative, or unclassified example. The next subsections detail the extraction of the relevant features from the memory write distributions considered and discuss how to translate them into feature vectors suitable for our SVM classifier. To select the most effective SVM parameters in our setting, we conducted repeated experiments and performed cross-validation on the training data. All the experiments were conducted using LIBSVM [4], a very popular and versatile SVM implementation. Our experiments showed that the linear kernel with C-SVC $= 10$ and $\gamma = 10$ give the best results in terms of accuracy in the setting considered.

4.2 Feature Selection

The features that constitute the feature vector should each ideally detail how a particular component of the browser reacts to the injection. To achieve this goal, we need to identify a single feature for each of the m MPC distributions. The memory activity associated to a particular MPC is a relevant feature since it documents both how often particular code paths are executed and the volume of memory writes performed in particular memory regions. The next question we need to address is how to represent every single feature associated to a particular MPC. In other words, starting from a MPC distribution, we need to determine a single numeric feature value that is suitable for SVM-based classification.

To address this concern, we immediately observe that different MPC distributions may reflect a completely different behavior of the browser for a particular MPC. If there is no browser activity for a particular MPC, we will observe a corresponding zero MPC distribution. If there is some browser activity but unrelated to the event distribution being injected, we will observe a corresponding MPC distribution that is very dissimilar from the original injection vector. Finally, if the browser activity associated to a particular MPC represents indeed a reaction of the browser to the injection, we will observe a corresponding MPC distribution that closely

resembles the original injection vector. To identify every single scenario correctly, we need a correlation measure that can reliably ascertain whether two distributions are correlated and causality can be inferred with good approximation. For our purposes, we adopt the Pearson Correlation Coefficient (PCC) [28] to measure the correlation between the injection vector and every single MPC distribution.

The PCC is suitable for our purposes since it is both scale and location invariant, properties that make the measure resilient to linear transformations of the distributions under analysis. This translates to the ability to compare the original injection vector with any given MPC distribution, even in face of several memory writes performed for each bogus event injected (scale invariance property) and uniformly distributed browser activity performed in the background (location invariance property). Given two generic distributions P and Q, the PCC is defined as follows:

$$PCC\left(P,Q\right)=\frac{\sum_{i=1}^{N}\left(P_i-\bar{P}\right)\left(Q_i-\bar{Q}\right)}{\sqrt{\sum_{i=1}^{N}\left(P_i-\bar{P}\right)^2}\sqrt{\sum_{i=1}^{N}\left(Q_i-\bar{Q}\right)^2}}$$

In our model, the PCC is used to ascertain whether a particular MPC distribution reflects a reaction of the browser to the injected events. High correlation values indicate browser activity directly triggered by the injection. This is important for two reasons. First, the PCC is used as a feature selection mechanism in our model. If a particular MPC distribution is not correlated to the injection vector for a given browser configuration, the MPC is assumed not to be a relevant feature for the to-be-generated feature vector. All the features deemed irrelevant for all the examples in the training set are automatically excluded from the analysis. Second, the PCC is used to determine whether a particular feature is relevant for the browser in a pristine state (i.e., the baseline behavior of the browser with no extension enabled). This is important when comparing the memory write distribution of a particular extension with the memory write distribution of the baseline to filter out background browser activity and improve the accuracy of the analysis, as explained later.

Once all the relevant features have been identified, we quantify the numerical value of a single feature associated to a particular MPC distribution as the amplification factor computed with respect to the original injection vector. Given that these two distributions exhibit high correlation, we ideally expect an approximately constant amplification factor in terms of number of bytes written for each event injected over all the time intervals considered. This is representative of the intensity of the memory activity associated to a particular MPC and triggered by our injection. Moreover, in order to model the behavior of a particular extension more accurately, the intensity of the memory activity is always measured incrementally, in terms of the number of additional memory writes performed by the extension for each event injected with respect to the baseline. In other words, for each extension, the feature vector can be directly derived from the memory write distribution obtained for the extension, the memory write distribution obtained for the baseline, and the predetermined injection vector used in the experiments. The next subsections present the feature vector used in our model more formally.

4.3 Feature Vector: Ideal Case

Let us consider the ideal case first. In the ideal case, we assume no background memory activity for the browser. This translates to all the MPC distributions reflecting only memory activity triggered by the artificially injected events. As a consequence, a given MPC distribution is either fully correlated with the injection vector (i.e., PCC $= 1$), or is constantly zero over all the time intervals if no event-processing activity is found. The latter assumptions are valid for all the possible memory write distributions (i.e., baseline or extension(s) enabled). Under these assumptions, the number of bytes written for each event is constant (assuming deterministic execution) and so is the amplification factor over all the time intervals.

