Side-Channel Attacks on Everyday Applications: Distinguishing Inputs with FLUSH+RELOAD

Taylor Hornby
taylor@defuse.ca

University of Calgary
Zcash

Abstract

Side channel attacks are typically used to break implementations of cryptography. Recently, side-channel attacks are being discovered in more general settings that violate user privacy. We build on this work by showing that the FLUSH+RELOAD L3 cache side channel from Yuval Yarom and Katrina Falkner [33] can be used to distinguish between inputs to non-cryptographic programs on multi-user systems.

We describe how input-distinguishing attacks can be discovered automatically (with human assistance) and we present three concrete attacks. Our first attack is against the Links command-line web browser. We show that an attacker can determine which of the top 100 Wikipedia pages the user visited (correct 94% of the time). Our second attack is against the Poppler PDF rendering library. We show that an attacker can determine which of a set of 127 PDFs were rendered by Poppler (correct 98% of the time). Finally, we show how an attacker can determine whether the TrueCrypt volume a user just mounted contained a hidden volume or not (correct above 80% of the time, but the attack only works on one of our two test systems).

1 Introduction

Side channels are a well-known category of vulnerabilities where an adversary is able to learn information about their victim by observing implicit, rather than explicit, channels of information. For example, information may be unintentionally leaked out through radio signals, execution time, power usage, or through the state of a computer processor's memory cache.

Side channels are most commonly used to extract secret keys from implementations of cryptography. For example, Thomas Messerges et al. were able to extract the secret RSA key from a smart card by looking at the amount of power it uses [26]. More recently, Genkin et al. were able to extract a secret RSA key just by listening to the sounds a computer makes as it is decrypting data [14].

Side channel attacks are also being used to compromise user privacy. For example, Liang Cai and Hao Chen have shown that keystrokes (including typed PIN numbers) can be reliably recovered from accelerometer data on smartphones [9]. Shuo Chen et al. showed that observing a web application's behavior reveals information about the user's input. They gave real-world examples where they could determine which medical conditions the user was searching for, what their income range was, and how they allocated their investment funds [11].

In this paper, we study additional ways of using side channels to attack the user's privacy. We show that the generic L3 cache side channel called "FLUSH+RELOAD" [33] can be applied in non-cryptography settings to meaningfully violate confidentiality. We give three example attacks, where each attack determines which of a set of inputs the user passed to a program. Specifically, we show how an attacker on a shared system can (1) determine which of the top 100 Wikipedia pages a user visited in the Links web browser, (2) determine which of 127 PDF files a user passed to the `pdftops` command, and (3) determine whether the TrueCrypt volume a user just mounted was a hidden volume or not. We chose to attack Links instead of a more popular web browser because it was easier to automate and its codebase was smaller and easier to become familiar with; we expect similar attacks to work against the more-popular browsers. Our attacks work across user accounts on a shared system; determining whether they also work across virtual machine boundaries is left for future work.

In addition to the three attacks, we also describe how the process of discovering new attacks can be partially automated.

The attacks were tested two different Intel processors: a Core2 Duo P8700 (launched Q4 2008 [19]) and a Xeon E3-1245 V2 (launched Q2 2012 [20]). Our attack tools don't work on AMD processors because their caches

are non-inclusive [33], but Lipp et al. have shown that FLUSH+RELOAD can be implemented on ARM processors with non-inclusive caches [24]. Making our attacks work on AMD processors is left for future work.

The source code to our attack tools experimentation framework is public, and we encourage you to try to reproduce our experiments on your own computer. See Appendix A for instructions to obtain the source code.

Our work is not the first to apply cache side channels to compromise privacy. We are aware of at least four other papers that use cache-based side-channel attacks to compromise privacy [16, 34, 27, 24]. Section 7 describes how our work fits in to the existing landscape of privacy-compromising cache attacks.

In the next section, we summarize the FLUSH+RELOAD side channel that all of our novel attacks are based on. Section 3 describes how FLUSH+RELOAD can be used to distinguish inputs and how we partially automated the process of discovering new attacks. Section 4 describes the software tools the attacks and experiments are built from. Section 5 describes our experimental setup. The three attacks are described and evaluated by experiment in Section 6. Section 7 highlights other attacks that use side channels to violate privacy and explains how our contributions fit in with that work. Finally, we list some ideas for future work in Section 8 and then conclude in Section 9.

2 The Flush+Reload Attack

The attacks we present use the FLUSH+RELOAD attack by Yuval Yarom and Katrina Falkner [33]. The side channel was first described by Bangerter et al. [17] where it was used on AES lookup tables to extract keys and plaintext during encryption. Yarom and Falkner realized the technique could be applied more generally to spy on the code a process executes. It was given the name "FLUSH+RELOAD" and has notably been used to break GnuPG [33] and OpenSSL [5, 32].

