Layered Binary Templating

Martin Schwarzl[1], Erik Kraft[1], and Daniel Gruss[1]

[1]Graz University of Technology, Austria

Abstract. We present a new generic cache template attack technique, *LBTA*, layered binary templating attacks. *LBTA* uses multiple coarser-grained side channels to speed up cache-line granularity templating, ranging from 64 B to 2 MB in practice and in theory beyond. We discover first-come-first-serve data placement and data deduplication during compilation and linking as novel security issues that introduce side-channel-friendly binary layouts. We exploit this in inter-keystroke timing attacks and, depending on the target, even full keylogging attacks[1], e.g., on Chrome, Signal, Threema, Discord, and the passky password manager, indicating that all Chromium-based apps are affected.

1 Introduction

Techniques like Flush+Reload [77] advanced cache attacks from cryptographic [4] to non-cryptographic applications operating on secret data have been the research focus, e.g., breaking ASLR (address-space layout randomization) [35,28], attacking secure enclaves [27,6,44,19], spying on websites and user input [41,69], and covert channels [76,77,42,53]. In particular, user input, especially keystrokes, has become a popular attack target for inter-keystroke timing attacks [61,50,56]. Gruss et al. [32] showed that libraries leak more information than just inter-keystroke timings, e.g., distinguishing groups of keys.

Compilers and linkers [36] can facilitate or even introduce side-channel leakage [47,60], invisible on the source level, through optimizations targeting runtime, memory footprint, and binary size. Similarly, JIT compilation can also introduce timing side channels [7]. These side channels are typically invisible in the source code and often remain undetected. Numerous works explored the automatic identification of cache side-channel leakage, albeit with a focus on cryptography and the goal of making code constant-time [13,9]. However, for general-purpose applications, e.g., browsers, it is not feasible to linearize the entire instruction stream to constant-time code, especially for different user inputs that trigger vastly different program behavior. Cache templating takes a practical approach by scanning for leakage on real systems, providing a leakage template either to a defender (to close the side channel) or an unprivileged attacker who maps binaries as shared memory and infers events from side-channel

[1] Demo: The user first announces via Signal messenger to send money to a friend, then switches to Chrome to visit a banking website and enters the credentials there. All keystrokes are correctly leaked. `https://streamable.com/dgnuwk`.

activity. The templating itself runs on an attacker-controlled system with full privileges, a binary is mapped into the address space of the templating process to profile which memory locations show side-channel activity upon specific events. Since cache templating works with binary offsets it is entirely unaffected by mechanisms such as ASLR. While the fine cache-line granularity is beneficial in the attack phase, it also leads to extremely high templating runtimes. For instance, templating the binary, shared libraries, and memory-mapped files used by the Chrome browser (about 210 MB) with the published cache template attack tool [32] on our test system, would take 113.17 days. Unfortunately, this prohibits integration of cache leakage analysis into development workflows. Hence, we need to ask the following questions:

Which role does spatial granularity play for template attacks? Does a coarser granularity bear benefits in the templating phase?

In this paper, we answer both questions with *LBTA*, *Layered Binary Templating Attack*. *LBTA* introduces the previously unexplored dimension of *spatial granularity* into software-based templating attacks. *LBTA* combines the information of multiple side channels that provide information at different spatial granularity to accelerate the search for secret-dependent activity substantially. Our templating starts with the channel with the most coarse spatial granularity and, based on the activity, uses more fine-grained spatial granularity to detect the exact location (cache-line granularity 64 B).

Our evaluation of *LBTA* on state-of-the-art systems shows that a variety of hardware and software channels with different granularity are available. We focus in particular on a combination of a software channel, the page-cache side channel, with 4 kB granularity, and the cache side channel, with 64 B granularity. Page cache attacks are hardware-agnostic [30], resulting in cross-platform applicability, *i.e.*, our templater supports both Windows and Linux with the 4 kB page-cache side channel. We show that this two-layered approach already speeds up cache templating [32] by three orders of magnitude (*i.e.*, 1848x).

We evaluate *LBTA* on different software projects, including Chrome, Firefox, and LibreOffice Writer. The most significant finding is that **first-come-first-serve data placement** and **data deduplication during compilation and linking** during compilation and linking introduce side-channel-friendly binary layouts, with spatial distances of multiple 4 kB pages between key-dependent data accessed during a keystroke. Using *LBTA* [62], we find distinct leakage for all alphanumeric keys, allowing us to build a full unprivileged cache-based keylogger using Flush+Reload that leaks all keystrokes from Chromium-based applications involving password input fields, e.g., Chrome on banking websites, popular messengers including Signal, Threema, Discord, and password manager apps like passky. Based on our findings, we conclude that any app using the Chromium framework should be considered affected [23]. In addition, we demonstrate that where full keylogging is not possible, *LBTA* still finds enough leakage for inter-keystroke timing attacks [61,50,79,20,32], e.g., on Firefox and LibreOffice Writer.

We confirm that the Linux `preadv2` syscall can be used instead of the now mitigated `mincore` syscall [30] for page cache attacks [37] albeit with a lower temporal resolution of about 2 seconds. Since system-level defenses like ASLR have no effect on our attack, we provide a systematic discussion of the possible mitigation vectors specific to *LBTA*.

Contributions. The main contributions of this work are:

1. We introduce a new dimension, side-channel granularity, into cache template attacks and use it to speed up the templating by three orders of magnitude.
2. We show that the leakage discovered by *LBTA* can be exploited in hardware (*i.e.*, Flush+Reload) and software attacks (*i.e.*, via the page cache).
3. We discover first-come-first-serve data placement and data deduplication generate amplify and introduce side-channel leakage, invisible on the source level.
4. We present inter-keystroke timing and, depending on the target, full keylogging attacks, e.g., Chrome, Signal, and the passky password manager.

Responsible Disclosure. We responsibly disclosed our findings to the Chromium team. The underlying issue is tracked under CVE-2022-2612. anonymized (6.5, medium severity) and was patched in the M104 release in August 2022.

Outline. In Section 2, we provide the background. In Section 3, we explain the *LBTA* building blocks. In Section 4, we describe first-come-first-serve data placement and data deduplication. In Section 5, we evaluate *LBTA* on different targets. In Section 6, we discuss mitigations, and we conclude in Section 7.

2 Background

In this section, we provide background on hard- and software cache attacks, side-channel discovery, and compiler- and linker-introduced side channels.

Shared Memory Operating systems (OSs) apply various optimizations to reduce the system's general memory footprint. One such optimization is shared memory, where the OS actively tries to remove duplicate data mappings. An example would be shared libraries, such as the `glibc`, used in many programs, and thus, can be shared between processes. Moreover, with the `mmap` respectively `LoadLibrary` functions, a user program can request shared memory from the OS by mapping the library as `read-only` memory. Another optimization to reduce the memory footprint commonly used for virtual machines is memory deduplication on a page-wise level. The OS deduplicates pages with identical content and maps the deduplicated page in a copy-on-write semantic.

Deduplication The concept of deduplication is generic and can be applied in the context of various memory systems to save memory. For storage systems, one example is cloud storage systems that deduplicate files to minimize the amount of storage required [34,38]. For main memory, there are multiple mechanisms: Copy-on-write avoids duplicating memory during process creation, the OS's page cache [30] avoids duplicating memory pages from the disk, and the OS also avoid duplicating the zero page when zeroed memory is requested. However, the most prominent example is data-based page deduplication [63].

With data-based page deduplication, the OS or hypervisor scans the main memory page-wise and identifies identical pages, e.g., using hashes or byte-wise data comparison, deduplicating them. In all above types of deduplication, attempting to modify the deduplicated memory triggers a 'copy-on-write' operation which is known to introduce side-channel leakage, e.g., for file deduplication [34,2], page deduplication [63], from JavaScript [29,17] and even remotely [58].

While all above types of deduplication target memory, there is also deduplication in other contexts. In this paper, we focus on a different type of deduplication that has little to do with the above or memory systems in general. We instead focus on deduplication during compilation and linking. The goal of deduplication here is similar though, *i.e.*, reducing memory usage, and improving runtime performance due to reduced memory or cache utilization. However, the security implications of deduplication during compilation and linking are unknown.

Cache Attacks Caches introduce exploitable timing differences between cached and uncached data. While the first cache attacks targeted cryptographic primitives [39,4], more recent ones target secure enclaves [27,6,44,19], monitor user interaction and keystrokes [41,56,69], and build stealthy and fast covert channels [76,42,53]. The Flush+Reload attack technique requires shared memory with the victim, e.g., shared libraries [77]. However, as Flush+Reload works on the attacker's own addresses pointing to the same physical shared memory, there is no need to know the victim's ASLR offsets, as file offsets are used instead.

