2012 IEEE Symposium on Security and Privacy

Memento: Learning Secrets from Process Footprints

Suman Jana and Vitaly Shmatikov
The University of Texas at Austin

***Abstract*—We describe a new side-channel attack. By tracking changes in the application's memory footprint, a concurrent process belonging to a different user can learn its secrets. Using Web browsers as the target, we show how an unprivileged, local attack process—for example, a malicious Android app—can infer which page the user is browsing, as well as finer-grained information: whether she is a paid customer, her interests, etc.**

This attack is an instance of a broader problem. Many isolation mechanisms in modern systems reveal accounting information about program execution, such as memory usage and CPU scheduling statistics. If temporal *changes* in this public information are correlated with the program's secrets, they can lead to a privacy breach. To illustrate the pervasiveness of this problem, we show how to exploit scheduling statistics for keystroke sniffing in Linux and Android, and how to combine scheduling statistics with the dynamics of memory usage for more accurate adversarial inference of browsing behavior.

I. Introduction

Modern software increasingly leverages OS mechanisms to improve its security and reliability. For example, every Android application runs as a separate user, relying on the OS to ensure that different applications do not affect each other. Network daemons run as user "nobody" rather than "root," using the same mechanism for privilege separation. The Chrome browser forks a new process for every site instance, relying on the OS process isolation. Other browsers, including Internet Explorer, are moving to the process-per-tab model so that crashes in one tab do not affect other tabs.

In this paper, we show that the reliance on OS abstractions has unintended consequences. It prevents malicious programs from reading the files or memory of another application, but the "accounting" API supported by standard OS isolation mechanisms can indirectly leak information about an application to concurrently executing programs—even if they belong to a different user!

Consider an attacker who gets to execute an unprivileged, user-level process on the victim's machine. For example, he convinces the victim to run a utility or game app on her Android smartphone—a seemingly safe decision, because each Android app runs as a separate user. Any Android app, however, can measure the memory footprint (data+stack size) or CPU scheduling statistics of another app using the standard Unix `proc` facility without any permissions or the phone owner's consent. At first glance, this feature appears harmless: it may reveal that the other app is a memory or CPU hog, but seems unlikely to leak any secrets.

Our contributions. In the 2000 movie "Memento," the main character, suffering from anterograde amnesia, writes out his memories in small increments using snapshots and tattoos. When put together, these snippets reveal the answer to a murder mystery. In this paper, we show that the dynamics of memory footprints—sequences of snapshots of the program's data resident size (DRS)—are correlated with the program's secrets and allow accurate adversarial inference. This robust side channel can be exploited in any multi-user environment: for example, by a malicious app on an Android smartphone or a nosy user on a shared workstation.

We focus on Web browsers as an example of a sophisticated application that keeps important secrets (browsing behavior). After a brief introduction to memory management in modern browsers, we explain how differences in content rendered by the browser manifest in the browser's DRS, i.e., total size of its heap, stack, and `mmap`-allocated memory, and how a concurrent attack process can measure the browser's DRS even if it belongs to a different user.

We used Chrome, Firefox, and the default Android browser to render the front pages of Alexa top 100,000 websites and measured the corresponding patterns of changes in each browser's DRS. Depending on the browser, between 30% and 50% of these pages are *distinguishable*: they produce patterns that are both stable (similar across visits to the same page) and diverse (dissimilar to visits to other pages). The attacker can thus pre-compute a database of browser-specific "signatures" for the distinguishable pages.

We give an algorithm for matching attack measurements against this database. Stability and diversity ensure that the matching threshold can be set to produce no false positives: any successful match to a signature of some distinguishable page is correct. We also measure the true positive rate (*recognizability*), i.e., how often a visit to each distinguishable page produces a match.

In addition to inferring which pages the victim is browsing, we show how to combine the dynamics of the browser's memory usage with secondary side channels to (1) improve accuracy of inference, and (2) track the state of Web sessions and infer finer-grained information, including the victim's relationship with the site, her interests, etc. Our attack also works against OP [8] and all WebKit-based browsers, but we omit the detailed results for space reasons.

Attacks exploiting the dynamics of memory usage are a symptom of a bigger problem and have implications for all multi-user systems. *Any* fine-grained accounting information

DOI 10.1109/SP.2012.19

about program execution revealed by the OS, sandbox, or another containment mechanism is a potential leakage channel, as long as temporal *changes* in this information are (a) correlated with the program's secrets and (b) can be observed by the attacker.

To illustrate this thesis, we show how the CPU scheduling statistics, which are public by default for any process in Linux and Android, can be exploited for keystroke sniffing and improving accuracy of the memory-footprint attack.

II. Related work

Zhang and Wang were the first to observe that the information revealed by the Unix `proc` filesystem can be used for adversarial inference. Their attack exploits the correlation between the program's system calls and its ESP (stack pointer) value for keystroke sniffing [20]. This attack does not work well against nondeterministic programs, including non-trivial GUI applications such as Web browsers. Our main attack exploits an entirely different side channel (dynamics of the browser's memory usage) and infers the page being browsed by the victim, as well as finer-grained information, even if they are not associated with keystrokes (e.g., the user clicks on a link). Unlike ESP, which has no legitimate uses for other processes, the size of the memory footprint is used by popular utilities and revealed by most operating systems even when `proc` is not available.

Starting from [11], there has been much work on analyzing encrypted communications, including webpage identification [6, 16], state of Web applications [5], voice-over-IP [19], and multimedia streams [13]. Our attack model is fundamentally different. In contrast to a *network attacker* who observes the victim's network communications, ours is a *local attacker* who simply runs an unprivileged process as a different user on the victim's machine. This attacker can observe only very coarse information: the target application's memory usage, total number of its context switches, etc. In Section IV, we explain why webpage fingerprinting based on object sizes does not work in this setting.

A famous bug in the TENEX operating system for PDP-10 allowed malicious users to trigger memory-page faults in the middle of password matching, thus enabling letter-by-letter password guessing. Side-channel attacks can also exploit compromising radiation [7, 18], sounds [1], and reflections [2]. On smartphones, side channels include on-board sensors [14] and touch-screen motion [3]. Timing analysis of keystrokes is discussed in [9, 15].

III. Overview of the attack

The basic setting for our attack is two processes executing in parallel on the same host. The processes belong to different users. We refer to them as the *target process* (victim) and *attack process* (attacker). The attack process is unprivileged and does not have root access to the host.

We focus mainly on learning the secrets of Web-browser processes. For example, the victim is an Android browser, while the attacker is masquerading as a game or utility while trying to infer which page the phone owner is browsing, her relationship with the site (is she a paying customer or not?), etc. Similarly, on a multi-user workstation (e.g., in a computer lab on a university campus), a malicious user may be trying to learn which pages are being browsed by the concurrent users of the same workstation.

Measuring the target's memory footprint. The only information needed for our basic attack is the size of the target's memory footprint. By default in Linux and Android, the `drs` field in the `/proc/<pid>/statm` file reveals *data resident size* of the process identified by `pid`, i.e., the combined size of its anonymous memory (allocated via `mmap`), heap (allocated via `brk`), and stack. This value is calculated in the kernel as `mm->total_vm - mm->shared_vm`. In FreeBSD, memory footprints can be measured via `kvm_getprocs` and via utilities like `ps`.

In Android, an application can use this method to measure the memory footprint of another application *regardless of what it lists in its manifest*. This requires neither the phone owner's consent, nor any permissions beyond what the application needs for the legitimate side of its operation.

In Windows, the Performance Data Helper (PDH) library can be used to install a counter for measuring private, non-shared memory pages of a process [12] (equivalent to data+heap+code in Linux). `GetProcessMemoryInfo` can be used to measure the process's working set size (private+shared pages).[1] iOS provides no per-process memory usage information, but `host_statistics` shows system-wide free, resident, wired, active, and inactive pages.

In addition to exploiting memory usage information revealed by `proc`, in Section VIII we show how to exploit CPU scheduling statistics. In some systems, `proc` may also reveal certain networking information, such as the system-wide list of IP addresses, but if the host is connected to the Internet via a proxy, this list contains only the address of the proxy. As a local, user-level process, the attacker cannot observe the target's network traffic.

Building the signature database. The attacker first profiles the target program and creates *attack signatures*. They capture the relationship between the program's secrets and the pattern of changes in its memory usage.

To attack browsers, this stage requires large-scale webpage analysis, repeated for every browser because footprints are browser-specific. There are sufficiently few popular browsers that the signature database can be pre-computed for every major version of every common browser. We did this for Chrome, Firefox, and the default Android browser.