In this section we give a brief explanation of the FLUSH+RELOAD attack with enough detail to understand our attacks. For full details, refer to Yarom and Falkner [33].

FLUSH+RELOAD is a generic L3 (or last-level, in case the system does not have an L3 cache) cache side-channel attack. It takes advantage of executable code page sharing between processes. If Alice is the first to run a program, the operating system will load the program into physical memory. When Bob runs the same program, instead of loading a second copy into memory, the operating system will set Bob's page tables to use the copy that was loaded into memory for Alice. The result is that both Alice and Bob are using the same physical memory. This is fine, because the code is stored in read-only memory, so neither user should be able to affect the other. However, if Bob can determine which cache lines of the shared memory are present in the cache over time, he can tell which cache lines Alice's process accesses over time. This is what the FLUSH+RELOAD attack does.

Suppose an attacker knows their victim is running a certain program and would like to see which code the victim's process is running. FLUSH+RELOAD lets the attacker select a handful of cache lines to watch, called "probes," which are specified as addresses in the program's executable code. The attacker flushes those lines out of the cache, waits for a certain number of clock cycles, then times how long it takes to read those lines. If the read is fast (i.e. consistent with being in the L3 cache), it means the victim accessed the line during the waiting period. If the read is slow, the victim did not access the line. The attacker can repeat the process (flushing, then reloading) to see which code the victim process is executing over time.

The attack relies on certain assumptions. First, it assumes only one instance of the spied-on program is running at a time. If multiple instances are running, an access to a probed cache line by any instance will trigger the probe. Second, only one attacking process can run at a time on a CPU core. If a pair of attacking processes have to contend for CPU time, they will miss measurements.

In summary, the attacker specifies a few (approximately 1 to 5, where adding more probes makes the measurements less reliable) probe locations within a binary and the amount of clock cycles to wait between measurements. For example, the attack against GnuPG puts probes on the RSA modular exponentiation routines and uses a waiting time of 2048 clock cycles [33]. As output, the attacker learns the sequence of probes that were accessed during the waiting periods.

3 Using FLUSH+RELOAD to Distinguish Inputs

In the previous section we described how an attacker can use FLUSH+RELOAD against a program. In this section, we give a novel procedure for distinguishing between a set of possible inputs through the FLUSH+RELOAD side channel. Later, we present actual attacks against Links and Poppler that use this technique.

In the scenario we are interested in, the attacker knows the victim is going to run a program on some input. The attacker also knows that the input is one of a set of inputs, all of which are available to the attacker. By spying on the program with FLUSH+RELOAD, the attacker hopes to figure out which input in the set the program was run on.

For example, suppose Alice has just been diagnosed with an illness and is using Wikipedia to research it. The

attacker, Mallory, knows Alice was just diagnosed, and wants to find out which illness she has. Mallory knows that Alice will visit one of, say, 100 Wikipedia pages that are about illnesses. Mallory's goal is to find out, from the FLUSH+RELOAD probe sequence he observes, which of the 100 pages Alice visited.

The first step in the attack is to figure out where to place the FLUSH+RELOAD probes to best distinguish the input. This can be done by sifting through the program's code and finding functions whose frequency and order of execution are likely to depend heavily on the input. It is possible to do this by hand, but some automation can make it easier, a topic we return to in Section 3.1. While the side channel allows probes to be placed on any cache line, we only consider cache lines containing function entrypoints.

With the probes selected, the attack proceeds in three stages. The first stage is a training stage. The attack is most successful when the training stage is carried out on the victim's machine, but the attack still works at a lower success rate when the attacker trains on a different system. Next, the actual attack happens: the attacker spies on the victim as they execute the program on one of the inputs. Finally, the attacker uses the training data and output from the attack stage to identify the input was given to the program.

In the training stage, the attacker simply runs the FLUSH+RELOAD attack tool against themselves as they run the program on every input in the set multiple times. We refer to the number of times each input is sampled by T. As T increases, the training stage takes longer to perform, but the success rate of the attack can increase. If there are N different inputs, the attacker will be left with NT training samples. The training stage can be done either before or after the attack stage, and the attacker can re-use a training set multiple times, so they only need to train once to carry out many input identifications.

In the attack stage, the attacker runs the FLUSH+RELOAD attack tool against the victim as the victim passes one of the inputs to the program. The attack tool records the observed probe sequence.