Cache attacks were also demonstrated from JavaScript to spy on keystrokes and break memory randomization [46,41,28]. Most cache attacks focus on hardware caches with a 64 B cache line granularity. Cache attacks on the TLB instead have a spatial granularity of 4 kB, 2 MB, 1 GB, or 512 GB [31,66].

In particular, for SGX, so-called controlled channels have been demonstrated as powerful attack primitives [75,64,44] with high spatial and temporal resolution, as well as a very high accuracy. Controlled channels are side channels running with elevated privileges, e.g., kernel privileges, with a typical attack target being secure enclaves that are protected against regular kernel access.

There are also software caches, e.g., the page cache in Linux and Windows. Both Linux and Windows also offer functions to verify whether a specific virtual address is resident in memory or not, namely `mincore` and `QueryWorkingSetEx` respectively. Gruss et al. [30] demonstrated cache attacks on the page cache by either using these functions or by measuring timing differences. Despite hardening attempts on Linux and Windows, the Linux `preadv2` system call can still be used to mount cache attacks [37] in the same way as with the `mincore` syscall: Using the `RWF_NOWAIT` flag, an attacker can observe whether a page is resident in the page cache or not, yielding the same side-channel information as `mincore`. The results of the `preadv2` templating attacks can be found in Section 5.

Automated Discovery of Side Channel Attacks Templating attacks have been first shown and mentioned on cryptographic primitives running on physical devices [13,49,43]. Brumley and Haka [9] first described templating attacks on caches. Doychev et al. [22] presented a static analyzer that detects cache side-channel leakage in applications. Gruss et al. [32] showed that the usage of certain cache lines can be observed to mount powerful non-cryptographic attacks, namely on keystrokes. Lipp et al. [41] showed cache attacks and cache template attacks on ARM. Van Cleemput et al. [65] proposed using information gathered in the templating phase to detect and mitigate side channels. Wang used symbolic execution and constraint solvers to speed up cache templating of cryptographic software [70]. Schwarz et al. [54] demonstrated template attacks on JavaScript to enable host fingerprinting in browsers. Weiser et al. [72] and Wichelmann et al. [73] showed that Intel PIN tools can be used to automatically detect secret-dependent behavior in applications, especially in a cryptographic context. Wang et al. [69] presented a similar automated approach to detect keystrokes in graphics libraries. Carre et al. [10] mounted an automated approach for cache attacks driven by machine learning. With that approach, they were able to attack the `secp256k11` OpenSSL ECDSA implementation and extract 256 bits of the secret key. Brotzmann et al. [8] presented a symbolic execution framework to detect secret-dependent operations in cryptographic algorithms and database queries. Li et al. [40] demonstrated a neural network to perform power analysis attacks automatically. Yuan et al. [78] demonstrated that manifold learning can be used to detect and locate side-channel leakage in media software.

Compiler-introduced Side Channels While developers typically focus on the source code level and care is taken to not introduce side channels there, the compiler translates the source code to a binary, essentially a different language. However, this step can introduce program behavior that is not visible on the source level and introduces or amplifies side-channel leakage. Page [47] demonstrated that dynamic compilation in Java leads to power side-channel leakage in a side-channel-secured library. Simon et al. [60] showed that mainstream C compilers optimizations can break cryptographically secure code by introducing timing side channels. Brennan et al. [7] showed that timing side channels can be introduced by exploiting JIT compilation.

Due to this significant influence of compilers on side-channel leakage in binaries, they are also frequently used for new mitigation proposals against side-channel leakage [16,48,18,11,26,5].

3 *Layered Binary Templating Attack*

For large binaries, like the Chrome browser with multiple shared libraries (220 MB), templating with fine granularity like prior work [32,41,69], e.g., a cache line, becomes impractical. *LBTA* takes advantage of coarser granularity channels, which usually are considered a disadvantage for the attacker. In this section, we present the high-level view on *LBTA* and show how *LBTA* reduces the templating runtime by three orders of magnitude (*i.e.*, 1848x).

3.1 Threat Model

The **templating** (or profiling) runs on a fully attacker-controlled system with any privileges the attacker wants to use to facilitate the templating. This system

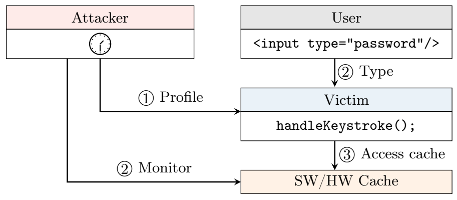


Fig. 1: Overview of the *LBTA*

is assumed to have the same side channels as the victim system, such as the page cache and CPU cache, and the same software versions as on the victim system were deployed, e.g., from package repositories. For this reason, the typical template attack threat model only restricts the attacker in the exploitation phase [32], which we follow in this work.

In the **exploitation phase**, the attacker runs an **unprivileged** attack program on the victim's system, possibly under a separate user account. Hence, we assume the victim application is started independently by the victim user, and cannot be started, stopped, or debugged by the attacker. This also excludes "preloading", which, e.g., on Wayland (the default Ubuntu display server), would allow monitoring all inputs to the application [3]. For non-Wayland systems, we assume that the attacker cannot use other keylogging techniques (e.g., on X11 [1]), or Windows (e.g., using the `getasynckeystate` API call [68]), e.g., due to system hardening or enforced security policies. Some of the applications we attack provide auto-fill features or are password managers. However, since we focus on the **keylogging scenario**, we assume that the victim user enters the password in these applications manually, *i.e.*, the user does not use an auto-filler or another password manager to unlock the password manager. Furthermore, many websites set the `autocomplete="off"` option for sensitive input fields, suppressing the in-built auto-fill and password management features.

3.2 High-Level Overview of the Templating Phase

Figure 1 illustrates the steps of *LBTA*. First, the attacker templates the library and creates templates of the cache usage for different keystrokes. After the templating phase, the attacker monitors the cache usage to infer inter-keystroke timings and, depending on the target, even distinguish key values.

3.3 Templating with Different Spatial Granularity

A novel aspect of *LBTA* is to utilize the spatial granularity of different side channels, forming a practical and generic multi-layered approach.

64 B Granularity. Previous cache template attacks [32] used cache-line granularity (64 B). One disadvantage of this approach is the runtime of the templating phase. When templating a single cache line with Flush+Reload, we observe an average runtime of 490 cycles ($n = 1000000, \sigma_{\overline{x}} = 20.35\%$). On a 4.0 GHz CPU, this would take 122.5 ns. The Chrome binary has a file size of about 210 MB leading to 2 949 120 addresses to template with Flush+Reload. This leads to a runtime of 0.36 s for templating every cache line once. However, Gruss et al. [32] describe that multiple rounds of Flush+Reload are required to get reliable cache templating results. Running an Intel 6700k CPU at 4.0 GHz with a Ubuntu 20.04, templating 1 MB of the Chrome browser (version 100.0.4896.60) with the provided implementation of Gruss et al. [32], we observe a runtime of 817.652 s for 1 MB and a total runtime of 1.98 days for the full binary, including shared libraries, of 210 MB. Moreover, this templating tool only reports whether a certain address was cached or not and does not match the cache hits with the entered keystrokes. To template, for instance, the 57 different common keys sequentially with the method by Gruss et al. [32], we would need an **impractical** total runtime of 113.17 days to obtain useful templates. We conclude that such an approach is not feasible for browser developers as the code base changes frequently, and releases sometimes occur on a monthly basis [14].

4 kB Granularity. Page cache attacks exploit the OS page cache, which works at a coarser granularity of 4 kB [30]. Page cache attacks have the advantage of working independent of the underlying hardware. To identify the exact memory locations causing leakage, they also resorted to templating. However, they did not combine this information with timing differences from hardware caches.

Our intuitive idea here is to combine the 4 kB-granularity side channel with the more fine-grained side channel into a two-layered approach. Hence, we **do not** template all cache lines on a 64 B granularity but instead, filter memory locations on a 4 kB granularity. Instead of 2 949 120 memory locations, we then only monitor 46 080 memory locations for the Chrome example, *i.e.*, a templating runtime speedup of at least 64. In addition, the templating phase on the 4 kB granularity level implicitly identifies locations exploitable via the page cache.

The templating phase runs on the attacker's own machine (cf. Section 3.1). Hence, we can use the page cache side channel or privileged channels, e.g., controlled-channel attacks [75] via page-table bits [64] during the templating, *i.e.*, we use the kernel's idle bit for tracking. For the full Chrome binary (cf. Section 3.5), this results in a runtime of only 1.47 hours for all 57 keystrokes.