[1] http://msdn.microsoft.com/en-us/library/windows/desktop/ms683219(v=vs.85).aspx

To build the signature database, the attacker selects w webpages (w should be as large as feasible) and visits each page n times. While the browser is loading the page, the attacker periodically measures the browser's memory footprint as described in Section V. The sequence of measurements for each page is converted into a **memprint**, a set of (E, c) tuples where E is an integer representing a particular footprint size, c is how often it was observed during measurement. Memory usage is reported by the OS at page granularity, thus each footprint value E is the number of memory pages used by the process.

For example, this is the partial memprint of `google.com` when loaded by Chrome:

```
(2681,1) (2947,2) (2948,1) (3203,2)
                 ..............
(17168,1) (17172,1) (17204,1) (17210,1)
```

On the left side of these tuples are `drs` values read from the 6th field of `/proc/<pid>/statm`. They represent Chrome's data resident size (DRS) measured in memory pages. The values on the right are how many times this DRS was observed while Chrome was loading the webpage.

The number of tuples in a memprint depends on both the webpage and the browser. Table I shows the distribution of memprint sizes for the front pages of popular websites.

Table I
MEMPRINT SIZES FOR THE FRONT PAGES OF ALEXA TOP 1,000 SITES.

Browser	Memprint size		
	Min	Max	Avg
`Chrome`	13	536	80
`Firefox`	81	1117	247
`Android`	6	343	79

Given two memprints m_1 and m_2, $m_1 \cap m_2$ and $m_1 \cup m_2$ are computed in the standard way:

$$((E, c_1) \in m_1) \wedge ((E, c_2) \in m_2) \Rightarrow (E, min(c_1, c_2)) \in m_1 \cap m_2$$
$$((E, c_1) \in m_1) \wedge ((E, c_2) \in m_2) \Rightarrow (E, max(c_1, c_2)) \in m_1 \cup m_2$$

We compute similarity between two memprints using the Jaccard index: $J(m_1, m_2) = \frac{|m_1 \cap m_2|}{|m_1 \cup m_2|}$. The higher the index, the more similar the memprints.

Different visits to the same page may produce different memprints due to the nondeterminism of the browser's memory allocation behavior even when rendering the same content (see Section IV). The attacker may store all n memprints in his database as the "signature" associated with the page. This requires $O(mnw)$ storage, where m is the size of a single memprint (see Table I). An alternative is to cluster the memprints and use the set of cluster centers as the signature. For simplicity, we used the former method in the experiments presented in this paper.

Some pages produce highly variable memory allocations in the browser due to content variation between visits (see Section IV). They are removed from the database, leaving only the pages for which a significant fraction of the repeated visits produce similar memprints. The similarity threshold is a parameter of the system; it controls the tradeoff between false negatives and false positives (see Section VI).

Performing the attack. The attack process runs concurrently with the browser process and periodically measures the latter's memory footprint as described above. Different browser versions have different base footprints, enabling the attacker to infer which signature database to use. The attacker can download the database or send attack memprints to a remote server for offline or online matching.

We use the term *attack memprint* to refer to the attacker's measurements of the browser's memory footprint as the browser loads some webpage. The attack memprint is matched against the signature database using Algorithm 1.

Algorithm 1 Main steps of the matching algorithm

Input: Signature database D, attack memprint s_m
Output: Matched page or no match

```
for each page p in D do
  for each signature sig_p for page p in D do
    if J(s_m, sig_p) > threshold then
      Return matched page p
    end if
  end for
end for
Return no match
```

In Section VI, we show how to tune the parameters of the algorithm so that it produces at most one match, with no false positives. A successful match thus reveals which page the victim is browsing. In Section VII, we extend the attack to infer finer-grained information by exploiting the semantics of pages within the site and secondary side channels.

IV. MEMORY MANAGEMENT IN MODERN BROWSERS

To explain why the attack of Section III can use temporal changes in the size of the browser's memory footprint to infer which webpage is being loaded, we give a brief overview of memory management in modern browsers. The same principles apply to other applications, too. The discussion below is Linux-centric, but memory allocators in other operating systems use similar techniques.

Memory allocation in browsers. Fig. 1 shows an overview of browser memory management. With a few exceptions (e.g., Lobo browser implemented in Java), browsers are implemented in languages from the C family and thus responsible for their own memory management. Most memory allocations within the browser are caused by loading and rendering Web content: running the JavaScript engine, storing DOM trees, rendering large images, etc.

Different browsers use different allocators. For example, the main Firefox and Chrome processes use `jemalloc` and `tcmalloc`, respectively. Dynamically and statically linked libraries often have their own allocators. Libraries linked to Firefox allocate memory in 22 different ways.[2]

In Linux, memory is requested from and returned to the OS using either `brk`, or `mmap`/`munmap`. `brk` increases or decreases contiguous heap space by the requested amount; if the top of the heap is being used by the program, free space below cannot be returned to the OS. `mmap` can allocate non-contiguous space at page granularity and, unlike `brk`, freed pages can be returned to the OS. On some systems, `mmap` is significantly slower than `brk`. Memory usage reported by the OS is rounded up to page size (4KB in Linux).

Most allocators try to minimize the overhead of system calls and do not call the OS on every `malloc` and `free`. Instead, they use `mmap` or `brk` to obtain a chunk of memory from the OS and manage it internally, allocating and freeing smaller blocks in response to the program's requests.

The allocator maintains a list of free blocks, which can be of different fixed sizes (bins) to minimize fragmentation when the program's allocation request does not match the size of an available block exactly. When the program frees memory, it may be added to the list and not returned immediately to the OS. This has several consequences. First, a process's memory footprint from the OS's viewpoint is always an overestimate of its actual memory usage. Second, small allocations do not result in changes in data resident size and thus remain invisible to the attacker.

Whether a particular `malloc` call made by the program results in a `mmap` or `brk` call to the OS depends on the allocator. For example, by default the `malloc` implementation in `glibc` uses `mmap` for allocations greater than 128 KB. This allocator is employed by the Qt user-interface library, which is used in the OP browser. Most allocation requests made by the user-interface component of OP are below the threshold and thus managed internally by the allocator, resulting in no visible changes in the observable footprint. (This does not affect the efficacy of our attack against OP because we target the rendering process, not the UI.) Another factor is the amount of fragmentation in the allocator's internally managed memory. It depends on the allocation sequence, as well as `mmap`/`brk` usage.

We use the term *sensitivity* for the extent to which the program's memory allocation behavior "filters through" the allocator and becomes visible to the OS and—via memory usage reports—to the attacker. We say that an allocator has good sensitivity if (1) big differences in the size and order of objects allocated by the program result in big differences in the allocator's `mmap`/`brk` behavior, but (2) small differences in the program's allocation behavior result in few differences in the allocator's `mmap`/`brk` behavior. Intuitively, our attack succeeds if visits to different pages produce different `mmap`/`brk` patterns, but variations between visits to the same page produce the same pattern.

Figs. 2 through 5 illustrate that (a) the browser's allocation requests do *not* directly translate into OS-visible changes in the memory footprint, and (b) the pattern of OS-visible footprints varies between different pages.

Unlike C/C++ allocators, allocators for managed languages like Java have very low sensitivity. Therefore, the attack does not work against browsers implemented in Java.

Memory management varies significantly not only between browsers, but also between major releases of the same browser, thus attack signatures are only valid for a particular browser and major version. Fortunately, only a few browsers are in common use, thus the requirement to compute multiple signature databases is unlikely to present a significant obstacle to the attacker.

Justly or unjustly, the size of the memory footprint is a popular metric for comparing browsers. Therefore, browser implementors try hard to minimize it. For example, MemShrink is an active software engineering task force that works on reducing the footprint of Firefox.[3] Most of these changes improve the efficacy of our attack because they make OS-visible footprints more sensitive to the browser's inputs. For instance, in recent versions of Firefox, as soon as the browser frees a large image, the allocator immediately returns memory to the OS. This produces an observable change in the footprint, benefiting the attacker.