In the identification stage, the attacker finds the probe sequence in the training set that is closest to the one obtained from the victim. Closeness is measured using the Levenshtein distance [23], which is defined as the smallest number of basic edits (single-character insertions, deletions, and replacements) needed to bring one string to the other. The attacker computes the Levenshtein distance between the victim's probe sequence and all of the training probe sequences, and the one with the smallest Levenshtein distance is assumed to correspond to the input the victim passed to the program.

Probe sequences are represented by strings of single-character probe names. An example is given in Figure 1.

```
BDBABABABABABABABABABABABCABABCBABCBCABC
ABABABCBABCABCBCBDABABACACADBABDBCBABCBA
BABDBCBABDBCABDBCABDBCABCBCBCBABCBCBABAB
DBABABDABDBCABDABABCABDBCBCBABCBCBABCBCB
ABCABCABDABDCBCABCABCBABDABCABDBCABDBCBA
```

Figure 1: The first 200 bytes (of 24,984) of a condensed observed probe sequence while visiting the Facebook Wikipedia page in Links. From run links/0015. `A` corresponds to `kill_html_stack_item()`, `B` to `html_stack_dup()`, `C` to `html_a()`, and `D` to `parse_html()`.

Before computing the Levenshtein distance, the probe sequences are first condensed by collapsing sequentially repeated hits of the same probe down into a single hit of that probe, and then the whole string is truncated to 1000 characters. This is done to make the identification stage faster, since the algorithm we use to compute the Levenshtein distance has quadratic running time.

3.1 Using Automation to Find Probes

In our experience, the hardest part of creating a new input distinguishing attack is finding the set of probes to spy on. We want to find a set of functions whose order of execution best exhibits the differences between the inputs. Statically, one way to do this is to read the program's code, understand it, and then manually select some functions that are heavily involved in processing the input. This takes time and effort.

In this section, we present a method for partially automating probe discovery. We describe a tool which takes as input the victim program and two distinct victim-program inputs. It finds a small subset of the victim program's functions that are good at distinguishing the victim-program inputs when spied on. The tool requires symbols, but not source code: as long as the attacker has access to a non-stripped binary, they can use the tool to find a good probe set without having a deep understanding of the program. Once a good set of probes is discovered, symbols are no longer necessary, and the attack will work without them. This tool works at the function level of granularity instead of the cache-line level of granularity to keep performance reasonable (each candidate probe must be tested individually) and because functions have human-readable names which assists with the manual filtering (see below).

The tool first looks at the symbols in the binary to list all of the functions. Next, some human intervention is needed to narrow down the list of functions into a smaller set that are more likely to depend on the input. This filtering can be as simple as keeping all functions whose names

contain "`html`" to get the list of html-parsing functions, as we have done for Links, or by looking for functions whose names begin with "`Gfx::op`" to get the list of functions responsible for PDF commands, as we have done for Poppler.

Once the list of functions has been reduced to a manageable size (e.g. less than 100 functions), each potential probe is tested individually and automatically. A candidate probe is tested by placing a GDB breakpoint on the first instruction in the function and counting the number of times it gets executed as the program is run. This is done three times for each probe: twice on one input, giving counts c_1 and c_2, and once on a second, different, input, giving the count c_3. A score is given to each probe, equal to $|c_3 - c_1| - |c_2 - c_1|$. This rewards functions whose execution counts are stable on the same input but vary on different inputs, meaning they likely depend on the input. The list of candidate probes are sorted by their scores, and all but the top 10 are rejected.

The reduced and sorted list of functions is then used to generate all possible sets of size 4. Sets that contain functions that are within 3 cache lines of each other are immediately removed. If two probes are too close together, the CPU's instruction prefetch may trigger one whenever another one is actually executed, in which case there is no point having both probes. The remaining sets are sorted by the sums of their rank in the input list (which was sorted by score), and this sorted list is given to the user. In our attacks, we always used the first probe set in this output list.

4 Attack Implementation

The attacks are built on the following tools.

- A FLUSH+RELOAD attack tool, written in C, takes a binary and probe addresses with single-character names as arguments. When probes are hit, their single-character names are printed to standard output, with pipe characters separating the waiting periods. This tool is a re-implementation of the tool the original paper's ([33]) authors provided us.
- A Ruby class that makes it easy for Ruby scripts to run the attack tool and monitor its output. It provides a callback whenever there is a burst of probe hits. This lets the attack script react to events (such as the entire execution of a command). Our attacks are implemented as Ruby scripts that use this class.
- A Ruby implementation of the semi-automated attack discovery procedure described in Section 3.1. The tool takes as input a list of executable code addresses in a program and two different inputs. It uses the two inputs to test the quality of each candidate probe, and outputs a set of probes that are likely able to distinguish the inputs.
- A Ruby implementation of the attack procedure described in Section 3. There is one program for each stage in the attack.