2 MB Granularity. Each page-table layer provides `referenced` bits that are set by the hardware when a location in this region is accessed. The 2 MB-granularity side channel is also exposed via various side channels [31,66]. We observe the activity on 2 MB pages via the PMD paging structure and the `referenced` bit. We use PTEditor [55] to check and clear the referenced bit of the PMD, *i.e.*, a 2 MB page, with a runtime of 661.965 ns ($n = 1000000, \sigma_{\overline{x}} = 0.049\%$) per check. Hence, to template the 57 different common keys in Chrome with 20 repetitions per key, we estimate the total runtime of templating to be about 0.15 seconds.

3.4 Beyond Huge Pages

LBTA extends to arbitrarily coarser granularity channels.

1 GB, 512 GB and 256 TB Granularity. For the 1 GB granularity level and beyond, we experimentally validated that we can again use controlled-channel attacks [75,64], using the corresponding higher page-table layers. Following a similar approach as for the previous levels, we use PTEditor to template and clear the referenced bit for the single offset in a 1 GB (respectively 512 GB or 256 TB) range. The runtime for checking and clearing the referenced bit on these layers is the same as for the PMD (661.965 ns ($n = 1000000, \sigma_{\overline{x}} = 0.049\%$)). We emphasize that scanning layers that exceed the binary size, e.g., the 1 GB layer for a 180 MB binary, provides no additional information and does not reduce the search space, as the search will always proceed to the next smaller layer for the entire memory range then. Therefore, in the evaluation, we skip all layers that exceed the binary size. Still, these layers of *LBTA* may become relevant in the future with constantly growing binaries and libraries.[2]

3.5 Templating Phase Implementation

The high-level idea is that the templater tracks page usage and actively filters pages not related to keystrokes to reduce the search space of pages to template and, as a result, reduce the overall runtime of the templating. We implement our templater in Python and provide the code in our Github repository[3]. The templater takes as input the set of different keys, the PID or process names that should be monitored, and the number of samples per key.

Algorithm 1 summarizes the steps of the templating. First, the templater runs a warmup phase, where all keystrokes to template are entered once to load all related memory locations into RAM. Then the templater collects all the memory mapping information from all files from the target processes where activity has been found. These memory mappings include all shared libraries. The templater generates random key sequences based on the set of keys to template. For each key in the sequence, the templater iterates over all the memory locations on the current granularity level, and resets the access information, *i.e.*, resets the `referenced` or `idle` bit, or flushes the cache line depending on the side channel used. Based on the number of samples, the templater computes the hit ratio for each location. Subsequently, the templater repeats this step for all memory locations above a specific hit ratio with the next lower spatial granularity. With this search strategy, the templater continues down to the lowest level, where only regions are templated that showed activity on coarser granularity. On the lowest level, the templater determines a hit ratio for each single cache line.

Linux. On the upper layers, we start by obtaining the memory mappings for the target process. On Linux, we read these mappings from procfs (with root privileges in line with the threat model). We group the memory locations then according to the most coarse granularity we use in our templating. By using the `referenced`-bit side channel according to Algorithm 1, we narrow down the set of memory locations for the next layer.

[2] The Chrome binary had 100 MB in 2017 and 180 MB in 2022, an increase of 80 %.

[3] `https://github.com/IAIK/LayeredBinaryTemplating`

Algorithm 1: *LBTA* Templating Algorithm

```
Input: Set of keys $K$, target PIDs $P_n$, number of samples $N$
Output: hit ratio matrix of all memory mappings $H$
 1: Enter all keys in $K$ once // Warmup
 2: Collect all valid memory mappings of $P_n$
    (possibly from previous layer)
 3: for $i = 0; i < N; i++$ do
 4:   for each $k \in K$ do
 5:     Reset memory mappings (reset referenced/idle bits or flush)
 6:     Enter key $k$
 7:     Check state for all present memory mappings (via interface or timing)
 8:     Compute hit ratios for $k$ and update $H_k$
 9:   end for
10: end for
11: return $H$, and repeat algorithm for next layer
```

Windows. On Windows, we obtain a list of memory mappings using the `EnumProcessModules` PSAPI call, which lists all loaded libraries and executables, and `GetModuleInformation` for their actual sizes. Subsequently, we again use the `referenced`-bit channel to narrow down the set of memory locations using Algorithm 1. Subsequently, we continue with the next layer.

4 kB Page Granularity While for the upper layers, we read referenced bits using PTEditor [55], we use a more optimized approach for the page granularity.
Linux. Our page usage tracker iterates over active mappings, reads the idle bit for the corresponding physical page from `/sys/kernel/mm/page_idle/bitmap` and checks if the page was accessed. We start by resetting the bit so that the page usage tracker is ready. We use the Python3 `keyboard` library to inject keystrokes into an input field. After the templater performs the sequence of keystrokes, we check all pages that are still in the candidate list for activity. A 1 at the page offset in the bitmap means the page was not accessed. Conversely, if we observe a 0 at the page offset, we reset the page offset and add it to the set of correlated pages to track on the next layer. This approach is fully hardware-agnostic, implemented in software in the Linux kernel. After each iteration, we reset the state again by marking the pages as idle again and repeat the measurements.

In case of a sequential read access pattern, the Linux kernel speculatively prefetches further pages of the same file after a new page was added to page cache. This optimization is called `readahead` [33]. On Ubuntu 20.04 (kernel 5.4.0), the default read-ahead size is 128 kB and can be found in the sysfs (`/sys/block/<block_device>/bdi/read_ahead_kb`). For file-mappings the kernel performs a different optimization called `read-around`.[4] There, the kernel prefetches pages surrounded by the page causing the pagefault e.g., 16 pages before the page causing the pagefault and 15 pages after. To reduce triggering

[4] https://elixir.bootlin.com/linux/v5.4/source/mm/filemap.c#L2437

read-ahead prefetching for sequential reads, we use the madvise system call with the `MADV_RANDOM` flag to indicate a random read order.

Overlapping event (*i.e.*, keystroke) groups for the current candidate page and pages that might trigger the read-ahead of the current candidate page could cause false positives in the Linux case. In addition, if the number of read-ahead suppress pages is too small, false positives can occur. Our classifier tries to reduce the number of false positives by checking out the read-ahead/read-around windows and systematically rule out other keystrokes. Based on the results of the templater, the classifier actively accesses surrounding pages from the target page to suppress the read-ahead/read-around optimization. Note that the read-ahead and read-around windows might overlap for some keys. If the keys to template are not on the same 4 kB-page, we can still distinguish two keys by checking the first and last surrounded pages being accessed. The templater actively creates warnings in the templating phase in case the keys are still indistinguishable.

Windows. Windows uses a different page replacement strategy with working sets [52]. For the page usage tracker on Windows, we use the PSAPI call `QueryWorkingSetEx` and monitor the `Shared`, `ShareCount` and `Valid` flags. If the page is marked as valid and shared and the share count is larger than 1, we mark the page as used. For the reset, `EmptyWorkingSet` is used to remove the pages from all workings sets. This PSAPI call is only available for unprotected processes, which is no issue during the templating phase (cf. Section 3.1).

On Windows, we observed no prefetching optimization within working sets, *i.e.*, read-ahead does not affect hit ratio or spatial accuracy. Alternatively, the templating could also be performed via controlled side channels [75,25,59], tracing tools such as Intel PIN, machine learning [69,10] or architecturally monitoring the accesses of pages using PTEditor [55].

Classifier. On the 4 kB level, we collect the page-hit ratios for all events (*i.e.*, keys) and pages, showing the link between event and observable page hit. To distinguish 'no activity' from 'activity', we also template a dummy idle event [32] to measure which hit ratios are observed as a baseline. This idle event will not be linked to any page hit but rather should represent unrelated system activity the templater might pick up while profiling events. Our classifier links events or groups of events with single page hits to keep the number of observed pages as low as possible. This is a trade-off between search time and completeness of the search that can be chosen differently for any *LBTA* on any target application. Furthermore, a more sophisticated attack could increase detection accuracy from monitoring multiple pages or cache lines for each event. However, we decided to use the search-time-optimized path, as side-channel attacks typically cannot observe an arbitrary amount of memory addresses anyway, *i.e.*, we focus on a more practical set of leaking addresses.

The algorithm to find a suitable page hit for an event e works as follows:

1. We normalize page-hit ratio vector for event e by average page-hit ratio from other events (baseline activity). The resulting vector is the correlation strength between each page and the event e.
2. We select the page with the highest correlation strength as a candidate.

3. If the candidate is not above the **location-specific** baseline activity, our algorithm merges events (e.g., going from single keys to key groups) until it is. We continue with the resulting event group $E = \{e_1, \ldots, e_M\}$ in step 1.
4. Once a candidate is found that is above the **location-specific** baseline activity, the algorithm returns this page to subsequent templating layers.

While running, the classifier collects information on potential read-around prefetching pages to filter them out. After successful classification, the attacker has a mapping of pages to events (*i.e.*, key) and groups of events (*i.e.*, groups of keys).