Let C^B be the baseline memory write distribution and C^E the memory write distribution when a given extension E is instead enabled, both generated from the same injection vector $\mathbf{e}$. The element x_j of the feature vector $\mathbf{x} = [x_1, \ldots, x_m]$ in the ideal case represents the constant amplification factor for the MPC distribution of the j-th memory performance counter, $1 \leq j \leq m$. Each element x_j for any given time interval i can be defined as follows.

$$x_j=\begin{cases}\frac{C^E_{i,j}-C^B_{i,j}}{k_i}+\varepsilon & \text{if } PCC\left(\mathbf{e},C^E_{*,j}\right)\geq T\\ 0 & \text{otherwise}\end{cases}$$

where T is a generic threshold, and ε is the baseline amplification factor.

The rationale behind the feature vector proposed is to have positive amplification factors for each feature that represents a reaction of the browser to our injection. The amplification factor grows as the number of bytes written for each event increases and departs from the baseline value. The correction factor ε is necessary to ensure positive amplification factors even for extensions that behave similarly to the baseline for some feature x_j (i.e., $C^E_{*,j} \approx C^B_{*,j}$). In addition, this guarantees that the feature vector used to represent the baseline during the training phase is always represented by $x_j = \varepsilon$, $1 \leq j \leq m$. Feature values that are not representative of the browser reacting to our injection (i.e., their corresponding MPC distribution is not correlated to the injection vector) are always assumed to be 0. This mapping strategy is crucial to achieve good separability in the feature space.

Note that the constructed feature vector contains only relative amplification measures and is independent of the particular injection vector used, as long as both the baseline and the extension memory write distributions have been generated by the same injection vector. This allows us to

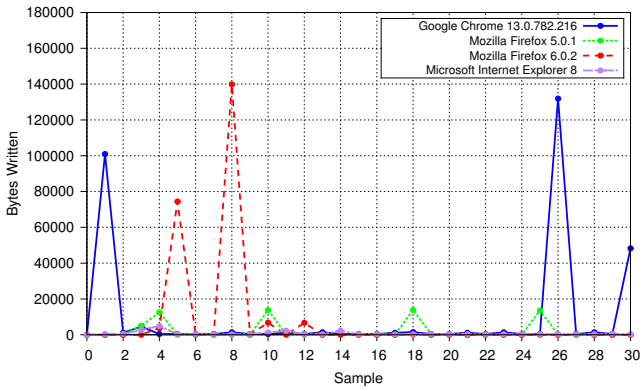

Figure 3. Memory activity of different idle browsers.

use different injection vectors in the training phase and the testing phase with no restriction. More importantly, this allows us to compare correctly amplification factors obtained for different extensions, as long as the baseline is maintained stable. In our model, the baseline characterizes the default behavior of the browser. When switching to a new browser version or implementation, the memory write distribution of the baseline may change and the classifier needs to be retrained. Finally, note the impact of the assumptions in the ideal case. First, the amplification factor is constant over any given time interval. Second, features that are normally irrelevant for the baseline but become relevant for a particular extension are always automatically assumed to be $(1/n)\sum_{i=1}^{n}\left(C_{i,j}^{E}/k_i+\varepsilon\right)$, given that the corresponding baseline MPC distribution is assumed to be constantly 0.

4.4 Feature Vector: Real Case

The construction of the feature vector we presented in the previous section did not address the case of background memory activities interfering with our analysis. Sporadic, but intensive memory writes could hinder the correlation or the amplification factors. Unfortunately, this scenario is the norm, rather than the exception in today's browsers. For example, Firefox is known to continuously garbage collect unused heap regions [15]. Chrome periodically checks for certificates revocation and platform updates. In addition, background activities are often performed by increasingly common AJAX-based web applications that periodically send or retrieve data from the web servers.

To investigate this issue, we recorded the aggregated memory write distribution of all the recent browsers in case of no foreground activity. Figure 3 depicts our findings: with the exception of Internet Explorer (IE), all the browsers perform memory-intensive background tasks. We also observe that the distribution of the background memory activities can considerably vary from one browser version to another.