Nondeterminism of allocation behavior. Typically, given the same sequence of requests, allocators are largely deterministic. Requests made by the browser, however, are not the same even when rendering the same HTML content because of threads, event-driven user interfaces, and JavaScript execution, which is nondeterministic because of garbage collection, just-in-time compilation, etc. Furthermore, repeated visits to the same page may return different HTML due to changes in ads and other dynamic content, thus changing

[2] http://blog.mozilla.com/nnethercote/2011/01/07/memory-profiling-firefox-with-massif-part-2/

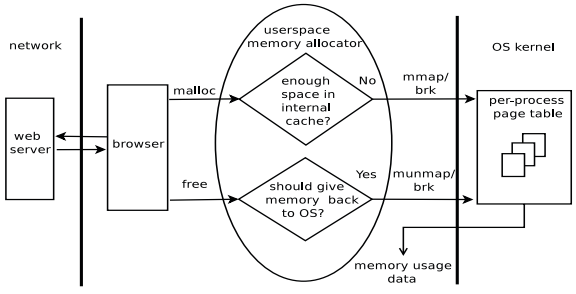

Figure 1. Overview of browser memory management and memory usage reporting.

[3] For example, see http://blog.mozilla.com/nnethercote/2011/10/19/memshrink-progress-weeks-13-18/

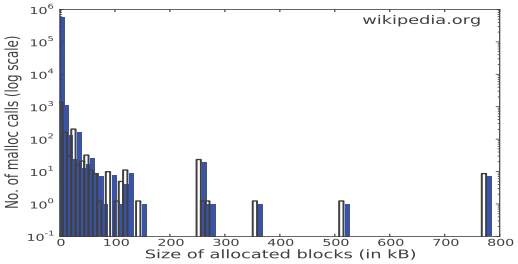

Figure 2. Firefox: Distribution of malloc'd block sizes.

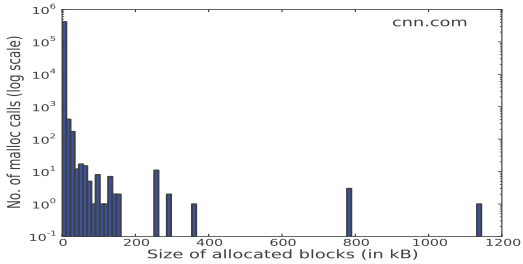

Figure 3. Firefox: Distribution of malloc'd block sizes.

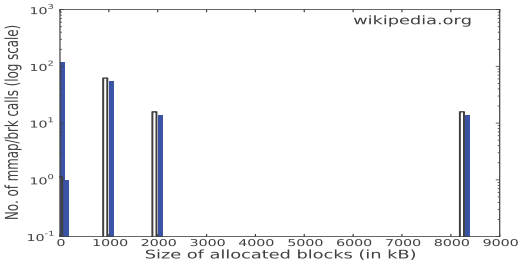

Figure 4. Firefox: Distribution of mmap/brk allocation sizes.

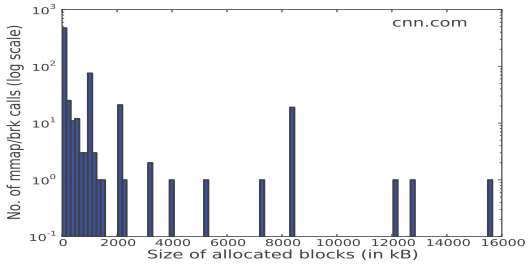

Figure 5. Firefox: Distribution of mmap/brk allocation sizes.

the browser's allocation and de-allocation patterns.

Fragmentation causes the allocator to issue more frequent requests to the OS. As a consequence, the pattern of changes in the OS-visible memory footprint varies even for the same HTML content. This introduces noise into the attacker's measurements and decreases accuracy of the attack. Fortunately, modern allocators aim to minimize fragmentation.

Requirements for a successful attack. Recall from Section III that the attack process periodically measures the target's OS-visible memory footprint. For the attack to work, these measurements must be diverse and stable. *Diversity* means that the sequences of changes in the monitored process's memory usage must vary significantly between different webpages. *Stability* means that these sequences—or at least a significant fraction of them—must be similar across repeated visits to the same page.

The key decision is which process to monitor. "Monolithic" browsers like Firefox run as a single process, but in modern browsers like Chrome or OP, a separate process is responsible for each piece of the browser's functionality. For example, OP has dedicated processes for the browser kernel, user interface, cookie management, networking, and database management. For these browsers, the attacker must choose between monitoring a particular browser component, with more measurements per process, or multiple components, with fewer measurements per process.

In our experiments, we opted for more measurements per process and monitored only the rendering process. This process takes HTML content as input, parses it, creates DOM trees, executes JavaScript, uncompresses images, and generates raw bitmaps to be displayed via the user interface. Most of these tasks are memory-intensive and memory usage of the rendering process satisfies both stability and diversity.

Memory usage of other processes—for example, cookie manager or HTML5 local storage handler—can provide secondary channels to improve accuracy of the attack. For example, a page setting multiple cookies will result in memory allocations in the cookie manager, differentiating it from a page that does not set cookies.

Our attack exploits temporal changes in the memory footprint caused by dynamic allocations from the heap and `mmap`'d regions. Data resident size visible to the attacker also includes stack, but we found in our experiments that the number of pages allocated for the stack rarely changes.

Comparison with network attacks. It is well-known that webpages can be fingerprinted using the number and size of objects on the page [5, 16], provided the attacker has access to the user's network packets and can thus directly observe the sizes of objects on pages requested by the user—even if the page is encrypted.

Fingerprinting webpages using object sizes does not work in our setting. There is no 1-to-1 correspondence between individual objects and memory allocation requests to the OS (see Figs. 2 and 4). There are many allocation sites in the browser and its libraries, and the attacker who monitors the browser's memory usage observes only the cumulative effect of multiple allocations corresponding to multiple HTML objects. This information is significantly more coarse-grained

than that available to the network attacker.

Allocations caused by executing JavaScript and parsing DOM trees are not related to the size of HTML objects on the page. Image decompression results in bigger allocations during rendering than the size of the image object, big objects may be written into temporary files piece by piece and never stored in memory in their entirety, etc. Fragmentation and the resulting nondeterminism of the allocator's behavior introduce further noise into our attacker's observations.

In Chrome, the attacker can also measure the footprint of the networking process alone. This does not enable the attack based on individual object sizes because it processes Web pages as blocks of bytes, without distinguishing individual objects. Furthermore, the networking process re-uses buffers through which it streams data to other processes, thus their sizes are useless for identifying individual objects.

V. Experimental setup

We report experimental results for Chrome, Firefox, and the default Android browser, but the principles of our attack are applicable to any browser that exposes its memory usage to other system users. Android experiments are representative of the smartphone setting, where a malicious application may spy on the phone owner's Web browsing. Chrome and Firefox experiments are representative of a shared-workstation setting, where one user may spy on another.

Chrome experiments were performed with a Chrome 13.0.782.220 browser running on an Acer laptop with an Intel Core Duo 2 GHz processor, 2 GB of memory, and Linux Ubuntu 10.04 operating system. By default, Chrome uses the process-per-site-instance model and forks a separate rendering process for each instance of a site visited by the user.[4] A site is identified by its protocol, domain, and port. The renderer process is responsible for parsing and executing JavaScript, building DOM trees, rendering images, etc., and serves as the target in our experiments.

In addition to the renderer, Chrome also forks one process per each type of plugins that are needed to display the site instance correctly. These can be distinguished from the renderer process because they have the "–type=plugin" option in their command line, as opposed to "–type=renderer".

Firefox experiments were performed with Firefox 3.6.23 on the same laptop. Unlike Chrome, Firefox is a monolithic browser: a single process is responsible for most of its functionality and serves as the target in our experiments. Firefox plugins run in separate "plugin-container" processes.

Whereas Chrome creates a new process for each instance of a site, Firefox uses one process for all sites visited for the user. The attack works against Chrome unconditionally, but against Firefox it works if the browser is fresh, i.e., it can identify the first page visited by the user after starting the browser, but accuracy drops off afterwards. We used fresh browser instances in the Firefox experiments. In Section VI, we describe variations of the attack that work even against a "dirty" Firefox browser.

To perform browsing measurements on a large scale, Android experiments were done with the default browser in Android 2.2 Froyo in the x86 simulator running in a VirtualBox VM. We verified that the results are the same for 3.1 Honeycomb in Google's ARM simulator, but the latter simulator is too slow for the scale of our experiments. We used a native attack process for better accuracy. Android provides developers with the Native Development Kit (NDK), thus a malicious process can easily masquerade as a game or another app that plausibly requires native functionality for performance reasons. Memory footprints of concurrent processes can also be measured by applications implemented in Android SDK, but with a lower measurement rate.

In contrast to desktops, an Android user is much likelier to open a fresh browser for each site visit. Most Android devices are charged based on their data usage. Keeping the browser open increases the amount of data transferred over the cellular network, thus the user has a monetary incentive to close the browser after viewing a page.