4 Compiler- and Linker-introduced Spatial Distance

Before we evaluate *LBTA*, we present one significant leakage-facilitating effect that we discovered while applying *LBTA* on a variety of targets. This effect is particularly critical as it originates in compiler optimizations in LLVM/clang that are enabled by default and the available compiler flags that can control this behavior come with serious limitations. Compiler optimizations aim for a minimal program runtime, small memory footprint, and small binary size. Moreover, linker optimizations try to further optimize the binary in the linking stage. We primarily found two effects to facilitate cache side-channel leakage: One is the other is first-come-first-serve data placement in readonly sections, the other one is **data deduplication during compilation and linking**. While memory deduplication at runtime has been explored as a security risk already (cf. Section 2), data deduplication (e.g., of strings) during compilation is not widely known and its security implications are entirely unexplored. The security of constant-time implementations has been analyzed for side channels being introduced by compilers [47,65,60,7]. In this section, we show that deduplication in combination with first-come-first-serve population during compilation (cf. Section 4.2) and linking (cf. Section 4.3) can amplify this effect by increasing the chance that secret-dependently accessed victim data is placed in an attacker-facilitating way. Deduplication can also be performed at the linking stage. The spatial distance between secret-dependent accesses can be introduce by both compiler and linker optimizations. We present two scenarios that we also found in widely used real-world applications, where the placement of read-only data, especially strings, amplifies side-channel leakage dramatically.

4.1 First-Come-First-Serve Data Placement

Lookup tables are frequently used to speed-up memory accesses and store constant data like locality strings. For the developer, it is not transparent how constants are stored in the compiled binary. Thus, even if the code seems to be placed in a cache line, *i.e.*, 64 B granularity, the compiler might reorder strings and add more spatial granularity between data. One optimization to reduce the binary size is to only populate the read-only data section if the compiler observes

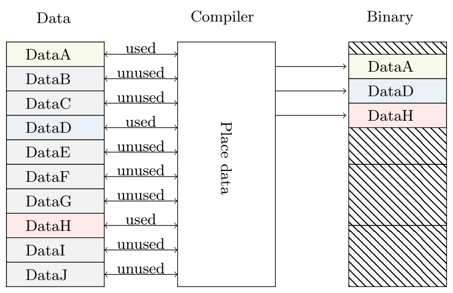


Fig. 2: First-come-first-serve population of the `.rodata` section in the binary.

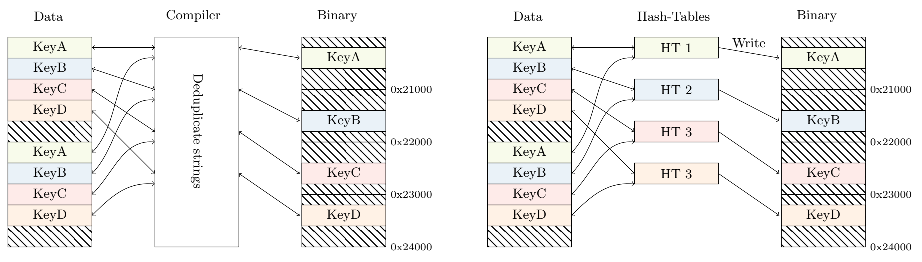
```
struct MapEntry {
  const char* key;
  const char* value;
};
#define LANGUAGE_CODE(key,value) \
  { key, value }
#define MAP_DECL constexpr MapEntry mappings[] = MAP_DECL {
  LANGUAGE_CODE("KeyA","DataA"), LANGUAGE_CODE("KeyB","DataB")
};
#undef MAP_DECL
void string_funcA(vector<string>& v) {
  string local_ro_string = "DataB";
  v.push_back(local_ro_string);
  string padding_string = "<64-byte-string>";
  v.push_back(padding_string);
}
void string_funcB(vector<string>& v) {
  MapEntry k1 = mappings[0];   //KeyA
  v.push_back(k1.value);
  MapEntry k2 = mappings[1]; //KeyB
  v.push_back(k1.value);
}
```

Listing 1.1: Strings are deduplicated in the binary and could lead to spatial distance between readonly-strings in the same array in combination with first-come-first-serve data placement.

that only certain indices of a lookup table are accessed. If the developer uses a macro to dynamically populate a lookup table, e.g., with key mappings or similar, compilers do not insert all elements into the read-only section of the binary to reduce the binary size. Instead, the compilers use a first-come-first-serve data placement strategy to place the data in the read-only section. Figure 2 illustrates how data can be placed in `.rodata` section caused by this strategy.

4.2 Data Deduplication during Compilation

Another optimization facilitating cache attacks, also in combination with the first-come-first-serve data placement we just discussed, is data deduplication during compilation. Deduplicating strings can reduce the binary size significantly but also the memory resident size when running the program, as strings do



(a) Compiler (b) Linker

Fig. 3: String deduplication in the compiler and linker causing spatial distance in `.rodata` section of the binary.

not have to be kept in memory multiple times. Figure 3a demonstrates how string deduplication can introduce spatial distance in sections of the binary, for instance, the `.rodata` section. C/C++ compilers deduplicate strings that occur more than once in the source code. Listing 1.1 illustrates a situation where string deduplication can be performed. Both the lookup table `mappings` and the function `string_funcA` contain the string `DataA`. The compiler traverses over the functions, and `DataB` is first inserted into the `.rodata` section. Again, data processed by the compiler (padding) could cause spatial distance between `DataA` and `DataB`. Before the compiler inserts `DataB` (`mappings[1]` in `string_funcB`), the compiler checks for duplicates and only points to the existing occurrence of `DataB` in the `.rodata` for all future usages. We evaluate Listing 1.1 for GCC and Clang. For Clang, we observe again for all optimizations levels the ordering `DataA.<64-byte-string>.DataB` in the `.rodata` section. For GCC, we observe the same result that for optimization levels O0/O1, both values are populated next to each other in the `.rodata` (`DataA.DataB`). For the other levels, the small strings are encoded as immediate values.

4.3 Deduplication in the linking step.

As we showed, string deduplication can cause spatial distance between strings and enable side-channel attacks in the compile step. For large software projects such as the Chromium project, it is important to merge strings also across object files. Since 2017, lld uses multiple hash tables to compensate some of the overhead caused by this link-time optimization by increasing concurrency using hash tables that can be accessed in parallel [51]. However, as there are multiple tables, inserting merged strings can cause a different layout for strings in the `.rodata` section than in the final linked binary. Figure 3b illustrates how the concurrent merging can lead to spatial distance in the final binary. With the highest optimization level of `lld` linker, *i.e.*, `-O2` [45], the linker merges duplicate substrings contained in larger strings. The smaller substring will be removed, and the tail of the larger string is used to index the substring. The security implications of string deduplication need to be considered in software projects since large spatial distance between secret dependent values, such as different key inputs, can lead to leakage of all user input, as we show in Section 5.

5 Evaluation and Exploitation Phase

In this section, we evaluate our templater on large binaries, such as browsers, that have not been targeted with templating attacks so far. We evaluate how well the templates work in the exploitation phase in terms of the attack F-Score. For the exploitation phase, we, the attacker, runs without privileges on a default configured system, with background activity (running e.g., browser, mail client, chat clients, music and video streaming, virus scanning, system updates running, etc.) leading to a realistic amount of system activity and noise. Overall we found that Flush+Reload is extremely noise-resilient, in line with previous works [77,32]. We also focus on widespread Chromium-based products and demonstrate that they are susceptible to *LBTA*. We analyze the root cause for the leakage and show that it is caused by a compiler optimization. Table 1 lists all the evaluated applications, including the Chromium-based browsers and applications, Firefox, and LibreOffice Writer.

Templating of HTML form input fields Chrome. We first run our templating tool while generating keystrokes. We run our templater on an Intel i7-6700K with a fixed frequency of 4 GHz running Ubuntu 20.04 (kernel 5.4.0-40) on Chrome version 100.0.4896.60. To get more accurate results during the templating phase, we recommend dropping the active caches before executing the templater via procfs (`/proc/sys/vm/drop_caches`). Moreover, we blacklist file mappings from the `/usr/share/fonts/` as they lead to inconsistent results during the evaluation phase. Our templater traces 57 different key codes of a common `US_EN` keyboard in HTML password fields over the total size of memory mappings in Chrome of 209.81 MB (including the main binary and shared libraries). For each key code, we sample 20 times. On average, we observe a runtime of 1.47 hours ($n = 10, \sigma_{\overline{x}} = 0.33\%$) for 57 key codes, including the time for key classification. For a single key code, the runtime is 92 seconds. For comparison, the cache template attack implementation by Gruss et al. [32] takes 113.17 days to template the same files. Thus, with 1.47 hours *LBTA* speeds up the templating by a factor of 1848.