The source code for all of these programs is available online; see Appendix A. Next, we describe how we experimentally evaluated our attacks using these tools.

5 Experiment Setup

We ran experiments on two systems. The first system is a laptop; the second system is a dedicated server hosting a low-traffic website and operating as a relay for the Tor network. The specifications of these systems are given in Table 1. We will refer to these systems as System 1 and System 2, respectively.

To test the input distinguishing attacks, we first perform the training stage, taking T samples of each input. Then, for every input in the set of size N, the vulnerable program is run on the input S times. The probe sequences from each of the SN runs are put through the identification stage independently, and we check how many of the identifications are correct.

All of the experiments have been automated so that they are easy to repeat. The source code and data from all experiment runs have been published so that other researchers can verify and reproduce our work (see Appendix A).

All experiments were run with low system load. The experiments are all run within the same user account for simplicity, but we confirmed independently that the attacks work when the attacker is on one account and the spy is on another.

6 Attacks

We present three attacks. The first attack lets an attacker tell which of the top 100 Wikipedia pages the victim visited using the Links web browser. The second attack does the same for the Poppler PDF rendering library, distinguishing between 127 PDF transcripts of 2014 Canadian parliamentary debates. The Links and Poppler attacks both use the input distinguishing procedure exactly as described in Section 3. The third attack, which is implemented differently for reasons explained later, determines whether a mounted TrueCrypt volume contained a hidden volume or not.

6.1 Links

Links is a command-line web browser. We are interested in learning which web page a user is visiting, out of

	System 1	System 2
Use	Laptop	Web server
OS	Arch Linux (February 2015)	Debian Wheezy
CPU	Intel Core2 Duo P8700 2.53GHz (2 cores)	Intel Xeon E3-1245 V2 3.40GHz (4 cores, 8 threads)
RAM	4GiB DDR2 800MHz	32GiB DDR3 1333MHz
L1 Cache	2x32KiB Instructions + 2x32KiB Data	4x32KiB Instructions + 4x32KiB Data
L2 Cache	3MiB Shared Unified	4x256KiB Unified
L3 Cache	None	8MiB Shared Unified

Table 1: System specifications. Cache specifications were obtained from `cpu-world.com`. System 1 does not have a L3 cache, but FLUSH+RELOAD works with its L2 cache as it is shared between cores.

a set of known pages. We found that it is possible to reliably distinguish between the top 100 Wikipedia pages of 2013 [2].

To find the Links probes, we ran the automatic probe finding tool on the HTML parsing functions inside Links. The names of these functions all contain the string "html", so we filtered the list of all functions for "html" and gave the resulting list of 77 functions to the probe finding tool. The "Bird" and "Feather" Wikipedia pages were chosen arbitrarily as inputs to the probe-finding tool. The tool gave us the following probe set:

- `html_stack_item()`
- `html_stack_dup()`
- `html_a()`
- `parse_html()`

On System 1 with $T = 10$ training samples and $S = 10$ trial runs,[1] the correct page was identified 940 times out of 1000 (94%). In this experiment, the average time it took to identify the page from the training sequences and the victim sequence was 4.1 seconds. All pages in the set were correctly identified at least twice out of the ten runs, and most were correctly identified all 10 times. The distribution is plotted in Figure 2.

A second run[2] on System 1 with $T = 10$ training samples and $S = 1$ trial run saw the correct page being identified 95 times out of 100 (95%). Again it took 4.1 seconds to perform an identification, on average.

On System 2 with $T = 10$ training samples and $S = 1$ trial run,[3] the correct page was identified 98 times out of 100. Recovery took 136 seconds on average. The identification took much longer on System 2 despite having a faster processor because System 2 was configured to use a pure Ruby implementation of a Levenshtein distance algorithm instead of the fast native implementation that

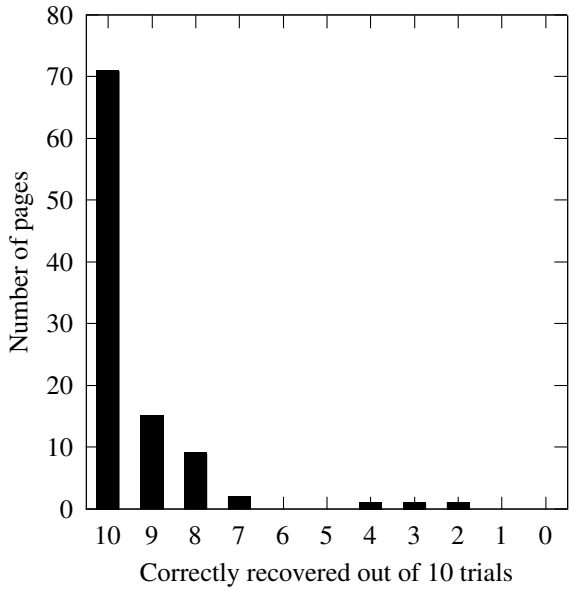

Figure 2: The distribution of success rates for individual pages. The x axis is the number of times the page was correctly identified out of ten trials; the y axis is the number of pages that were correctly identified that many times. Most pages were correctly identified in all ten trials.