Furthermore, memory is a scarce resource in most Android devices. Once memory usage grows high, the kernel starts killing inactive processes, reclaiming their memory, and storing application state—for example, the URL of the current page in the browser. When the user switches to the browser, the kernel starts a fresh browser instance with the saved URL as the destination. Since the browser app has a big memory footprint, its chances of getting killed when left inactive and then started afresh are high.

Automating page visits. Gathering memory signatures of a large number of pages is significantly more time-consuming than a simple Web crawl. Most memory allocations in the browser happen when rendering and executing content, *after* it has been fetched. Therefore, it is necessary to wait until the browser has finished retrieving and displaying the page. JavaScript- and Flash-heavy pages can take a long time to load. Many of those located far from our measurement site took as long as 30-40 seconds to fetch and render fully.

Because load times are extremely variable, we instrumented browsers with custom scripts that automatically close the tab 5 seconds after an on_load event. The default Android browser does not support user scripts. We started its instances remotely, using Google's adb tool to execute the `am start -a android.intent.action.view -d <url>` command, and closed the page after 20 seconds.

Measuring browsers' memory footprints. First, the attack process finds out the `pid` of the browser process using `ps` or a similar utility. It then reads the `/proc/<pid>/statm` file in a loop. Each time it observes a change in data resident size, it records the new value. Because the initial

[4] http://www.chromium.org/developers/design-documents/process-models

allocations of most browsers do not depend on the contents of the webpage, measurements only include values above the browser-specific threshold: 8MB (2048 memory pages) for Chrome and 64MB (16348 memory pages) for Firefox.[5] For the Android browser, the attack process records all values.

To read `proc` faster, the attack process uses the `pread` system call which does not change the current offset, allowing repeated reads without rewinding or closing the file. To conceal its heavy CPU use, the attack process exploits the well-known flaw in the Linux scheduling algorithm [17]. Fig. 18 shows that this CPU cheat has little effect on accuracy of the attack. Its only purpose is to hide the attack process's activity from other users.

To scale our experiments up to 100,000 webpages, we used a different measurement method that runs the browser as a child of the attack process. The attack process uses `ptrace` to stop the browser at every system call and measures `drs` in `/proc/<pid>/statm`. This enables up to 6 concurrent measurement and browser processes on the same CPU without compromising the measurement rate. For Firefox, the attack process starts `firefox-bin` as its child. For Chrome, it only measures the renderer process and starts the browser with the "–renderer-cmd-prefix=tracer" option. Chrome then starts a copy of the tracer as its renderer and the tracer forks the original renderer as its child. We also used the "–allow-sandbox-debugging" option to allow `ptrace` to monitor the renderer as this is not allowed by default. In the rest of the paper, we refer to measurements collected using this method as FixSched measurements.

Obviously, only Attack measurements are available during the actual attack. The sole purpose of FixSched is to scale our experiments. Fig. 14 shows that the measurement method does not significantly affect recognizability. The vast majority of pages that are recognizable in the FixSched experiments remain recognizable under the actual attack.

Stability of footprints across different machines. The memprints used in our attack are based on measurements of the browser's data resident size. For a given HTML content, these values are OS- and browser-specific, but *machine-independent* (except for minor variations described in Section IV). We used Chrome and Firefox to load cached copies of 20 random webpages on 10 machines with different processors (from 2 to 8 cores) and amounts of memory (from 2 to 16 GB), all running Linux Ubuntu 10.04. The memprints for each page were identical across machines.

User-installed customizations such as plugins, add-ons, and extensions run either in the browser's memory context, or as separate processes. For example, video players, PDF readers, Chrome extensions, etc. run as separate processes and have no effect on the memory usage of browser processes monitored by our attack. Firefox toolbars, on the other hand, change the browser's memory footprint in predictable ways: typically, each toolbar is associated with a small number of possible offsets, and each measurement of the browser's data resident size is shifted by one of the offsets. We conjecture that the attacker can compute a database of offsets for common Firefox extensions and account for their effect when matching memprints against page signatures. Extensions that significantly change the content rendered by the browser for a given page—for example, block scripts or suppress ads—result in very different memprints and thus require a separate database of webpage signatures.

[5]Firefox generally consumes more memory than Chrome. Also, the attack on Chrome only measures the rendering process.

VI. Evaluating the basic attack

We now show that our attack can identify many webpages as they are being rendered by the browser. We are concerned about two types of errors. A *false negative* means that the attack fails to recognize the page visited by the victim. False negatives can be caused by pages that significantly change their content depending on the visitor's IP address, cookies, or some other factor that varies between the attacker's visits when computing the signature database and the victim's visits. In this case, memprints observed during the attack will not match the pre-computed signatures. A *false positive* means that the victim visits page A, but the attack mistakenly "recognizes" the resulting memprint as another page B.

First, we show that there exists a subset of *distinguishable* webpages. For each such page, browser memprints are similar across visits to this page, but dissimilar to visits to any other page (including pages outside the distinguishable subset). The matching threshold of Algorithm 1 can thus be set so that any match to the signature of a distinguishable page is correct and there are no false positives.

Second, we measure the true positive rate for distinguishable pages, i.e., how often a visit to each page produces a memprint that matches that page's signature.

Third, we measure how accuracy of inference is affected by the measurement rate and concurrent workload, and describe variations of the attack.

We say that a memprint m and a webpage signature p *match* if their similarity (see Section III) is above a certain threshold (see Algorithm 1). A match is correct if m was indeed a visit to p, false otherwise. *Distinguishability* of a page is the difference between the (probabilistically) worst correct match of any memprint to this page's signature and the best false match. Positive distinguishability implies low false positive rate. *Recognizability* of a page is the true positive rate, i.e., the percentage of visits to this page whose memprints are matched correctly.

Measuring distinguishability. In our experiments, we measure distinguishability with respect to fixed ambiguity sets. Intuitively, a page is distinguishable if a visit to this page is unlikely to be confused with a visit to any page from the ambiguity set. We cannot rule out that a page may be confused with some page *not* from the ambiguity set. Our

reported false positive rates are thus subject to the "closed-world" assumption (i.e., the victim only visits pages in the ambiguity set). To ensure that our "closed world" is as big as possible given the constraints of our experimental setup, we use the front pages of Alexa top N websites as our ambiguity sets, regardless of whether they are themselves distinguishable or not. N varies between 1,000 and 100,000 depending on the experiment.[6]

For each visit to the page, we compute the similarity of the resulting memprint to the best-matching signature of that page. Let μ and σ be the mean and standard deviation of these values. We also compute the similarity of the page's memprint to its nearest neighbor in the ambiguity set. This neighbor may change from visit to visit: some visits may be most similar to page F, while other visits are most similar to page $F' \neq F$. Let μ_{false} and σ_{false} be the mean and standard deviation of the nearest-neighbor similarity values. We define distinguishability as $(\mu - \sigma) - (\mu_{false} + \sigma_{false})$, i.e., it is the probabilistically worst difference between a true and false positive. Because absolute distinguishability varies between pages, we normalize it by dividing by $(Max - Min)$, where Max is the maximum distinguishability across all pages and Min is the absolute value of the smallest distinguishability (the latter is always negative). Distinguishability is thus a conservative overestimate of the difference between the memprint of a page and the likeliest false positive in the ambiguity set (cf. *eccentricity* [10]).

A page is *distinguishable* if it has positive distinguishability. Such a page is very unlikely to be mistaken for its nearest neighbor. Therefore, when the matching algorithm recognizes this page, this is unlikely to be a false positive.

Distinguishability of popular webpages. The experiments in this section employ the Attack and FixSched measurement methods described in Section V. For each experiment, we used the front pages of websites from the Alexa top site list as our ambiguity set, and selected a smaller *target set* of pages at random from the same list.

To create browser-specific signature databases, we visited each page from the target set 5 times with every browser and recorded the resulting memprints. We then visited each page from the ambiguity set 3 times and computed all pairwise similarities between the memprints of target pages and those of ambiguity pages. These values were used to compute distinguishability of target pages as described above.

Figs. 6 through 8 show that, depending on the browser, between 30% and 50% of the front pages of Alexa top sites are distinguishable. Distinguishability is worse for Android than for desktop browsers due to higher non-determinism. Distinguishability appears to be higher in Attack experiments (Fig. 8) because they use a smaller ambiguity set. Since there are fewer potential false positives, the similarity

[6] 18% of Alexa top 100,000 websites were unreachable or did not load in our experiments. These sites were removed from our ambiguity sets.