To make our model resilient to spurious memory activities, we extend our original feature vector in two ways. First, we filter out spurious memory writes monitored for the baseline. This is done by conservatively replacing any MPC distribution with a zero distribution when no significant correlation is found with the injection vector. This operation removes all the background noise associated to features that are not correlated to the event-processing activity in the baseline. This alone is insufficient, however, to eliminate any interference in the computation of the amplification factors when correlated MPC distributions present spurious patterns of background memory activity. To address this problem, we can first increase the granularity of our code ranges in the memory snapshots. This strategy can further isolate different code paths and greatly reduce the probability of two different browser tasks revealing significant memory activity in the same underlying MPC distribution.

In addition, we consider the distribution of the amplification factors over all the time intervals and perform an outlier removal step before averaging the factors and computing the final feature value. To remove outliers from each distribution of amplification factors, we use Peirce's criterion [32], a widely employed statistical procedure for outlier elimination. Peirce's criterion is suitable for our purposes as it allows an arbitrary number of outliers, greatly reducing the standard deviation of the original distribution when necessary. This is crucial for our model, given that we expect a low-variance amplification factor distribution once all the spurious elements have been eliminated. In our experiments, for any reasonable choice of the number of time intervals n, we hardly observed any distribution value distant from the mean after the outlier removal step. We now give the formal definition of the final feature vector used in our model.

Definition 4. *Let the feature vector $\boldsymbol{x}=[x_1,\ldots,x_m]$. The single element x_j of the feature vector measures the average amplification factor for the MPC distribution of the j-th memory performance counter, $1\leq j\leq m$. Each element x_j is defined as follows.*

$$x_j=\begin{cases}\frac{1}{n}\sum_{i=1}^{n}\omega_i\frac{C_{i,j}^{E}-C_{i,j}^{B}}{k_i}+\varepsilon & \text{if } PCC\left(\boldsymbol{e},C_{*,j}^{E}\right)\geq T,\\ & PCC\left(\boldsymbol{e},C_{*,j}^{B}\right)\geq T\\ \frac{1}{n}\sum_{i=1}^{n}\omega_i\frac{C_{i,j}^{E}}{k_i}+\varepsilon & \text{if } PCC\left(\boldsymbol{e},C_{*,j}^{E}\right)\geq T,\\ & PCC\left(\boldsymbol{e},C_{*,j}^{B}\right)<T\\ 0 & \text{otherwise}\end{cases}$$

where T is a generic threshold, ε is the baseline amplification factor, and $\omega_i\in\{0,1\}$ is an outlier removal factor.

5. Case of Study: Keylogging Extensions

This section exemplifies the application of our model to extensions with keylogging behavior and details the steps of the resulting detection process. To instantiate our detection model to a particular class of privacy-breaching extensions, we need to (i) carefully select the injection events to trigger the reaction of interest; (ii) define an appropriate training set that achieves sufficient representativeness and separability between the samples. To satisfy the former, we simply need

to simulate user-issued keystrokes in the injection phase. While we could easily collect several legitimate and privacy-breaching browser extensions to construct the training set to satisfy the latter, in practice we found a minimal synthetic training set to be more convenient for our purposes. Our default training set comprises only 3 examples: the baseline (negative example), a synthetic shortcut manager (negative example), and a synthetic keylogger (positive example).

We implemented all the synthetic examples for each browser examined and found them to be highly representative for our analysis. The baseline accurately models all the extensions that do not intercept keystroke events. Our synthetic shortcut manager, in turn, models all the legitimate extensions that do intercept keystroke events but without logging sensitive data. Our synthetic keylogger, finally, models the privacy-breaching behavior of all the extensions that eavesdrop and log the intercepted keystroke events.

The proposed training set is advantageous for two reasons. First, it can be easily reproduced for any given browser with very little effort. Second, given the simplicity of the synthetic extensions described, the same training set can be easily maintained across different browsers. The only limitation of such a small training set is the inability to train our SVM classifier with all the possible privacy-breaching behaviors. Note that, in contrast, legitimate behaviors are well represented by the baseline and the synthetic shortcut manager. While one can make no assumption on the way privacy-breaching extensions leak sensitive data, our detection strategy is carefully engineered to deal with potential unwanted behaviors that escaped our training phase, as discussed later.

We now detail the steps of the proposed detection process. First, we select suitable injection parameters to tune the detector. We use a random high-variance distribution for the injection vector. This is to achieve low input predictability and stable PCC values. The number n and the duration t of the time intervals, in turn, trade off monitoring time and reliability of the measurements. The larger the duration of a single time interval, the better the synchronization between the injection and the monitoring phase. The larger the number of the time intervals, the lower the probability of spurious PCC values reporting high correlation when no causality was possible.