System 1 was was configured to use. With $T = 10$ training samples and $S = 10$ trial runs,[4] the correct page was identified 977 times out of 1000 (98%). As on System 1, most of the pages were correctly identified 9 or 10 times out of 10.

A plot of the Levenshtein distances involved in a successful identification are shown in Figure 3, and those from an unsuccessful identification are shown in Figure 4.

All of the Wikipedia pages in the set have distinct lengths; it is reasonable to ask if this attack is only distinguishing the pages by their lengths. We know this is not the case, however, because the probe sequences are truncated to 1000 characters before the Levenshtein distance is computed, and in all of the experiment runs mentioned above, most of the training samples have lengths

[1] links/0014; interpretation of these labels is explained in Appendix A.

[2] links/0013

[3] links/0015

[4] links/0016

above 1000 and thus most of the comparisons are between constant-length 1000-character strings. It is therefore the order of the probe hits that matters, and we are really learning information about the content of the page, not just its length.

We were curious whether the attack would work if the training were done on a different system. To simulate this, we used the training set from an experiment[5] on System 1 to identify victim samples from another experiment[6] on System 2. The result was that 76% of the pages were identified correctly. Using the training set from System 2 and the victim samples from System 1, the identification was successful 82% of the time. So it is better to train on the victim's machine, but the attack can still work when the training is done on a different system.

Before we built the probe finding tool, we ran experiments with a different set of probes that we found manually. We chose these probes by trial and error, looking through the Links source code to find functions whose frequency and order of execution should depend on the contents of Wikipedia pages. These are:

- `parse_html()`
- `html_stack_dup()`
- `html_h()`
- `html_span()`

Note that two of the manually-selected probes were re-discovered by the automation tool.

With these probes, we saw 76% correct classification on System 1 with $T = 5$ training samples and $S = 1$ trial run.[7] On System 2 with $T = 5$ training samples and $S = 10$ trial runs[8] we saw 88% correctness. With $T = 10$ training samples and $S = 1$ trial run we saw 91%.[9] This suggests that using more training samples increases the success rate of the attack, and that the automatically discovered probes are just as good or slightly better than the ones we found manually.

6.2 Poppler

Poppler is a PDF rendering library that gets used in software such as Evince and LibreOffice. For ease of automation, we attacked the `pdftops` program, which converts PDF files into PostScript files using Poppler. As our input set, we used 127 transcripts of 2014 parliamentary debates made available by the Canadian government [1].

[5] links/0014

[6] links/0016

[7] links/0002. This experiment was run before we started truncating to 1000 characters, so the full strings were compared.

[8] links/0010

[9] links/0005. This experiment was also run before we started truncating to 1000 characters.

Unlike Links, we did not have a working attack against Poppler before we used the probe finding tool. The only human input in the creation of this attack was the idea to look at the set of functions that execute PDF commands, an intuitively obvious thing to try.

We used the automatic probe finding tool to find the best probes amongst the functions responsible for executing PDF commands. These functions, of which there are 77, are easily identified because their names begin with "`Gfx::op`." The probe finding tool returned the following set of probes when we ran it with the input files `HAN040-E.PDF` and `HAN050-E.PDF` (which we chose arbitrarily).

- `Gfx::opShowSpaceText(Object*, int)`
- `Gfx::opTextMoveSet(Object*, int)`
- `Gfx::opSetFont(Object*, int)`
- `Gfx::opTextNextLine(Object*, int)`

On System 1 with $T = 5$ training samples and $S = 10$ trial runs,[10] the correct PDF was identified 1258 times out of 1270 (99%). All of the PDFs were reliably identifiable: all but one were correctly identified 9 or 10 times out of 10; the other one was identified correctly 8 times. In a repeat run on same system with $T = 5$ training samples and $S = 1$ trial runs,[11] the correct PDF was identified 124 times out of 127 (98%).