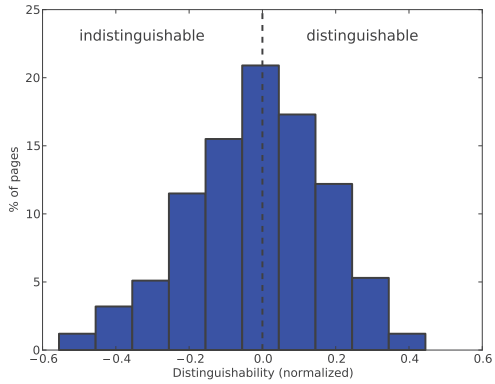

Figure 6. Chrome: Distinguishability of 1,000 random pages, 20,000-page ambiguity set (FixSched measurement). 48% of pages are distinguishable.

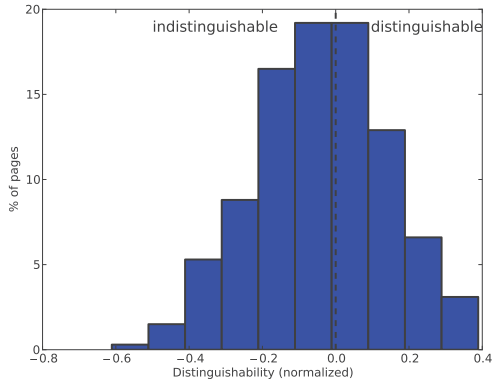

Figure 7. Chrome: Distinguishability of 1,000 random pages, 100,000-page ambiguity set (FixSched measurement). 43% of sites are distinguishable.

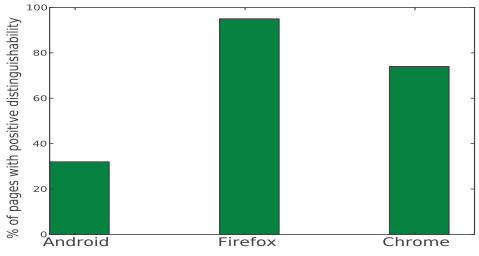

Figure 8. Percentage of pages with positive distinguishability, out of 100 random pages, 1,000-page ambiguity set (Attack measurement).

gap between correct matches and the "best" false positive is higher than in FixSched experiments.

Fig. 9 plots cross-site similarity for 100 random webpages. Distinguishable pages are dark along the diagonal (repeated visits to the page produce similar memprints) and light elsewhere (they are *not* similar to other pages).

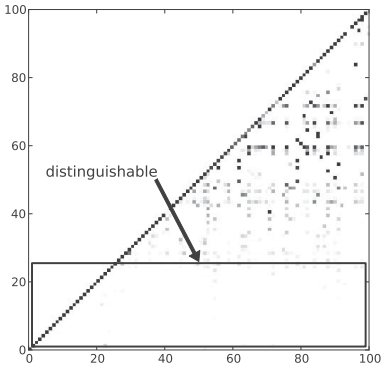

Figure 9. Chrome: Cross-page similarity for 100 random webpages (Attack measurement).

Image-heavy pages often have high distinguishability. For example, Fig. 10 shows how distinctive the memprints of visits to `perfectgirls.net` are (this is a hardcore porn site - beware!). They cannot be mistaken for the front page of any other Alexa top-1,000 site.

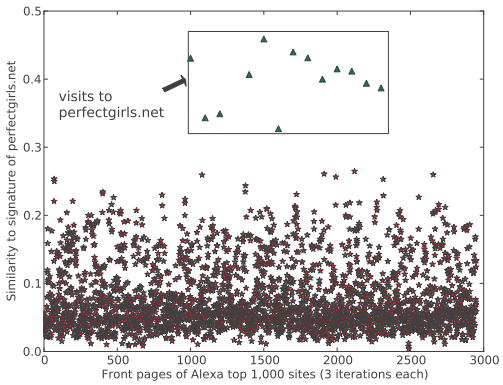

Figure 10. Chrome: Similarity between perfectgirls.net and the front pages of Alexa top 1,000 sites (3 iterations each).

On the other hand, pages based on the same template—for example, `google.com` and `google.de`, or Wordpress blogs—have low distinguishability. In Section VII, we describe how other side channels help disambiguate pages that use the same template. Others reasons for low distinguishability are animation, frequently changing advertisements, and dynamic, visitor-specific content variations. For example, the main `bbc.com` page has low distinguishability because embedded ads change on every visit and result in widely varying memory allocations in the browser.

Lowering the matching threshold of Algorithm 1 increases the false positive rate and decreases the false negative rate. Figs. 11 and 12 show, for the distinguishable pages, the tradeoff between the average recognition rate (percentage of visits correctly recognized by the attack process) and the false positive rate. Observe that even if the parameters of the matching algorithm are tuned to produce no false positives, recognition rate remains relatively high, demonstrating that many pages can be reliably recognized by their memprint without ever mistaking them for another page.

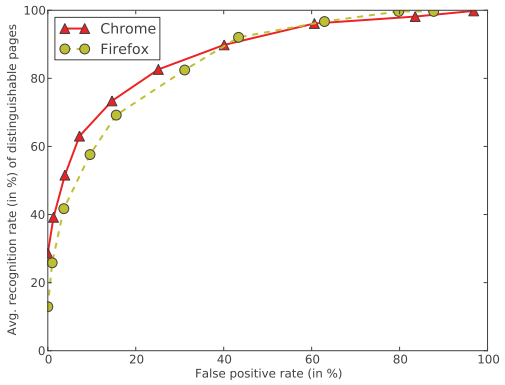

Figure 11. Chrome and Firefox: Average recognition rate vs. false positive rate for 1,000 pages, 10 visits each, with a 20,000-page (Chrome) and 10,000-page (Firefox) ambiguity set (FixSched measurement).

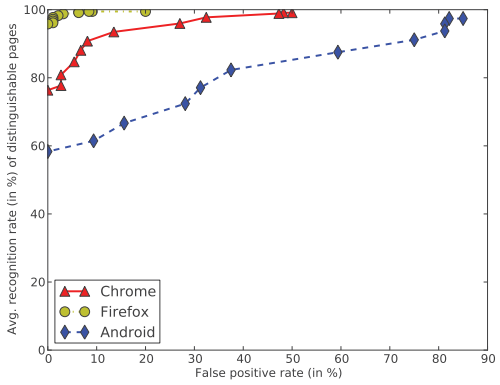

Figure 12. Chrome, Firefox, Android: Average recognition rate vs. false positive rate for 100 pages, 10 visits each, with a 1,000-page ambiguity set (Attack measurement).

Measuring recognizability. If a page is distinguishable, a match is unlikely to be a false positive, but not every visit to a distinguishable page produces a match. Recall that

recognizability of a page is the percentage of visits whose memprints are successfully matched by Algorithm 1 to any of this page's signatures.

To measure recognizability, we visit each distinguishable page 5 or 15 times (in FixSched and Attack experiments, respectively) and set the threshold of Algorithm 1 equal to the highest similarity between the signature of any target page and the memprint of any visit to an ambiguity page. This ensures that memprints of ambiguity pages do not match any signatures and thus Algorithm 1 cannot produce any false positives with respect to the ambiguity set.

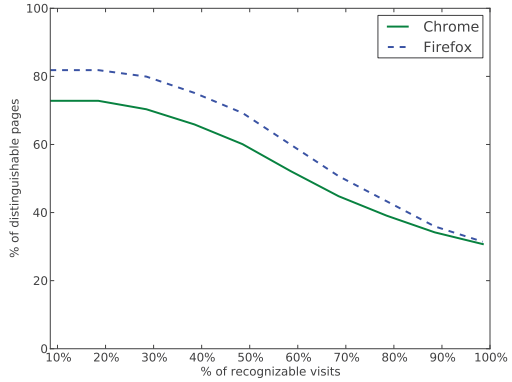

Figure 13. Chrome and Firefox: Recognizability of 1,000 random distinguishable pages (FixSched measurement). No false positives.

Fig. 13 shows the results for Chrome and Firefox. As many as 75% of the distinguishable pages have recognizability above 20% (i.e., at least 1 out of 5 visits produces a recognizable memprint). For a quarter of the pages, *every visit produces a recognizable memprint*, with no false positives. Figs. 11 and 12 show that if a non-negligible false positive rate is acceptable, the recognition rate is much higher.