Subsequently, we train our SVM classifier for the target browser. For each training example we conduct an experiment to inject the predetermined keystroke vector and monitor the resulting memory write distribution produced by the browser. The same is done for the browser with no extensions enabled. The feature vectors are then derived from the memory write distributions obtained, as described in Section 4.4. The training vectors are finally used to train our SVM classifier. The same procedure is used to obtain feature vectors for unclassified extensions in the detection phase.

Before feeding the detection vector to our SVM classifier, the detection algorithm performs a preprocessing step. The vector is checked for any new relevant features that we previously discarded in the feature selection step. If no such a feature is found, the detection vector is normally processed by our SVM-based detector, which raises an alert if the vector is classified as a privacy-breaching extension. If any new relevant feature emerges, in contrast, our detection algorithm always raises an alert indiscriminately. This step is necessary in the general case to eliminate the possibility of privacy-breaching behavior not accounted for in the training phase. This conservative strategy leverages the assumption that legitimate behavior is well represented in the training set, and previously unseen behavior correlated to the injection is likely to reflect unwanted behavior.

6. Evaluation

We tested our approach on a machine with Intel Core i7 2.13 GHz and 4 GB of RAM running Windows XP Professional SP3. We chose the most widespread versions of the browsers analyzed (as of September 2011 [36]): Firefox 6.0.2, Chrome 13.0.782.216, and Internet Explorer 8. In the experiments, we used the injection vector described in Section 5, with $n = 10$ and $t = 500ms$ for an overall detection time of $5s$. These values were sufficient to provide very accurate results.

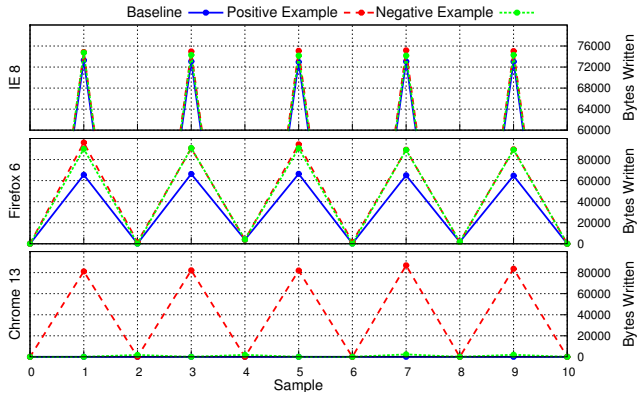

Figure 4. Memory write distributions obtained during a training phase with a sinusoidal-shaped injection vector.

Figure 4 shows the aggregated memory write distributions obtained for the training examples of each browser. As evident from the figure, the correlation alone was never sufficient to discriminate between negative and positive examples. And neither were the aggregated amplification factors, which, for instance, set positive and negative examples only a few bytes apart in Firefox and IE. Nevertheless, the weights assigned to the features during the training phase showed that even with negligible differences in the aggregated amplification factors, individual features can still be used to achieve sufficient discrimination power. For instance, in Firefox and IE we found that the JavaScript (JS) engine libraries (i.e., `mozjs.dll` and `jscript.dll`) played an important role in identifying high-quality features. Chrome, in

contrast, exhibited a limited number of features with very similar weights. While the discrimination power is clearly reduced in this case, Chrome's amplification factors were found far apart between positive and negative examples, thus still revealing a degree of separability suitable for accurate behavior classification.

6.1 False Negatives

To evaluate the effectiveness of our technique we gathered 30 different malicious extensions from public fora, online repositories [14, 25], blocked extensions lists [24], and antivirus security bulletins [37]. We then manually inspected all the samples via static and dynamic analysis and selected only those performing keylogging activities. The resulting dataset comprises 5 full-fledged extensions—also known as Browser Helper Objects (BHOs) in the case of IE 8—, and 1 JS user script compatible with both Firefox and IE, hence obtaining a set of 7 different detection experiments. JS user scripts are stripped down extensions with no packaging infrastructure or ability to modify the existing user interface. Chrome supports user scripts natively when limited privileges are required, whereas Firefox and IE depend upon the installation of the Greasemonkey [20] and Trixie [35] extensions respectively, which also provide access to privileged APIs. We point out that in all cases, regardless of the extension's type, the installation procedure never required superuser privileges.