On System 2 with $T = 5$ training samples and $S = 1$ trial run,[12] the correct PDF was identified 126 times out of 127 (99%). With $T = 5$ training samples and $S = 10$ trial runs,[13] the PDF was correctly identified 1260 times out of 1270 (99%). Again, all but one were identified 9 or 10 times out of 10, except for one that was identified correctly only 8 times.

Using a training set[14] created on System 1 we correctly identified the input from 70% of probe sequences recorded on System 2.[15] Using the training set from System 2, we correctly identified the input from 69% of the probe sequences recorded on System 1.

To test how well the probe finding tool works in the absence of any human guidance, we tried running it on the list of all 5,397 functions in the Poppler library. The result is the following set of probes:

- `gmallocn()`
- `PSOutputDev::writePSString(GooString*)`
- `PSOutputDev::drawString(GfxState*, GooString*)`

[10] poppler/0003

[11] poppler/0006

[12] poppler/0001

[13] poppler/0007

[14] poppler/0003

[15] poppler/0007

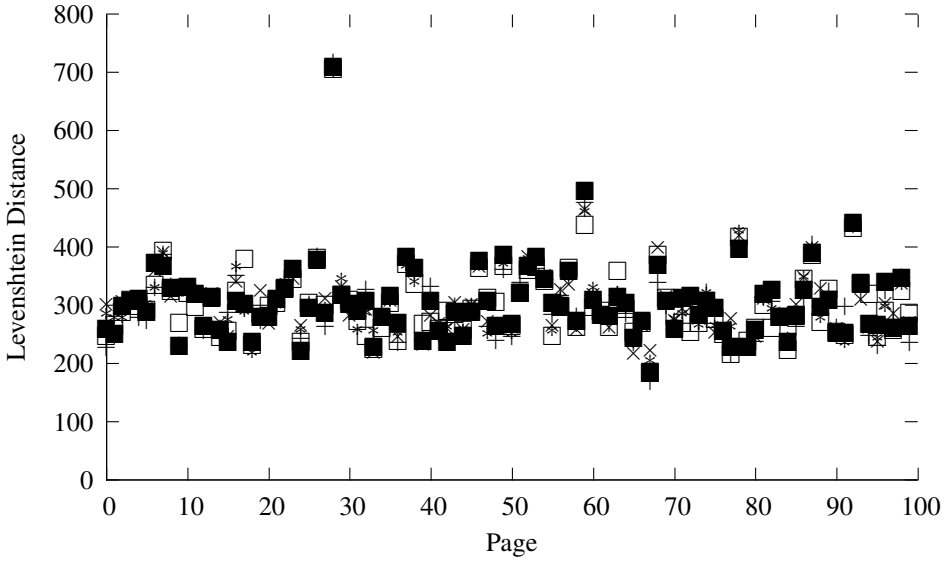

Figure 3: A successful identification. The Levenshtein distance between the training samples and a recording of the victim visiting the YouTube Wikipedia page. The shortest distance is visible at mark 68 on the page axis which corresponds to a YouTube training sample. The outlier at mark 29 corresponds to a disambiguation page that has a different format from the usual Wikipedia page. The different shapes in a column represent the five training samples of that page. The order on the page axis is not meaningful.

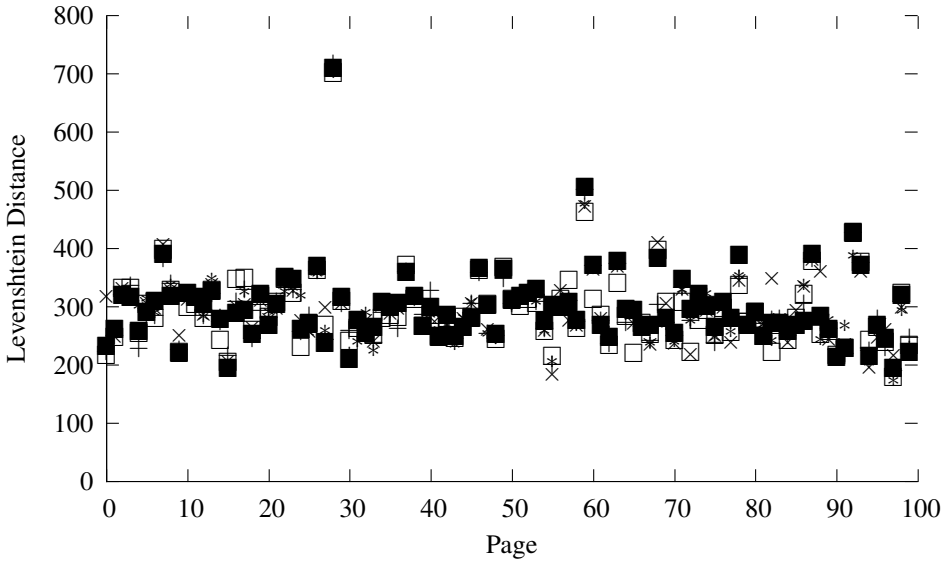

Figure 4: A failed identification. The Levenshtein distance between the training samples and a recording of the victim visiting the Nicki Minaj Wikipedia page. The shortest distance (97 on the page axis) corresponds to a training sample of the Eminem Wikipedia page. The Nicki Minaj training samples still stand out (55 on the page axis). The different shapes in a column represent the five training samples of that page. The order on the page axis is not meaningful.