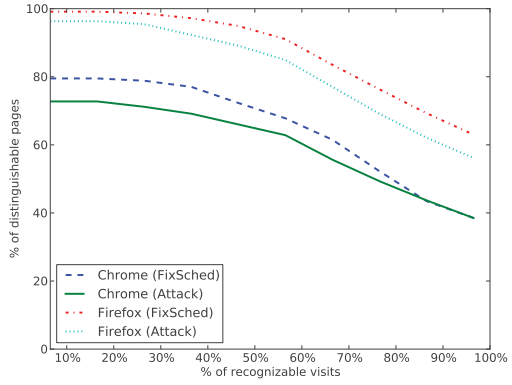

Figure 14. Chrome and Firefox: Recognizability of 100 random distinguishable pages (Attack and FixSched measurements). No false positives.

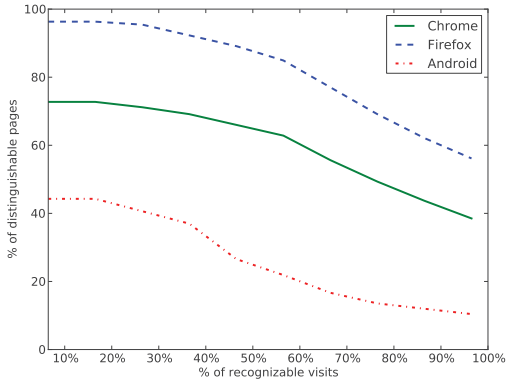

Figure 15. Chrome, Firefox, Android: Recognizability of 100 random distinguishable pages (Attack measurement). No false positives.

Effect of the measurement method. Fig. 14 shows FixSched and Attack results for Firefox and Chrome on the same chart, demonstrating that FixSched experiments accurately represent recognizability under the actual attack (we are using FixSched solely for scalability). Not only are the distributions similar, but the same pages that have high recognizability under FixSched overwhelmingly have it under Attack (this is not shown on the chart). Fig. 15 shows Attack results for all three browsers.

Recognizability appears to be higher in Attack experiments because they use a smaller ambiguity set. The smaller the set, the lower the maximum similarity between any target page and the "best" false positive from the ambiguity set, the lower the threshold that must be used by the matching algorithm to avoid false positives. Therefore, some memprints that match the page signature in Attack experiments no longer match in FixSched experiments, which use bigger ambiguity sets and thus higher matching thresholds.

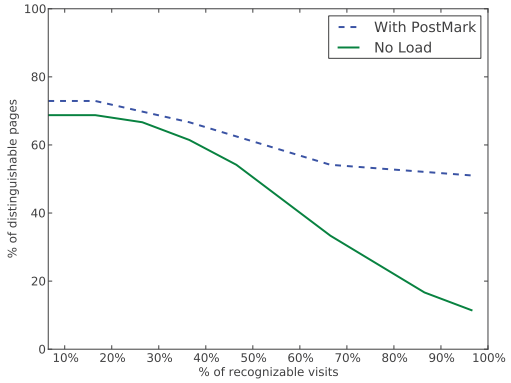

Figure 16. Chrome: Recognizability of 20 random distinguishable pages, no significant load vs. PostMark running concurrently (Attack measurement). No false positives.

Effects of concurrent workload. Other processes running concurrently with the victim and the attacker do not reduce the efficacy of the attack. They slow down both the victim's memory allocations and the attacker's measurements, but the patterns measured by the attacker remain roughly the same.

Fig. 16 shows the results for our attack in the presence of a concurrent, CPU- and I/O-intensive workload on the host. In this experiment, the victim and the attacker run in parallel with a PostMark benchmark, which simulates an email, network news, and e-commerce client. PostMark is executing in a loop with $100,000$ file-system transactions in each iteration, causing 36% CPU load. In this case, the concurrent workload slows the browser process more than the attack process, enabling the latter to obtain more measurements. The resulting memprint matches the signature of the page better than in the absence of the workload.

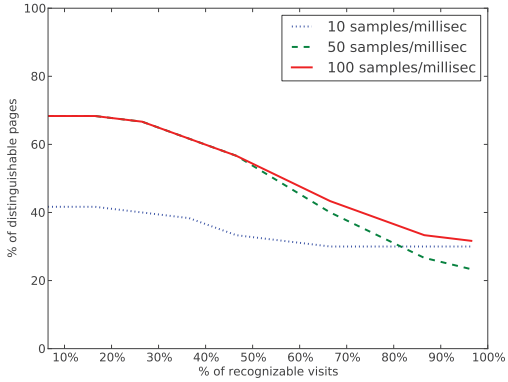

Figure 17. Chrome: Recognizability of 20 random distinguishable pages with different measurement rates (Attack measurement). No false positives.

Effects of the measurement rate. Fig. 17 shows that even if the attack process decreases the rate at which it measures the memory footprint of the browser process and thus produces smaller memprints, recognition rates remain high for distinguishable pages.

Fig. 18 shows that the Linux CPU cheat does not significantly affect accuracy of the attack. Our attack process only uses this cheat to decrease its reported CPU usage and thus hide its measurement activity.

Variations of the basic attack. Algorithm 1 has many variations. For example, matching can ignore total footprint sizes and only consider the sequence of deltas, or focus on changes caused by allocating and de-allocating large images.

These variations work well for some pages and browsers. For example, to process a large image, Firefox allocates a big buffer, uncompresses the image into this buffer, then frees the buffer after rendering the image. If the buffer is bigger than 4MB, de-allocation results in immediately returning memory to the OS. The corresponding change

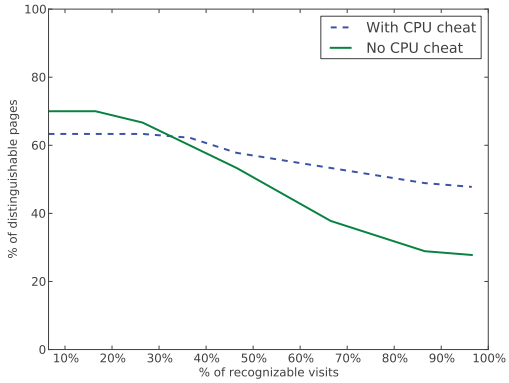

Figure 18. Chrome: Recognizability of 20 random distinguishable pages with and without CPU cheat (Attack measurement). No false positives.

Table II
FIREFOX: AMOUNT OF MEMORY ($>= 4$ MB) FREED IMMEDIATELY AFTER LOADING DIFFERENT WEBPAGES.

Webpage	Sequence of de-allocations (in KB)
`playboy.com` (adult)	9216, 4096, 14336, 4096
`exbii.com` (adult)	8196, 8196, 8196, 8196
`journaldesfemmes.com`	8196, 8196, 8196, 8196, 8196, 10240, 5120
`cnn.com`	5120

in the `drs` field of `/proc/<pid>/statm` is observable by the attacker. Therefore, the variation of the matching algorithm that correlates deltas in the footprint with images tends to do well at recognizing pages with many big, high-resolution images (see Table II). This category includes the front pages of many adult sites such as `playboy.com`.

VII. ADVANCED ATTACKS

In Section VI, we showed how to use the dynamics of memory usage to recognize pages browsed by the victim. We now show how to combine them with secondary side channels to infer more private information. All attacks in this section work against all tested browsers, but for clarity, each figure only shows the results for a particular browser.

Inferring the state of Web sessions. Most changes in the browser's footprint occur while a page is being loaded and rendered. The footprint then remains stable until the user requests a new page. For example, Fig. 19 shows changes in the footprint as the user enters a search query, views the results, and clicks on a link. The increments and the size of the stable footprints vary depending on the page within the site. These differences can leak sensitive information.

Figs. 20 and 21 show that a successful login into, respectively, Google Health and OkCupid (a dating site) results in a significant increase in the footprint since profile pages

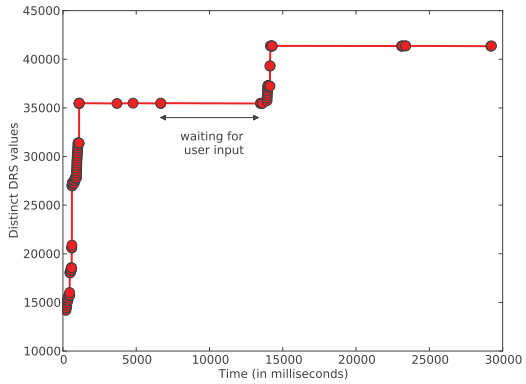

Figure 19. Evolution of the Firefox memory footprint during a Google search session.

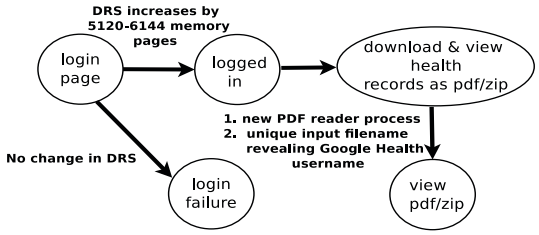

Figure 20. Google Health state transitions in Firefox. Changes visible to the attacker are shown in bold.