Browser	Extension	Detected
Chrome 13	`extensionkeylog.sourceforge.net`	✓
	`chrome.google.com/webstore/detail /afllmmeoaodlbocnfihkaihpkcakomco`	✓
Firefox 6	`addons.mozilla.org/addon/220858`	✓
	`userscripts.org/scripts/show/72353`	✓
IE 8	`flyninja.net/?p=1014`	✓
	`wischik.com/lu/programmer/bho.html`	✓
	`userscripts.org/scripts/show/72353`	✓

Table 1. Detection of privacy-breaching extensions performing keylogging activities.

Table 1 shows the results of our experiments. In all the cases, the SVM classifier successfully ascertained the privacy-breaching nature of the samples regardless of the extension type. The most interesting experiments were against the 2 BHO extensions in IE, which are implemented directly by a DLL. The ability of a DLL to independently manage memory may at times produce new relevant features that were nowhere found during the training phase, thus theoretically hindering detection. Our detection strategy, however, gracefully handles this situation in the preprocessing step, which immediately raises an alert when new relevant features are discovered in the detection vector. This ensured all the BHOs could be detected correctly in our experiments.

6.2 False Positives

To test the robustness of our approach against false positives, we put together a dataset of 13 extensions for each browser, comprising the 10 most common extensions [7, 21, 34] and the 3 most popular shortcut management extensions, carefully selected because prone to misclassification. Table 2 shows the results of our detector against all these extensions.

All the extensions for Chrome were correctly classified as legitimate. The grey-colored rows highlight all the shortcut management extensions selected. Despite the presence of keystroke interception APIs, none of these extensions was misclassified. This confirms the robustness of our technique.

In the case of Firefox, 12 out of 13 extensions were classified correctly. The NoScript extension, which blocks any script not explicitly whitelisted by the user, was the only misclassified sample. A quick analysis showed a memory write distribution unexpectedly similar to those exhibited by keylogging samples. A deeper inspection revealed a very complicated implementation of always-on shortcut management functionalities, with every keystroke processed and decoded several times. Other extensions that explicitly provide shortcut management functionalities (grey-colored rows) were instead classified correctly. Similarly to Firefox, only 1 extension (i.e., LastPass, a popular password manager) was erroneously classified for IE. A careful code inspection revealed that the implementation of the extension logs all the user-issued keystrokes indiscriminately. This allows the user to save any previously filled credentials after a successful login. Since the keystrokes are effectively logged and can potentially be leaked to third parties at a later time, our detection strategy conservatively flags this behavior as suspicious.

6.3 Performance

The ability to attach and detach our profiling infrastructure to the browser on demand (as arbitrated by the user) allows us to confine the performance overhead to the detection window. The previous sections have demonstrated that a window of 5 seconds (i.e., 10 samples with a 500ms time interval) is sufficient for our purposes. This confines the overhead to a very limited period of time, allowing the user to start a quick detection run whenever convenient, for example, when vetting unknown extensions upon installation.

Browser	Baseline	Normal use	Detection time
Chrome 13	1345ms	1390ms	11254ms
Firefox 6	1472ms	1498ms	12362ms
IE 8	2123ms	2158ms	14177ms

Table 3. Performance hit while loading `google.com`.

Table 3 show the performance impact of our online infrastructure by comparing the time required to load `http://www.google.com` in three different scenarios: (i) prior to the installation of our infrastructure (`Baseline`), (ii) with our infrastructure installed but completely detached

Google Chrome 13.0.782.216		Firefox 6.0.2		Internet Explorer 8	
Extension	**Identified**	**Extension**	**Identified**	**Extension**	**Identified**
Shortcut 0.2	✓	GitHub Shortcuts 2.2	✓	Shortcut Manager 7.0003	✓
Shortcut Manager 0.7.9	✓	ShortcutKey2Url 2.2.1	✓	ieSpell 2.6.4	✓
SiteLauncher 1.0.5	✓	SiteLauncher 2.2.0	✓	IE7Pro 2.5.1	✓
AdBlock 2.4.22	✓	AdBlock Plus 1.3.10	✓	YouTubeVideoDwnlder 1.3.1	✓
ClipToEvernote 5.1.15.1534	✓	Down Them All 2.0.8	✓	LastPass (IEanywhere)	
Download Master 1.1.4	✓	FireBug 1.8.4	✓	OpenLastClosedTab 4.1.0.0	✓
Fastest Chrome 4.2.3	✓	FlashGot 1.3.5	✓	Star Downloader 1.45.0.0	✓
FbPhoto Zoom 1.1108.9.1	✓	GreaseMonkey 0.9.13	✓	SuperAdBlocker 4.6.0.1000	✓
Google Mail Checker 3.2	✓	NoScript 2.2.1		Teleport Pro 1.6.3	✓
IETab 2.7.14.1	✓	Video Download Helper 4.9.7	✓	WOT 20110720	✓
Google Reader Notifier 1.3.1	✓	Easy YouTube Video 5.7	✓	CloudBerry TweetIE 1.0.0.22	✓
Rampage 3	✓	Download Statusbar 0.9.10	✓	Cooliris 1.12.0.33689	✓
RSS Subscription 2.1	✓	Personas Plus 1.6.2	✓	ShareThis 1.0	✓