Leakage Source in Chrome. As we discovered the page offsets related to the different keystrokes, we want to find the exact cache line causing the cache leakage. We extend our monitor with Flush+Reload to determine the cache line within the page. To speed up the templating time and obtain precise information on which cache line has the highest correlation, we disable most of the Intel prefetchers by writing the value `0xf` to MSR `0x1a4` [67], as otherwise multiple cache lines would have the highest correlation. We map the Chrome binary as shared memory and perform Flush+Reload on all mapped cache lines to determine the corresponding cache lines for each key. We analyze the Chrome binary and lookup the offsets causing the leakage for a specific keystroke. Each cache line causing the leakage of a certain character contains a string for the key event, e.g.,

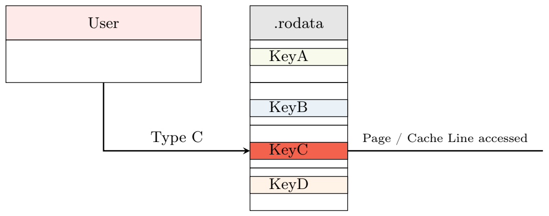
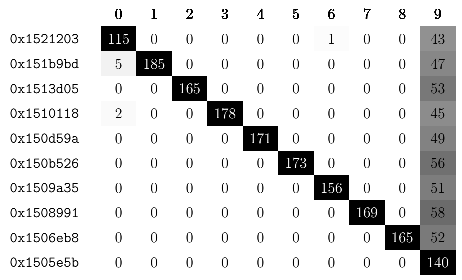


Fig. 4: Key code strings in the `.rodata` section introduce cache leakage.

"KeyA". We observe that all offsets lie in the read-only data (`.rodata` section of the binary. The leakage source are key-dependent accesses to the key code strings in the dom code table,[5] e.g., `DOM_CODE(0x070004, 0x001e, 0x0026, 0x001e, 0x0000, "KeyA", US_A);`. Figure 4 illustrates the leakage source for a user typing in a certain character and the corresponding DOM_CODE for the UI event. To verify if the leakage is related to string deduplication, we download the Chromium source, disable the string deduplication `-fno-merge-all-constants` and rebuild the Chromium browser. We still observe, that the single keystrokes are spread over multiple pages in the `.rodata` section, which can still be exploited by the attacker despite the overheads in binary size and execution runtime caused by disabling the optimization. Hence, the compiler flag to disable string deduplication *does not fully close the side channel.* As a next step, we analyze the compiled object files after the build process. We observe that the created object file `keycode_converter.o` still contains all the key event strings adjacent to each other in the binary. This indicates that the linker introduces the spatial distance between key event strings. We perform a binary search on older Chrome binaries from a public Github repository containing archived Chrome Debian packages [71] to see when the spatial distance for key event strings was introduced. As a result, we observe that between version 63 and 64 of Chrome (year 2017), the single key event string was placed in the `.rodata` at different 4 kB pages. According to [51], the linker optimizations have been constantly improved since 2017. As discussed in Section 4, the parallelism in string deduplication can also cause spatial distance between key events. Disabling the string merging optimization is currently only possible by *disabling all optimizations* using optimization level `O0` for the linking with `-Wl,-O0`. This removes the spatial distance between the key event strings but comes with a substantial overhead as optimizations are disabled. At any higher optimization level, e.g., `-Wl,-O1`, the spatial distance reappears as strings are again deduplicated. This confirms that one of the effects we exploit is introduced by the linker. In comparison to state-of-the-art keyloggers on Linux like xkbcat [1], our keylogger does not rely on running as the same user within the same X-session. We verify this by running our keylogger as a different user and can still recover the keys from Chrome.

[5] `https://source.chromium.org/chromium/chromium/src/+/main:ui/events/keycodes/dom/dom_code_data.inc`



	0	1	2	3	4	5	6	7	8	9
0x1521203	115	0	0	0	0	0	1	0	0	43
0x151b9bd	5	185	0	0	0	0	0	0	0	47
0x1513d05	0	0	165	0	0	0	0	0	0	53
0x1510118	2	0	0	178	0	0	0	0	0	45
0x150d59a	0	0	0	0	171	0	0	0	0	49
0x150b526	0	0	0	0	0	173	0	0	0	56
0x1509a35	0	0	0	0	0	0	156	0	0	51
0x1508991	0	0	0	0	0	0	0	169	0	58
0x1506eb8	0	0	0	0	0	0	0	0	165	52
0x1505e5b	0	0	0	0	0	0	0	0	0	140

Fig. 5: Cache-hit ratio using Flush+Reload for all digits letters in Chrome.

Keylogging in Chrome with Flush+Reload. We run our monitor in three experiments for 180 seconds with all lowercase alphanumeric characters and observe cache activity for every single keystroke. The first experiment runs with fast user input with 1 ms between each keystroke. We count cache hits following a keystroke as true positives if they occur on the cache line that is correct according to our template and as false positive otherwise. To obtain the number of false positives, we run the monitor in a second experiment without performing any keystrokes in the input field, *i.e.*, idling. To complete our data on false negatives and true positives, we run the monitor in a third experiment while performing user input with 1 s between each keystroke. Over the total 540 second measurement time frame, we observed no false negatives. Figure 7 (Appendix) shows the cache-hit ratio for the cache lines detecting lowercase letters in Chrome. Figure 5 shows the cache-hit ratio for the cache lines detecting numeric digits in Chrome. As shown from Figure 5, the different digits can be highly-accurately classified. As can be seen, the cache line accessed for digit 9 also contains other data that is constantly accessed by code handling other events in Chrome. Therefore, the cache line is constantly accessed also in an idle state and, in practice, cannot be used to spy on digit 9. From all the 36 alphanumeric keys, this is the only character where code or data is co-located with other (unrelated) frequently accessed code or data. The F-Score is the harmonic mean of precision and recall. Section 5 illustrates the F-Score for all alphanumeric characters. We also observed that a single keystroke causes up to three cache hits. These cache hits could be related to the window events `key_up`, `key_pressed` and `key_down`. To avoid printing the same character multiple times, a cache miss counter between the keystrokes can be used [32]. Note that multiple cache lines can be considered to further increase the accuracy of the keylogger [41,69,10].

Keylogging with the page cache. To demonstrate that the Chrome leakage is not specific to a certain CPU, we run our keylogger on Chrome version 99.0.4844.84. Our test device runs Ubuntu 20.04 (kernel 5.18.0-051800-generic), equipped with an AMD Ryzen 5 2600X CPU, 16 GB of RAM, and a Samsung 970 EVO NVME SSD. We circumvent the read-around and read-ahead optimization, as explained in Section 3.5. The keylogger uses the keystroke template for the main Chrome binary and monitors the page cache utilization for the corresponding pages using the `preadv2` syscall. It then reports the detected activity

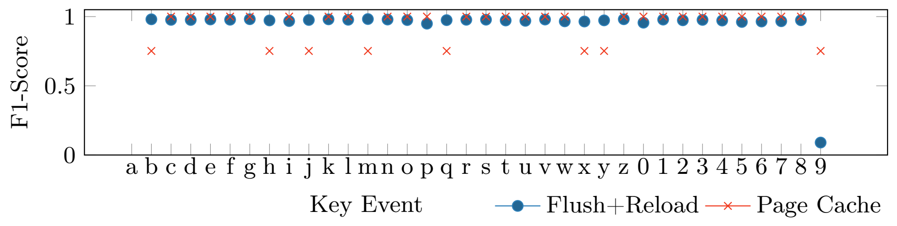


Fig. 6: F-Score per key using Flush+Reload and page cache attacks for all alphanumeric characters in Chrome.

as keystrokes and subsequently evicts the page cache. While the page cache attack using the `mincore` syscall was able to observe keystrokes on a fine temporal granularity, we observe that using `preadv2` comes with practical limitations. In particular, with large eviction set sizes, guessed by the attacker, we conclude that only very slow keyboard interaction with gaps of 2 s and more can be observed. However, our evaluation of the page-cache side channel is generic and would also apply to scenario where the `mincore` syscall is available, which allowed fast and non-destructive continuous probing.

Based on the page cache accesses, we compute the page-hit ratio for Chrome over the page cache. Figure 7 (Appendix) shows the page-hit ratio for the page cache detecting alphanumeric letters in Chrome. For the Chrome version, we observe that the characters `b,m,9,h,y,x` are grouped and cannot be uniquely distinguished. We again perform the experiment in three phases to determine true positive, false positive, and false negative rate by simulating fast, slow, and no user input. Section 5 shows the F-Score for all alphanumeric characters running the page cache attack. While most characters have very high F-Scores, the character group `b,m,9,h,y,x` has a lower F-Score due to false positives when other keys are pressed. Also, same as in the Flush+Reload attack, the character `9` suffers from a high number of false positives, negatively impacting the F-Score.