- `PSOutputDev::updateTextShift(GfxState*, double)`

We ran an experiment[16] with these probes and $T = 5$ training samples and $S = 1$ trial run on System 1, and the result was that the correct PDF was only identified 21 times out of 127, only 17%. This is much worse than with the other set of probes, but is still better than random guessing. With these probes, the FLUSH+RELOAD tool ran into an error condition, where the CPU's `RDTSC` counter changes non-monotonically, much more frequently than it did with the other probes, and that is probably the reason for the poorer result.

6.3 TrueCrypt

TrueCrypt is a popular disk encryption utility that supports storing encrypted filesystems in files called TrueCrypt volumes. TrueCrypt gives users the option to place a hidden volume inside a normal volume. Given the passphrase to the outer normal volume, it is not supposed to be possible to determine whether an inner hidden volume exists. This is to protect the user in case they are coerced into revealing their passphrase – they can reveal the passphrase to the outer volume and the contents of the hidden volume will be safe.

Our attack watches the victim mount a TrueCrypt volume and determines whether they mounted a hidden volume or a normal one. This attack is not implemented the same way as the Links and Poppler attacks. Our automatic probe discovery tool does not work against TrueCrypt, since our tool only supports debugging newly launched processes, and TrueCrypt's volume mounting code runs in a background process. Even if the required functionality were added to the tool, we would not expect it to work with TrueCrypt since the difference in code execution between mounting a normal volume and mounting a hidden volume is minimal.

6.3.1 Attack Implementation

TrueCrypt is written in C++ and has two classes defining the layout of normal and hidden volumes. They are `VolumeLayoutV2Normal` and `VolumeLayout-V2Hidden`, respectively. There are three other classes for the operating system encryption layout and volume layouts from older versions, making five volume layout classes in total.

When TrueCrypt mounts a volume, it instantiates all five layout classes and tries to decrypt the volume using each one. It has to do this because, by design, TrueCrypt volumes are supposed to be indistinguishable from random data, so there is no way to tell which volume layout is the right one in advance; the only way is to try to decrypt the volume and see if it works.

Each class implements a `GetDataSize()` method. This method is only called on a layout object after the volume has been successfully decrypted using that layout, making it a good candidate for a FLUSH+RELOAD probe.

At first we tried to place probes on both `VolumeLayoutV2Normal`'s `GetDataSize()` and `VolumeLayoutV2Hidden`'s `GetDataSize()`. This did not work because in the binary, the hidden method immediately follows the normal method, and instruction prefetching triggers the hidden probe when the normal method gets executed.

To work around that problem, we placed two FLUSH+RELOAD probes. The first is placed on the entry point to the TrueCrypt binary, so that we can tell when the user runs a TrueCrypt command. The second is placed in `VolumeLayoutV2Normal`'s `GetDataSize()`, which will get hit once when the TrueCrypt binary is loaded, and then once again only if the volume is normal.

To find out whether the volume is normal or hidden, the attacker records the probe sequence as the victim mounts the volume. They check if the probe sequence ends in the `VolumeLayoutV2Normal` probe. If it does, the volume was normal. If not, the volume was hidden (or what the attacker captured was not the result of a mount command; we assume the attacker knows that the user is mounting a volume).

6.3.2 Experiment

Our experiment creates two 1MB TrueCrypt volumes. One is a normal volume, the other contains a hidden volume; both are protected by the same passphrase.

The experiment assumes the attacker knows the TrueCrypt command was run to mount a volume, and not some other task (like unmounting a volume). This could be done in practice by looking at the process list to see which command-line options were passed to TrueCrypt.

The experiment starts the attack tool on the TrueCrypt binary. After waiting for the tool to start, it randomly mounts either the normal or hidden volume with TrueCrypt's command-line tool. It stops the attack tool and checks if the last probe hit was the `GetDataSize()` probe. If it was, it guesses that the volume is normal. If not, it guesses that the volume is hidden.