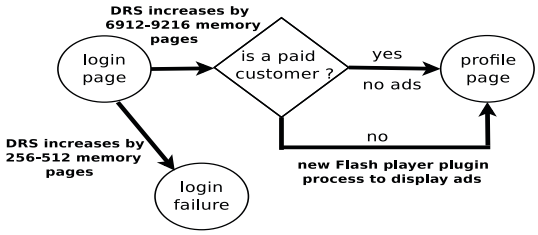

Figure 21. OkCupid state transitions in Firefox. Changes visible to the attacker are shown in bold.

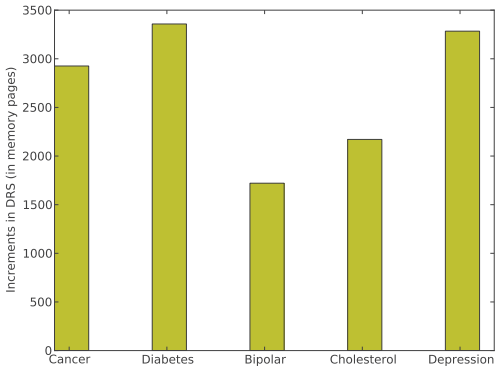

Figure 22. Chrome footprint increments for transitions from the webmd.com main page to different illness-related pages.

tend to use more JavaScript than login pages. The attacker can thus infer whether the victim is a member of the site. Paid users of OkCupid do not see ads, while free users see Flash advertisements. The Flash plugin runs as a separate process, allowing the attacker to infer whether the victim is a paid member of the dating site or not.

When the victim views medical records from Google Health, a new PDF reader process appears. The file name in its command-line arguments (available via `proc` on Linux, but not Android) reveals the victim's username.

Fig. 22 shows that the attacker can infer the medical condition the victim is interested in by measuring the increase in the browser's memory footprint after the victim has clicked on a link from `webmd.com`.

Disambiguating similar memprints. Secondary side channels can help disambiguate pages that otherwise have similar memprints. For example, `google.com` and `google.de` use the same template, thus their memprints are very similar. Fig. 23 shows how they can be be distinguished by their *duration* if the browser is located in the US.

Once the attack process is running concurrently with the browser, it can directly observe which shared libraries are used by the browser. Fig. 24 shows that if the matching algorithm considers changes in the size of shared memory in addition to changes in the main footprint (DRS), the recognition rate improves for moderately stable pages.

Other useful side channels include timing of DNS resolutions (they reveal whether a particular domain is in the DNS cache), command-line arguments of various processes, etc. In section VIII, we show how to use CPU scheduling statistics—conveniently revealed by `proc`—to disambiguate pages with similar memprints.

VIII. Exploiting CPU scheduling statistics

Zhang and Wang showed that the ESP (stack pointer) value revealed by `proc` leaks information about keystroke timings [20]. Their attack is unlikely to work on Android because Dalvik-based Android applications—such as the MMS app we attack below—are highly nondeterministic.

To illustrate our thesis that any accounting information about a process can leak its secrets, we show how to use scheduling statistics for keystroke sniffing. Unlike ESP, these statistics are used by `top` and thus available in all versions of Unix. Zhang and Wang mention the possibility that the number of interrupts can be used for keystroke sniffing but do not describe a concrete attack. The interrupt count is global, not process-specific, thus the feasibility of this attack remains open. Scheduling statistics, on the other hand, provide a much more robust, process-specific channel.

Linux. In Linux, the number of context switches made by a process can be found in `/proc/<pid>/schedstat`,

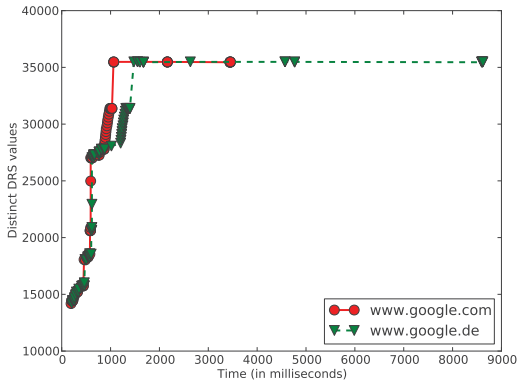

Figure 23. Evolution of the Firefox memory footprint when loading google.com and google.de (US-based browser).

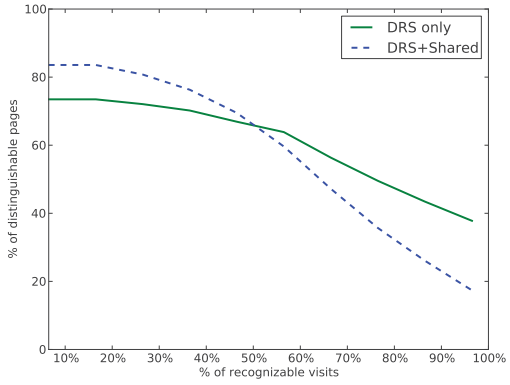

Figure 24. Chrome: Recognition rate when considering shared memory in addition to DRS (100 pages, 1,000-page ambiguity set). No false positives.

which has three fields: time spent on the CPU, time spent waiting, and timeslices run on the CPU. It turns out that the timeslice counter leaks precise information about the keystrokes the process is taking as input.

In programs like bash and ssh, user input is much slower than the program itself. Therefore, they get off the run queue whenever they wait for a keystroke and only get back once the user presses a key. Every time the user presses a key, bash and ssh execute for 1 and 2 timeslices, respectively, before going back to wait for the next keystroke.

The attacker can thus continuously monitor `schedstat` to find out when keys are pressed and calculate precise inter-keystroke timings. Table III compares these measurements with an actual keylogger as the first author is typing his name. These timings can be used to reconstruct the user's input using an appropriate natural language model [20], but we leave this as a topic for future research.

Android. In Android, `/proc/<pid>/schedstat` is not available, but the number of voluntary and involuntary

Table III
INTER-KEYSTROKE TIMINGS IN MILLISECONDS: KEYLOGGER VS. SCHEDSTAT MEASUREMENTS (LINUX).

Timings	bash		ssh	
	True	Measured	True	Measured
1	127	128	159	160
2	191	191	111	111
3	88	87	184	184
4	159	161	199	198
5	111	112	119	119

context switches made by a process can be read from `/proc/<pid>/status`. While the process waits for a call to retrieve the user's keystroke, the kernel removes it from the run queue. This results in a context switch and enables the attacker to measure inter-keystroke timings.

We experimented with two Android applications: an MMS app for sending text and multimedia messages and a bash shell. For bash, we monitored the shell process's context-switch counts in `/proc/<pid>/status`. Every time the user presses a key, the count increases by 1. Unlike bash, the MMS app has an input loop, so the context-switch count is increasing even while it is waiting for keystrokes. This is typical of many GUI applications because in Android devices, a keystroke is usually processed by an input method editor (IME) before it is passed to the application. In our case, LIME IME is handling the key-press events, thus the attack process monitors the number of context switches in a LIME process named `net.toload.main`. For every key press, the LIME process typically makes $3-5$ context switches, but as Fig. 25 shows, the intermediate context-switch delays are very small compared to the delay caused by waiting for keystroke inputs because user input is much slower than computation. Table IV shows that this enables the attacker to precisely measure inter-keystroke timings.

This attack is not specific to the MMS app. All keystrokes handled by the LIME process are potentially vulnerable.

Table IV
INTER-KEYSTROKE TIMINGS IN MILLISECONDS: KEYLOGGER VS. STATUS MEASUREMENTS (ANDROID).

Timings	MMS app		bash	
	True	Measured	True	Measured
1	445	449	256	256
2	399	399	320	320
3	176	176	165	175
4	236	240	393	391
5	175	173	255	256

Combining side channels. We now show how keystroke information inferred from the CPU scheduling statistics can enhance our basic attack based on memory usage dynamics.

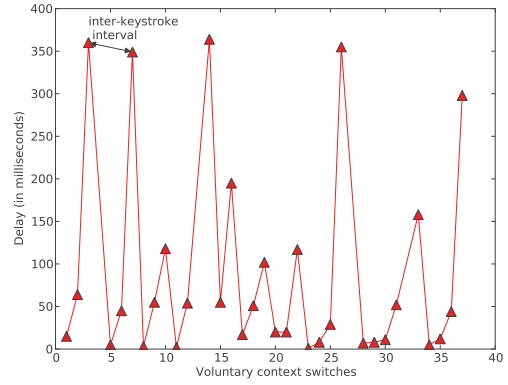

Figure 25. Context-switch delays (LIME in Android).