Electron. As we observed the leakage of keystrokes within the Chrome binary, we further analyzed Chromium-based applications like the Electron framework. As Chromium-based applications largely use the same keystroke handling code, we can directly scan the `.rodata` section for the keystroke offsets. We evaluate the templates on Chromium, Threema, Passky, VS-Code, Mattermost, Discord and observe similar leakage rates to Chrome with F-Scores of at least 85 %. Table 1 contains the F-Scores for the different applications. Based on these clear results, we deduce that, in principle, all Electron applications are susceptible to *LBTA* and cache-based keylogging attacks.

Chromium Embedded Framework. The Chromium Embedded Framework (CEF) is widely used and another interesting target for *LBTA*. While Electron directly uses the Chromium API, CEF tries to hide the details of the Chromium API [24]. CEF is actively run on more than 100 million devices [12]. We target Spotify, and the Brackets editor application, which are both based on CEF. To attack a CEF application, an attacker needs to read out the `.rodata` section from the shared library `libcef.so`. We run our monitor again with Flush+Reload and observe an F-Score of 96 % over the lowercase alphanumeric characters. For Brackets (1.5.0), we observe, that the `libcef.so` was built with an older linker version as the different key-event related strings for the lowercase alphanumeric characters are co-located in three different cache lines. Therefore, we consider all CEF applications to be susceptible to cache templating in principle. We observe an F-Score of 94 % for detecting key events. However, we also observe that hardware prefetching practically thwarts the distinction of different blocks in this scenario more than in the other attack scenarios, leaving only inter-keystroke timing attacks as an option for the attack phase.

Firefox. Firefox uses a different build system where optimizations such as data deduplication may still apply but with slightly different behavior than with LLVM/clang. Therefore, we templated Firefox and found cache activity for each keystroke in the `libxul.so` library (offset: 0x332d000). However, we did not find leakage to distinguish keys. However, an attacker can still determine whether a user is typing and perform an inter-keystroke timing attack [61,50,79,20,32] to recover the keystrokes. The accuracy we observed for such an attack is 96 %.

LibreOffice Writer. We profile the LibreOffice Writer version 6.4.2 on our Linux setup. Our profiler shows that the library `libQt5XcbQpa.so.5.12.8` (offset: 0x51000) offset reveals cache activity on all letters but no digits. The library `libswlo.so` (0x53e000) leaks keystrokes reliably with an F-Score of 1.

Chrome on Windows. Chrome on Windows is built with a different compiler and linker. Therefore, we tested Chrome versions 103.0.5060.53 and 114 on an Intel i5-4300U notebook running Windows 10 (1803, 17134.1726). We use the `LoadLibrary` function create read-only shared mappings with victim applications. We observe that in the `chrome.dll` (offset: 0xa4ee000) the different key values are co-located instead of having a spatial distance of multiple 4 kB pages. With our Flush+Reload cache monitor we are able to observe all key presses and distinguish presses in the key groups A-F, G-S, T-Z and 0-4, and 5-9, with an F-Score of 99 %. However, due to prefetching we can only monitor a single key group at a time. We also found user input leakage on many other locations, e.g., `msctf.dll` (0x45000), and `imm32.dll` (0x3000).

Search bar. Templating user queries in the browser would tremendously reduce the privacy of browsers. Running the templater on the search bar of Chrome 103.0.5060.53 revealed that the search bar uses a different method to load the keys and there is only a single page (offset: 0x91d4000) in Chrome with cache activity upon keystrokes. Based on our results, we conclude that the search bar does not use the same internal structures for key events as HTML input data. Still, the leakage we discovered enables inter-keystroke timing attacks on keystrokes. Running the profiling experiment with all alphanumeric, we achieve an F-Score of 99 % for detecting key presses.

Table 1: Evaluated applications. Page cache (PC) and cache line (CL) indicate whether precise keystroke attacks are possible on that granularity. Inter-Keystroke Timing (IK) indicates that key events can be detected on the application via Flush+Reload or the page cache.

Name	Category	CL	PC	IK (key groups)	Avg. F-Score (Flush+Reload)
Chrome (99.0.4844.84)	Browser	✓	✓	✓	94 %
Signal-Desktop (5.46.0)	Private Messenger	✓	✓	✓	98 %
Threema (2.4.1)	Private Messenger	✓	✓	✓	84 %
Passky (7.0.0)	Password Manager	✓	✓	✓	99 %
VS-Code (1.69.1)	Editor	✓	✓	✓	85 %
Chromium Browser (103.0.5060.114)	Browser	✓	✓	✓	99 %
Mattermost-Desktop (5.1.1)	Collaboration Platform	✓	✓	✓	94 %
Discord (0.0.18)	Text and Voice Chat	✓	✓	✓	98 %
Spotify (1.1.84.716)	Audio Streaming	✓	✓	✓	96 %
Brackets (1.2.1)	Editor	✗	✗	✓	94 %
Chrome 103.0.5060.134(Windows)	Browser	✗	✗	✓	99 %
Chrome 103.0.5060.53 (Search Bar)	Browser	✗	✗	✓	99 %
libxul.so (Firefox 102)	Browser	✗	✗	✓	99 %
LibreOffice Writer (6.4.2)	Office Software	✗	✗	✓	99 %

6 Mitigation and Discussion

Different mitigation vectors could prevent either *LBTA* or the underlying leakage utilized in the exploitation phase, albeit at a significant performance and usability cost. We identified five conditions for an attack to succeed:

Golden device availability Templating attacks consist of two phases. In the templating phase, the attacker uses a setup that is similar to the victim system [13,9,32]. This is trivial for cache attacks on most desktop and laptop processors, as they are virtually identical in terms of attacks like Flush+Reload (*i.e.*, the processor has cache lines and eviction or flushing of these is possible). Software diversity [18], in principle, could break the link between templating and exploitation, but is not widely used. Thus, in practice, the vast majority of users runs binaries obtained from the official repositories or websites, making it trivial to create templates for them. Furthermore, even with software diversity, once the attacker knows what the target byte sequences (e.g., strings) in the binary are, the attacker can simply search for these on the victim system (without the need for templating again) and attack the victim binary in the same way again. Hence, we also consider software diversity no mitigation to *LBTA*.

Disable Compiler and Linker Optimizations For the Chromium example, disabling the linker optimizations (deduplication and spatial distancing) would reduce the accurate keylogging to inter-keystroke timings for key groups in 4 different cache lines. However, this may still enable inferring user input accurately [61]. On the negative side, removing these optimizations typically increases binary sizes and cache utilization due to runtime use of duplicated data. Note that this type of deduplication and spatial distancing is introduced on the compiler and linker level, which is completely transparent to the OS. While the OS could dynamically rewrite binary pages at runtime to counteract this behavior, this would introduce huge amounts of complexity, overhead, and the potential for unhandled corner cases. Instead, the Chromium team opted for a compiler- and linker workaround, which triggers the string placement explicitly by placing and initializing dummy data structures such that the current compiler and linker versions do not spatially separate the secret data. However, this approach is fragile as it depends on the specific behavior of the compiler.

Secret-dependent execution For cryptographic code, the state of the art against side channels is the linearization to so-called constant-time code, *i.e.*, constant code and data accesses, regardless of the secrets, albeit with a considerable performance cost [15]. For general purpose code, always running all the code and accessing all the data is infeasible. Different works linearized the control flow of general purpose code [21,57,5] and observed a prohibitively high runtime overhead for realistic workloads. Hence, the problem of secret dependency on user input in large applications remains an open problem.

Side-channel observability Tools like CacheAudit [22] or CaSym [8] follow the cryptography-focused notion of constant time to consider an application leakage-free. However, in practice, distinguishing keys may be infeasible for an unprivileged attacker when key-dependent execution exists but does not cross, e.g., page or cache-line boundaries, depending on the side channel. In particular, within a page, the hardware prefetcher is a substantial obstacle introducing spurious cache activity on the target cache lines, foiling exploitation in practice [32]. The compiler could utilize this effect by grouping potentially secret-dependent accesses, minimizing the number of cache lines data structures are spread across, and placing strings interleaved with frequently used code or data.

Noise resilience Since user input cannot be triggered and repeated by the attacker millions of times, noise resilience is also one condition. Hence, inducing noise, unsuitable to secure cryptographic operations, can provide strong security guarantees for user input [56]. A low number of memory accesses could substantially limit the presented attacks, especially if user annotations of potentially secret data tell the compiler where to add these accesses.