Table 2. Classification of legitimate extensions.

(`Normal use`), and (iii) with our infrastructure attached to the browser, hence during detection (`Detection time`). All the experiments have been performed multiple times and their results averaged—with negligible variance. The last two experiments represent the performance overhead perceived by the user during normal use and during detection, respectively. The infrastructure attached to the browser at detection time introduces overhead, ranging from $6.67\times$ for IE to $8.39\times$ for Firefox. When comparing our memory profiler with other solutions that rely on dynamic instrumentation [30], our infrastructure yields significantly lower overhead, for our ability to ignore memory regions of no interest a priori. Finally, the performance variations introduced by our infrastructure when detached is always negligible. This confirms that our technique does not interfere with the normal browsing experience.

7. Discussion

A number of interesting findings emerge from our evaluation. Our model can be effectively used across different browser versions and implementations. We presented results for the most widespread versions of the 3 most popular browsers. We have also experimented with other major releases of Firefox and Chrome obtaining very similar results.

Even if we never found false negatives in our experiments, it is worth considering the potential evasion techniques that a malicious extension may adopt to escape detection. We consider two scenarios. First, an extension could attempt to leak sensitive data by using some browser functionality that was already represented as a training feature. By definition, however, the extension cannot avoid exhibiting relevant memory activity for the particular feature used. The resulting feature value will inevitably reveal a more intensive memory activity with respect to the baseline and contribute to classifying the extension correctly. Conversely, an extension could attempt to rely on some browser functionality that did not emerge as a training feature. In this case, the suspicious behavior will still be detected from the correlation found between the injection vector and the MPC distribution of the emerged feature. The only chance to escape detection is to lower the resulting correlation by performing disguisement activities. While more research is needed to assess the viability of this strategy in the context of browser extensions, prior approaches using PCC-based detection have already discussed the difficulty of such an evasion technique [28]. Finally, an attacker could instruct an extension to perform privacy-breaching activities only in face of particular events, e.g., when the user visits a particular website. To address this scenario, our solution allows the user the start a detection run on all the active extensions at any time, for example before entering sensitive data into a particular website.

Finally, we comment on how to apply our detection model to other classes of privacy-breaching extensions. As done for keylogging extensions, we can easily instantiate our model to any class of extensions that react to certain sensitive events, as long as it is feasible to (i) artificially inject the events of interest into the browser and (ii) determine a training set that achieves separability between positive and negative examples. As an example, to detect form-sniffing behavior, we would need to simulate form submission events and train our model with both form sniffers and regular extensions that do not record form submission events.

8. Related Work

Many approaches [11, 18, 19] have been initially proposed to detect privacy-breaching browser extensions, and in particular the class of malicious software known as spyware add-ons. These approaches relied on whole-system flow tracking [11] and on monitoring the library calls between browser and BHOs [18]. Besides being tailored to IE and hence not meeting the requirement of a cross-browser detection model, they are either dependent on the adopted window of observation for a successful detection, or unable to set apart malicious add-ons from legitimate extensions using the same

library calls. In the case of [19], the interactions of a BHO with the browser are regulated by a set of user-defined policies. However, this approach can not be applied to extensions where the code run in the same context of the browser.

Recently new approaches focused on taint tracking the execution of JS by either instrumenting the whole JS engine [9, 33], or rewriting the JS scripts according to some policies [17]. In both cases the underlying idea is that an object containing sensitive information shall not be accessed in an unsafe way. In our setting this translates to an extension that shall never be allowed to disclose the user's private data to a third-party. All these approaches however, besides incurring high overheads, can not be disabled unless the user replaces the instrumented binary with its original version. Furthermore they fail to meet our cross-browser requirements. In particular, given the complexity of modern JS engines, porting and maintaining them to multiple versions or implementations is both not trivial and requires access to the source code. Besides being feasible only for browsers which source-code is freely available, e.g., Firefox and Chrome, only the vendor's core teams have all the knowledge required for the job. In contrast, our approach merely requires to retrain the model to retrofit different versions and implementations. This does not require any specific knowledge about the browser, takes a limited amount of time, and can also be carried out by unexperienced users.