Consider two sites whose front pages have relatively similar memprints: `rediff.com` (an Indian Web portal) and `plentyoffish.com` (a popular dating site). Their URLs require a different number of keystrokes (Table V), enabling the attack process to disambiguate them if the user types the URL directly into the Android browser.

Table V
INTER-KEYSTROKE TIMINGS IN MILLISECONDS: KEYLOGGER VS. `STATUS` MEASUREMENTS (ANDROID BROWSER).

Timings	`rediff`		`plentyoffish`	
	True	Measured	True	Measured
1	386	393	400	400
2	405	400	690	689
3	325	327	768	768
4	315	313	943	943
5	329	331	803	803
6			220	220
7			298	294
8			518	519
9			88	89
10			927	927
11			199	201

IX. DEFENSES

Changing the OS. Attacks described in this paper are not specific to `proc`. Most operating systems reveal fine-grained accounting statistics about program execution. As long as temporal changes in these statistics are correlated with the program's secrets, they can be used to infer the latter.

Even the blunt solution of removing the `proc` filesystem entirely may not eliminate this class of attacks. For example, `proc` is deprecated in FreeBSD, but memory usage information is still available via utilities like `ps` and `top`. An unprivileged attack process can run them directly or use the same mechanism they use (e.g., call `kvm_getprocs`) to measure memory usage of other users' processes. For example, the attacker can execute `ps -l -p <pid>`, which returns virtual set size (VSZ), equal to DRS + shared + code size. Code size is typically constant, thus this immediately leaks the information needed for the attack.

Furthermore, changes to the `proc` filesystem may break existing applications. Out of 30 applications from the Android standard installation, 24 do not access `proc`, while 6 use `/proc/self/cmdline` to obtain their command-line parameters. Therefore, removing memory usage information from `proc` on Android is unlikely to affect existing applications. If OS designers cooperate, this may be feasible defense—at the cost of breaking existing utilities. Information about the program's memory usage will still be available through indirect channels, such as the size of swap files, but these channels are more coarse-grained and likely to reduce the efficacy of the attack.

Some kernel-hardening patches[7] remove the ability to view processes outside of `chroot` even if `/proc` is mounted, remove addresses from `/proc/pid/stat` (this prevents the attack from [20]), or even restrict `proc` to show only the user's own processes (this breaks existing utilities). Even if the attack process cannot view information about other processes via `proc`, it can still exploit side channels like the loading time of shared libraries and `pid` ordering, as well as aggregate, system-wide information such as total free memory, total context switches, etc. These channels are significantly coarser and noisier, but may still be correlated with the secrets of the target application.

Changing the application. Without changes to the OS, an application cannot prevent a concurrent attacker from measuring its memory footprint and must modify its behavior so that its memory usage is not correlated with its secrets. Network-level defenses, such as browsing through proxies, Tor, or SSL/TLS, do not provide much protection, nor does browsing in "private" or "incognito" mode.

To reduce the correlation between the application's behavior and OS-visible changes in its footprint, the allocator should manage the application's memory internally, without exposing allocations and de-allocations to the OS. Implementing programs in managed languages largely solves the problem. We have not been able to stage our attack against the Lobo browser implemented in Java.

Modern browser architectures such as Chrome and OP create a new process for each tab or site instance, allowing the attacker to easily match his footprint measurements against the database of pre-computed signatures. As explained in Section V, mobile operating systems frequently restart the browser process, which again benefits the attacker. In contrast, monolithic desktop browsers such as Firefox reuse the same process for multiple sites. Matching a page visited in a fresh Firefox is the same as for Chrome, but

[7] http://grsecurity.net/features.php

matching subsequently visited pages requires a different matching algorithm that only looks at footprint deltas (i.e., how much it increased or decreased) and ignores the absolute size. Recognition remains feasible, but its accuracy drops off because pure-delta sequences are significantly noisier.

In Section VI, we described a variation of the matching algorithm that considers *reductions* in the footprint caused by the browser de-allocating large images after rendering them and returning the memory to the OS via `unmap`. This variation works well even if the Firefox process is "dirty," regardless of the previously rendered pages.

Therefore, even returning to monolithic browser architectures where the same rendering process is used for all pages may not completely foil the attack.

X. Conclusion

Many modern systems leverage OS user and process abstractions for security purposes—for example, to prevent Android applications from snooping on each other. This has unintended consequences because the OS reveals certain accounting information about every process, including the size of its memory footprint and CPU scheduling statistics.

It has been observed in other contexts [4] that even when a single piece of information is harmless, how it *changes* over time can leak secrets. In this paper, we demonstrated that the pattern of changes in the browser's memory footprint uniquely identifies thousands of webpages, allowing the attacker (e.g., a malicious Android application or another user on a shared workstation) to infer which pages the victim is browsing, her relationship with websites, and other private information. CPU scheduling statistics can be used for keystroke sniffing and to improve accuracy of the memory-footprint attack.

These attacks are a symptom of a bigger problem. Privacy risks of system isolation mechanisms are poorly understood and a worthy topic of further research.

Acknowledgments. We are grateful to Arvind Narayanan for insightful comments on a draft of this paper, to Chad Brubaker for pointing out that the attack is foiled by browser add-ons that block ads and/or scripts, and to our shepherd Simha Sethumadhavan for helping us improve and clarify the presentation. The research described in this paper was partially supported by the NSF grants CNS-0746888 and CNS-0905602, Google research award, and the MURI program under AFOSR Grant No. FA9550-08-1-0352.

References

[1] D. Asonov and R. Agrawal. Keyboard acoustic emanations. In *S&P*, 2004.

[2] M. Backes, M. Dürmuth, and D. Unruh. Compromising reflections - or - how to read LCD monitors around the corner. In *S&P*, 2008.

[3] L. Cai and H. Chen. TouchLogger: Inferring keystrokes on touch screen from smartphone motion. In *HotSec*, 2011.

[4] J. Calandrino, A. Kilzer, A. Narayanan, E. Felten, and V. Shmatikov. "You might also like:" Privacy risks of collaborative filtering. In *S&P*, 2011.

[5] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in Web applications: A reality today, a challenge tomorrow. In *S&P*, 2010.

[6] G. Danezis. Traffic analysis of the HTTP protocol over TLS. http://research.microsoft.com/en-us/um/people/gdane/papers/TLSanon.pdf, 2010.

[7] J. Friedman. Tempest: A signal problem. *NSA Cryptologic Spectrum*, 1972.

[8] C. Grier, S. Tang, and S. King. Secure Web browsing with the OP Web browser. In *S&P*, 2008.

[9] M. Hogye, C. Hughes, J. Sarfaty, and J. Wolf. Analysis of the feasibility of keystroke timing attacks over SSH connections. Technical Report CS588, University of Virginia, December 2001.

[10] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *S&P*, 2008.

[11] M. Padlipsky, D. Snow, and D. Karger. Limitations of end-to-end encryption in secure computer networks. Technical Report ESD-TR-78-158, MITRE Corporation, August 1978.

[12] G. Peluso. Design your application to manage performance data logs using the PDH library. http://www.microsoft.com/msj/1299/pdh/pdh.aspx, 1999.

[13] S. Saponas, J. Lester, C. Hartung, S. Agarwal, and T. Kohno. Devices that tell on you: Privacy trends in consumer ubiquitous computing. In *USENIX Security*, 2007.

[14] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X.Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, 2011.

[15] D. Song, D. Wagner, and X. Tian. Timing analysis of keystrokes and timing attacks on SSH. In *USENIX Security*, 2001.

[16] Q. Sun, D. Simon, Y-M. Wang, W. Russell, V. Padmanabhan, and L. Qiu. Statistical identification of encrypted Web browsing traffic. In *S&P*, 2002.

[17] D. Tsafrir, Y. Etsion, and D. Feitelson. Secretly monopolizing the CPU without superuser privileges. In *USENIX Security*, 2007.

[18] M. Vuagnoux and S. Pasini. Compromising electromagnetic emanations of wired and wireless keyboards. In *USENIX Security*, 2009.

[19] C. Wright, L. Ballard, S. Coull, F. Monrose, and G. Masson. Spot me if you can: Uncovering spoken phrases in encrypted VOIP conversations. In *S&P*, 2008.

[20] K. Zhang and X. Wang. Peeping Tom in the neighborhood: Keystroke eavesdropping on multi-user systems. In *USENIX Security*, 2009.