LBTA is also interesting as a defensive technique revealing leakage as part of a continuous integration pipeline [74], revealing leakage that is not or not to the actual extent visible to developers on the source level, but only in the binary due to compiler and linker optimizations introducing these spatial distances. Moreover, languages like JavaScript, Java, PHP, and Python also perform string deduplication (under the term ‘string interning’) to reduce memory utilization, potentially leading to similar effects.

We demonstrated that keystrokes in form input fields in Chrome can be detected using cache attacks on hardware and software caches. While Chrome is a valuable target, the dependency of many frameworks on the Chromium project, such as CEF and Electron, leads to a significantly higher impact as browser-based desktop applications, e.g., using the popular Electron framework [23], are susceptible to accurate keylogging with our attack.

7 Conclusion

First-come-first-serve data placement and data deduplication during compilation and linking facilitate side-channel leakage in compiled binaries. We show that this effect can even induce side-channel leakage where, without these optimizations, no secret-dependent accesses cross a 64-byte boundary. The foundation to discover this attack was our extension to cache template attacks, called Layered Binary Templating Attacks, *LBTA*. *LBTA* is a scalable approach to templating that combines spatial information from multiple side channels. Using *LBTA* we scan binaries compiled with LLVM/clang, which applies first-come-first-serve data placement and deduplication by default. Our end-to-end attack is an unprivileged cache-based keylogger for all Chrome-based / Electron-based applications, including many security-critical apps, e.g., the popular Signal messenger app. While mitigation strategies exist, they come at a cost, and further research is necessary to overcome the open problem of side-channel attacks on user input.

Acknowledgments

We want to thank our anonymous reviewers for valueable feedback on the draft. This work was supported by a generous gift from Red Hat Research. We want to thank Hanna Müller, Claudio Canella, Michael Schwarz and Moritz Lipp for valuable feedback. Any opinions or recommendations expressed are those of the authors and do not necessarily reflect the views of the funding parties.

References

1. Antti Korpi: xkbcat (2021), `https://github.com/anko/xkbcat`
2. Bacs, A., Musaev, S., Razavi, K., Giuffrida, C., Bos, H.: DUPEFS: Leaking Data Over the Network With Filesystem Deduplication Side Channels. In: FAST (2022)
3. Baert, M.: wayland-keylogger (2022), `https://github.com/Aishou/wayland-keylogger`
4. Bernstein, D.J.: Cache-Timing Attacks on AES (2005), `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`
5. Borrello, P., D'Elia, D.C., Querzoni, L., Giuffrida, C.: Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization. In: CCS (2021)
6. Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., Sadeghi, A.R.: Software Grand Exposure: SGX Cache Attacks Are Practical. In: WOOT (2017)
7. Brennan, T., Rosner, N., Bultan, T.: JIT Leaks: inducing timing side channels through just-in-time compilation. In: S&P (2020)
8. Brotzman, R., Liu, S., Zhang, D., Tan, G., Kandemir, M.: CaSym: Cache aware symbolic execution for side channel detection and mitigation. In: S&P (2019)
9. Brumley, B., Hakala, R.: Cache-Timing Template Attacks. In: AsiaCrypt (2009)
10. Carre, S., Dyseryn, V., Facon, A., Guilley, S., Perianin, T.: End-to-end automated cache-timing attack driven by Machine Learning. Journal of Cryptology (2019)
11. Cauligi, S., Soeller, G., Brown, F., Johannesmeyer, B., Huang, Y., Jhala, R., Stefan, D.: FaCT: A flexible, constant-time programming language. In: SecDev (2017)

12. CEF: Chrome Embedded Framework (2022), https://github.com/chromiumembedded/cef
13. Chari, S., Rao, J.R., Rohatgi, P.: Template attacks. In: CHES (2002)
14. Chromium: Speeding up Chrome's release cycle (2022), https://blog.chromium.org/2021/03/speeding-up-release-cycle.html
15. Chung, S.C., Lee, J.W., Chang, H.C., Lee, C.Y.: A high-performance elliptic curve cryptographic processor over GF(p) with SPA resistance. In: International Symposium on Circuits and Systems (ISCAS) (2012)
16. Coppens, B., Verbauwhede, I., De Bosschere, K., De Sutter, B.: Practical mitigations for timing-based side-channel attacks on modern x86 processors. In: S&P (2009)
17. Costi, A., Johannesmeyer, B., Bosman, E., Giuffrida, C., Bos, H.: On the effectiveness of same-domain memory deduplication. In: European Workshop on Systems Security. pp. 29–35 (2022)
18. Crane, S., Homescu, A., Brunthaler, S., Larsen, P., Franz, M.: Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In: NDSS (2015)
19. Dall, F., De Micheli, G., Eisenbarth, T., Genkin, D., Heninger, N., Moghimi, A., Yarom, Y.: Cachequote: Efficiently recovering long-term secrets of SGX EPID via cache attacks. In: CHES (2018)
20. Diao, W., Liu, X., Li, Z., Zhang, K.: No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis. In: S&P (2016)
21. Domas, C.: M/o/Vfuscator (2015), https://github.com/xoreaxeaxeax/movfuscator
22. Doychev, G., Feld, D., Kopf, B., Mauborgne, L., Reineke, J.: CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In: USENIX Security Symposium (2013)
23. Electron: Electron Apps (2022), https://www.electronjs.org/apps
24. Electron JS: Electron Internals: Building Chromium as a Library (2022), https://www.electronjs.org/blog/electron-internals-building-chromium-as-a-library
25. Fu, Y., Bauman, E., Quinonez, R., Lin, Z.: SGX-LAPD: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults. In: RAID (2017)
26. García, C.P., Brumley, B.B.: Constant-Time Callees with Variable-Time Callers. In: USENIX Security Symposium (2017)
27. Götzfried, J., Eckert, M., Schinzel, S., Müller, T.: Cache Attacks on Intel SGX. In: EuroSec (2017)
28. Gras, B., Razavi, K., Bosman, E., Bos, H., Giuffrida, C.: ASLR on the Line: Practical Cache Attacks on the MMU. In: NDSS (2017)
29. Gruss, D., Bidner, D., Mangard, S.: Practical Memory Deduplication Attacks in Sandboxed JavaScript. In: ESORICS (2015)
30. Gruss, D., Kraft, E., Tiwari, T., Schwarz, M., Trachtenberg, A., Hennessey, J., Ionescu, A., Fogh, A.: Page Cache Attacks. In: CCS (2019)
31. Gruss, D., Maurice, C., Fogh, A., Lipp, M., Mangard, S.: Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS (2016)
32. Gruss, D., Spreitzer, R., Mangard, S.: Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security Symposium (2015)
33. halolinux: Page Cache Readahead (2022), https://www.halolinux.us/kernel-architecture/page-cache-readahead.html
34. Harnik, D., Pinkas, B., Shulman-Peleg, A.: Side channels in cloud services, the case of deduplication in cloud storage. IEEE Security & Privacy (6) (2010)

35. Hund, R., Willems, C., Holz, T.: Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P (2013)
36. John Richard Moser: Optimizing Linker Load Times (2006), `https://lwn.net/Articles/192624/`
37. Jonathan Corbet: Fixing page-cache side channels, second attempt (2019), `https://lwn.net/Articles/778437/`
38. Keelveedhi, S., Bellare, M., Ristenpart, T.: DupLESS: Server-Aided Encryption for Deduplicated Storage. In: USENIX Security Symposium (2013)
39. Kocher, P.: Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: CRYPTO (1996)
40. Li, G., Liu, C., Yu, H., Fan, Y., Zhang, L., Wang, Z., Wang, M.: SCNet: A Neural Network for Automated Side-Channel Attack. arXiv:2008.00476 (2020)
41. Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., Mangard, S.: ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security Symposium (2016)
42. Maurice, C., Weber, M., Schwarz, M., Giner, L., Gruss, D., Alberto Boano, C., Mangard, S., Römer, K.: Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS (2017)
43. Medwed, M., Oswald, E.: Template attacks on ECDSA. In: WISA. Springer (2008)
44. Moghimi, A., Irazoqui, G., Eisenbarth, T.: CacheZoom: How SGX amplifies the power of cache attacks. In: CHES (2017)
45. nxmnpg.lemoda: Manual Pages - LD.LLD (2022), `https://nxmnpg.lemoda.net/1/ld.lld`
46. Oren, Y., Kemerlis, V.P., Sethumadhavan, S., Keromytis, A.D.: The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In: CCS (2015)
47. Page, D.: A note on side-channels resulting from dynamic compilation. Cryptology ePrint archive, Report 2006/349 (2006)
48. Rane, A., Lin, C., Tiwari, M.: Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In: USENIX Security Symposium (2015)
49. Rechberger, C., Oswald, E.: Practical template attacks. In: WISA (2004)
50. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In: CCS (2009)
51. Rui Ueyama: lld: A Fast, Simple and Portable Linker (2017), `https://llvm.org/devmtg/2017-10/slides/Ueyama-lld.pdf`
52. Russinovich, M.E., Solomon, D.A., Ionescu, A.: Windows internals. Pearson Education (2012)
53. Saileshwar, G., Fletcher, C.W., Qureshi, M.: Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion. In: ASPLOS (2021)
54. Schwarz, M., Lackner, F., Gruss, D.: JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits. In: NDSS (2019)
55. Schwarz, M., Lipp, M., Canella, C.: misc0110/PTEditor: A small library to modify all page-table levels of all processes from user space for x86_64 and ARMv8 (2018), `https://github.com/misc0110/PTEditor`
56. Schwarz, M., Lipp, M., Gruss, D., Weiser, S., Maurice, C., Spreitzer, R., Mangard, S.: KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In: NDSS (2018)
57. Schwarzl, M., Canella, C., Gruss, D., Schwarz, M.: Specfuscator: Evaluating Branch Removal as a Spectre Mitigation. In: FC (2021)
58. Schwarzl, M., Kraft, E., Lipp, M., Gruss, D.: Remote Page Deduplication Attacks. In: NDSS (2022)