Since browsers and their extensions were more and more both target and vector of malicious activities, many studies recently addressed the more general problem of assuring the security of the whole browser, extensions included. In particular, Djeric et al. [10] tackled the problem of detecting JS-script escalating to the same privileges of a JS-based extension, hence nullifying the protection provided by the browser's sandbox. This may happen for two different reasons: in case of bugs in the browser implementation or in case of a poorly programmed extension, where the input is not sufficiently sanitized. In the last scenario, [2] proposed a framework to detect these bad practices and help vetting extensions. In any case the mischief is always the ability to load arbitrary code, possibly acquiring higher privileges. No protection is provided against extensions intended to be malicious that disclose private data on purpose.

The basic idea of relying on the correlation between the activity of a program and its input has been initially introduced in [1, 28], where the main focus was the class of monitoring applications. These applications execute in the background and intercept all the keystrokes regardless of the application being used by the user. Besides being tailored to a limited class of privacy-breaching behaviors, monitoring a program in terms of network [1] and I/O activity [28] is a coarse-grained approach also bound to fail when the gathered private data is not immediately leaked away. The approach proposed in [29] raised the bar by adopting a more fine-grained approach where individual memory accesses were monitored; since memory accesses can not be delayed or postponed, they were able to overcome the limit of the adopted window of observation. However, all these approaches cannot be used to solve the problem of detecting privacy-breaching browser extensions. First, the class of events deemed sensitive is limited to user-issued keystrokes. Second, a browser always reacts to its input, thus making a correlation test prone to false positives. Third, they all assume the malicious program to run in the background, thus failing to identify a misbehaving browser because of a privacy-breaching extension installed.

9. Conclusions and Future Work

With their growing availability and ease of distribution, browser extensions pose a significant security threat. In particular, privacy-breaching extensions that intercept and log sensitive events are becoming increasingly widespread. Existing solutions designed to detect privacy-breaching extensions are typically tailored to a particular browser version or require significant efforts to support and maintain multiple browser implementations over time. Unfortunately, browsers undergo continuous changes nowadays and the need for cross-browser detection techniques is stronger than ever.

In this paper, we introduced a generic cross-browser detection model to address this important concern. In addition, we showed an application of the model to privacy-breaching extensions with keylogging behavior, and we evaluated both effectiveness and precision against a set of real-world extensions. We showed that the performance overhead introduced by our detection infrastructure is confined to a very limited time window, hence relieving the user from unnecessary overhead during the normal browsing experience.

In our future work, we plan to further validate our model against several classes of privacy-breaching extensions. In particular, due to the recent gain of momentum [6], our next focus is validating our model with extensions surreptitiously intercepting form submissions. In addition, we are planning to investigate context-specific policies to automatically initiate a detection run in the background (e.g., in face of particular events or when the browser is idle), thus increasing the dynamic coverage of our analysis to effectively address trigger-based behavior.

References

[1] Y. Al-Hammadi and U. Aickelin. Detecting bots based on keylogging activities. *Proceedings of the Third International Conference on Availability, Reliability and Security*, pages 896–902, 2008.

[2] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. Vex: Vetting browser extensions for security vulnerabilities. *Proceedings of the 19th USENIX Security Symposium (SSYM '10)*, pages 339–354, 2010.

[3] Bitdefender. Trojan.PWS.ChromeInject.B. `http://www.bitdefender.com/VIRUS-1000451-en--Trojan.PWS.`

ChromeInject.B.html. Accessed: November 2011.

[4] C.-C. Chang and C.-J. Lin. Libsvm: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2:1–27, May 2011.

[5] O. Chapelle, V. Vapnik, O. Bousquet, and S. Mukherjee. Choosing multiple parameters for support vector machines. *Machine Learning*, 46:131–159, March 2002. ISSN 0885-6125.

[6] G. Cluley. Mozilla pulls password-sniffing firefox add-on. http://nakedsecurity.sophos.com/2010/07/15/mozilla-pulls-passwordsniffing-firefox-addon/. Accessed: November 2011.

[7] CNET. Internet explorer add-ons. http://download.cnet.com/windows/internet-explorer-add-ons-plugins. Accessed: September 2011.

[8] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20:273–297, 1995.

[9] M. Dhawan and V. Ganapathy. Analyzing information flow in javascript-based browser extensions. *Proceedings of the 2009 Annual Computer Security Applications Conference (ACSAC 2009)*, pages 382–391, 2009.

[10] V. Djeric and A. Goel. Securing script-based extensibility in web browsers. *Proceedings of the 19th USENIX Security Symposium (SSYM '10)*, pages 355–370, 2010.

[11] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. *Proceedings of the 2007 USENIX Annual Technical Conference (ATC '07)*, pages 1–14, 2007.

[12] Google. Google Chrome Releases. http://googlechromereleases.blogspot.com, . Accessed: November 2011.

[13] Google. Chromebook. http://www.google.com/chromebook/, . Accessed: November 2011.

[14] Google. Chrome Web Store. https://chrome.google.com/webstore, . Accessed: November 2011.

[15] Graydon. Cycle collector landed. http://blog.mozilla.com/graydon/2007/01/05/cycle-collector-landed/. Accessed: November 2011.

[16] J. Han, J. Kwon, and H. Lee. Honeyid: Unveiling hidden spywares by generating bogus events. *Proceedings of The IFIP TC11 23rd International Information Security Conference*, pages 669–673, 2008.

[17] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in javascript web applications. *Proceedings of the 17th ACM conference on Computer and communications security (CCS 2010)*, pages 270–283, 2010.

[18] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer. Behavior-based spyware detection. *Proceedings of the 15th USENIX Security Symposium (SSYM '06)*, pages 273–288, 2006.

[19] Z. Li, X. Wang, and J. Y. Choi. Spyshield: Preserving privacy from spy add-ons. *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID 2007)*, pages 296–316, 2007.

[20] A. Lieuallen. Greasemonkey. https://addons.mozilla.org/en-US/firefox/addon/greasemonkey/. Accessed: November 2011.

[21] Y. Mankani. 12 Most Popular Google Chrome Extensions Of 2011. http://www.techzil.com/12-most-popular-google-chrome-extensions-of-2011. Accessed: September 2011.

[22] D. Meyer, F. Leisch, and K. Hornik. The support vector machine under test. *Neurocomputing*, 55(1-2):169–186, 2003.

[23] Microsoft. Microsoft Security Bulletin Search. http://www.microsoft.com/technet/security/current.aspx. Accessed: November 2011.

[24] Mozilla. Blocked Add-ons. https://addons.mozilla.org/en-US/firefox/blocked/, . Accessed: November 2011.

[25] Mozilla. Add-ons for Firefox. https://addons.mozilla.org/en-US/firefox/, . Accessed: November 2011.

[26] Mozilla. Firefox Releases. http://www.mozilla.com/en-US/firefox/releases/, . Accessed: November 2011.

[27] Nick Freeman. Feed sidebar firefox extension - privileged code injection. http://lwn.net/Articles/348921/. Accessed: December 2011.

[28] S. Ortolani, C. Giuffrida, and B. Crispo. Bait your hook: a novel detection technique for keyloggers. *Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection (RAID 2010)*, pages 198–217, 2010.

[29] S. Ortolani, C. Giuffrida, and B. Crispo. KLIMAX: Profiling memory write patterns to detect keystroke-harvesting malware. *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID 2011)*, pages 81–100, 2011.

[30] D. Quist. Covert debugging circumventing software armoring techniques. *Black Hat Briefings*, 2007.

[31] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation (PLDI '10)*, pages 1–12, 2010.

[32] S. Ross. Peirce's criterion for the elimination of suspect experimental data. *Journal of Engineering Technology*, 20, 2003.

[33] M. Ter Louw, J. Lim, and V. Venkatakrishnan. Enhancing web browser security against malware extensions. *Journal in Computer Virology*, 4:179–195, 2008.

[34] TricksMachine. The Top 10 Mozilla Firefox Add-ons, June 2011. http://www.tricksmachine.com/2011/06/the-top-10-mozilla-firefox-add-ons-june-2011.html. Accessed: September 2011.

[35] Various Authors. Trixie. http://www.bhelpuri.net/Trixie/. Accessed: October 2011.

[36] W3Schools. Web Statistics and Trends. http://www.w3schools.com/browsers/browsers_stats.asp. Accessed: December 2011.

[37] C. Wuest and E. Florio. Firefox and malware: When browsers attack. *Symantec Security Response*, pages 1–15, 2009.