59. Shih, M.W., Lee, S., Kim, T., Peinado, M.: T-SGX: Eradicating controlled-channel attacks against enclave programs. In: NDSS (2017)
60. Simon, L., Chisnall, D., Anderson, R.: What you get is what you C: Controlling side effects in mainstream C compilers. In: EuroS&P (2018)
61. Song, D.X., Wagner, D., Tian, X.: Timing Analysis of Keystrokes and Timing Attacks on SSH. In: USENIX Security Symposium (2001)
62. statcounter Global Stats: Browser Market Share Worldwide (2022), `https://gs.statcounter.com/`
63. Suzaki, K., Iijima, K., Yagi, T., Artho, C.: Memory Deduplication as a Threat to the Guest OS. In: EuroSys (2011)
64. Van Bulck, J., Weichbrodt, N., Kapitza, R., Piessens, F., Strackx, R.: Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In: USENIX Security Symposium (2017)
65. Van Cleemput, J., De Sutter, B., De Bosschere, K.: Adaptive compiler strategies for mitigating timing side channel attacks. TDSC (2017)
66. Van Schaik, S., Giuffrida, C., Bos, H., Razavi, K.: Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In: USENIX Security Symposium (2018)
67. Viswanathan, V.: Disclosure of Hardware Prefetcher Control on Some Intel Processors (2014), `https://web.archive.org/web/20160304031330/https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors`
68. Wajahat, A., Imran, A., Latif, J., Nazir, A., Bilal, A.: A Novel Approach of Unprivileged Keylogger Detection. In: iCoMET (2019)
69. Wang, D., Neupane, A., Qian, Z., Abu-Ghazaleh, N., Krishnamurthy, S.V., Colbert, E.J., Yu, P.: Unveiling your keystrokes: A Cache-based Side-channel Attack on Graphics Libraries. In: NDSS (2019)
70. Wang, S., Wang, P., Liu, X., Zhang, D., Wu, D.: CacheD: Identifying Cache-Based Timing Channels in Production Software. In: USENIX (2017)
71. Webnicer Ltd: chrome-downloads (2022), `https://github.com/webnicer/chrome-downloads/`
72. Weiser, S., Spreitzer, R., Bodner, L.: Single Trace Attack Against RSA Key Generation in Intel SGX SSL. In: AsiaCCS (2018)
73. Wichelmann, J., Moghimi, A., Eisenbarth, T., Sunar, B.: MicroWalk: A Framework for Finding Side Channels in Binaries. In: ACSAC (2018)
74. Wichelmann, J., Sieck, F., Pätschke, A., Eisenbarth, T.: Microwalk-ci: Practical side-channel analysis for javascript applications. arXiv preprint arXiv:2208.14942 (2022)
75. Xu, Y., Cui, W., Peinado, M.: Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In: S&P (2015)
76. Xu, Y., Bailey, M., Jahanian, F., Joshi, K., Hiltunen, M., Schlichting, R.: An exploration of L2 cache covert channels in virtualized environments. In: CCSW (2011)
77. Yarom, Y., Falkner, K.: Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium (2014)
78. Yuan, Y., Pang, Q., Wang, S.: Automated Side Channel Analysis of Media Software with Manifold Learning. arXiv preprint arXiv:2112.04947 (2021)
79. Zhang, K., Wang, X.: Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In: USENIX Security Symposium (2009)

A Cache-hit ratios (extended)

The cache hit ratio for all lowercase characters with Flush+Reload can be seen with Figure 8 and all alphanumeric characters for the page cache attack Figure 7.

	a	c	d	e	f	g	i	j	k	l	n	o	p	q	r	s	t	u	v	w	z	0	1	2	3	4	5	6	7	8	yb9mhx
0x14e7000	98	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
0x14e2000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	100
0x14df000	0	95	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14d2000	0	0	99	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14c1000	0	0	0	98	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0x14c0000	0	0	0	0	97	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14be000	0	0	0	0	0	98	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0x14bc000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	100
0x14bb000	0	0	0	0	0	0	99	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14ba000	0	0	0	0	0	0	0	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14b9000	0	0	0	0	0	0	0	0	98	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
0x14b3000	0	0	0	0	0	0	0	0	0	91	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6
0x14af000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	100
0x14ab000	0	0	0	0	0	0	0	0	0	0	97	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
0x14aa000	0	0	0	0	0	0	0	0	0	0	0	94	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0x14a9000	0	0	0	0	0	0	0	0	0	0	0	0	99	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0x14a8000	0	0	0	0	0	0	0	0	0	0	0	0	0	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14a2000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	97	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0x149b000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	97	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0x1494000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	96	0	0	0	0	0	0	0	0	0	0	0	0	0	2
0x1492000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	98	0	0	0	0	0	0	0	0	0	0	0	0	1
0x148f000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	93	0	0	0	0	0	0	0	0	0	0	0	1
0x148e000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	99	0	0	0	0	0	0	0	0	0	0	1
0x148c000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	100
0x148b000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	100
0x148a000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	98	0	0	0	0	0	0	0	0	0	1
0x1521000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	98	0	0	0	0	0	0	0	0	1
0x151b000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	99	0	0	0	0	0	0	0	1
0x1513000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	97	0	0	0	0	0	0	3
0x1510000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	98	0	0	0	0	0	1
0x150d000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	97	0	0	0	0	1
0x150b000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	97	0	0	0	2
0x1509000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	98	0	0	0
0x1508000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	97	0	0
0x1506000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	100	0
0x1505000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	100

Fig. 7: Cache-hit ratio using a page cache attack for alphanumeric characters in Chrome.

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
`0x14e7e5b`	99	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	2	0	0	2	0	0	8	7	0	0
`0x14e24a8`	2	178	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
`0x14df043`	0	0	151	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
`0x14d2ab9`	0	0	0	173	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
`0x14c1a5e`	0	1	0	0	101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
`0x14c0a76`	0	0	0	0	0	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
`0x14be05f`	0	0	0	0	0	0	96	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
`0x14bc5ad`	0	0	0	0	0	0	0	169	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
`0x14bb06c`	0	0	0	0	0	0	0	0	114	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
`0x14bae3b`	0	0	0	0	0	0	0	0	0	156	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
`0x14b981f`	0	1	0	0	0	0	0	0	0	0	171	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
`0x14b3350`	0	0	0	0	0	0	0	0	0	0	0	146	0	0	0	0	0	0	0	0	0	0	0	0	0	0
`0x14af573`	0	1	0	0	0	0	0	0	0	0	0	0	125	0	0	0	0	0	0	0	0	0	0	0	0	0
`0x14ab755`	0	0	0	0	0	0	0	0	0	0	0	0	0	148	0	0	0	0	0	0	0	1	0	0	0	0
`0x14aa938`	0	0	0	0	0	0	0	0	0	0	0	0	0	0	165	0	0	0	0	0	0	0	0	0	0	0
`0x14a9172`	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	87	0	0	0	0	0	0	0	0	0	0
`0x14a8e1c`	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15	156	0	0	0	0	0	0	0	0	0
`0x14a2f87`	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	141	0	0	0	0	0	0	0	0
`0x149b36d`	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	108	0	0	0	0	0	0	0
`0x149411c`	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	157	0	0	0	0	0	0
`0x1492c1d`	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	135	0	0	0	0	0
`0x148f151`	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	107	0	0	0	0
`0x148e546`	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	124	0	0	0
`0x148cb27`	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	138	0	0
`0x148b08b`	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	137	0
`0x148ac19`	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	141

Fig. 8: Cache-hit ratio using Flush+Reload for lowercase letters in Chrome.