In one run[17] of the experiment on System 1 with $S = 500$ trial runs, the guess was right 416 times, or 83%. (Recall that we call the number of trials in a run of the experiment S.) Of the 255 trials with normal volumes, the guess was right 184 times. Of the 245 trials with hidden volumes, the guess was right 232 times. So the error is skewed towards mistakenly believing that the volume is

[16] poppler/0004

[17] truecrypt/0001

hidden. By spying as the volume is mounted multiple times, the confidence can be increased. In one experiment[18] where the attacker is allowed to watch the volume get mounted 3 times and then take the majority of their decisions, they decide correctly 93 times out of 100. In a longer run[19] with majority-of-three, the attacker decides correctly 476 out of 500 times (95%). 237/256 (93%) of the normal samples were correct, 239/244 (93%) of the hidden samples were correct.

We could not reproduce the attack on System 2. We believe this is due to differences in the way the two CPUs do instruction prefetching. The code for normal volumes and hidden volumes is very close together, so the attack is easily foiled by prefetching. System 2's processor seems to prefetch backwards and read the normal volume `GetDataSize()` code when the hidden volume `GetDataSize()` is executing. This makes it impossible to distinguish the two cases on System 2 with our choice of probes. We were unable to find another choice of probes that would allow it to work.

We only tried the attack with TrueCrypt's command-line interface. The attack should extend to the graphical interface, since both interfaces are front ends to the same volume mounting code.

7 Related Work

Our attacks are based on the FLUSH+RELOAD attack [33], which in turn is based on work by Bangerter et al. where it was used to break an implementation of AES [17]. It has since been applied to GnuPG [33], OpenSSL [5, 32], and other cryptography libraries ([8] for example).

There is a vast body of literature on attacking cryptographic software and hardware with side channels. There are far too many examples to list here; we refer to the interesting cases of extracting RSA keys via power analysis [26] and sound recording [14] as well as other FLUSH+RELOAD attacks [33, 5, 32].

Continuing the trend of using cache side channels to violate users' privacy, our work further confirms that cache side-channel attacks can do worse than just breaking cryptography software. Other privacy-compromising side-channel attacks attacks are those in [34] which determines the number of distinct items in an e-commerce user's shopping cart, [16] which infers keystroke information on PCs, [27] which infers network and mouse activity from inside a sandboxed web page, and [24] which infers information about user input on Android phones (e.g. when the user swipes vs. taps on their phone). Cache side channels have also been used to break kernel address space layout randomization [18].

[18] truecrypt/0005
[19] truecrypt/0006

Readers interested in defending against last-level cache attacks like FLUSH+RELOAD should see the survey of defense options in the Countermeasures section of [16] as well as other works like CATalyst [25].

There is a growing body of work showing that other kinds of side channels are successfully breaking privacy, too. Here we highlight some of that work.

- Some web applications leak information about the user's input through their requests' size and timing, even when they are sent over an encrypted connection [6, 11].
- Malicious web pages can read text inside an `iframe` by exploiting differences in the time it takes to run SVG filters [29, 3].
- Memory deduplication features in new versions of Microsoft Windows make it possible for a malicious web page to check "whether the user has specific websites currently opened" [15] and, when combined with Rowhammer [22], to execute arbitrary code [7].
- Timing variations in databases make it possible to extract indexed records [13].
- Variable bit rate encoding can leak words spoken over encrypted VoIP links [30].
- In an Android app, the UI state can be inferred through side channels in the GUI framework [10] and through interrupt side channels [12].
- An attacker can learn a victim's Internet traffic volume, as well as individual packet times by exploiting a side channel in router scheduling algorithms [21].
- A smartphone app can infer what the user is typing from accelerometer and gyroscope measurements [28, 9].
- Text can be recovered from the sounds dot-matrix printers make [4].
- In an attack model where the operating system is malicious but a hypervisor and an application are untrusted, documents and images can be recovered through a page fault side channel [31].

We believe our attacks represent additional steps towards understanding the capabilities and limitations of using the FLUSH+RELOAD attack to compromise privacy, and we hope they will motivate more research on the topic.

8 Future Work

There are many more input distinguishing attacks waiting to be discovered. It should be easy to use the tools we developed to find new attacks, for example against a more popular web browser.

Quantitatively, our attacks only need to extract a small amount of information through the side channel in order to work. For TrueCrypt, we are only extracting one bit of information. The Links and Poppler attacks only need to extract a number of bits logarithmic in the number of inputs we want to distinguish between (6.6 bits for 100 pages, 7.0 for 127). We should find out how much information FLUSH+RELOAD can extract from non-cryptographic programs. In particular, can FLUSH+RELOAD extract previously-unknown user input from a program in any plausible scenario?

Some virtual machine hypervisors deduplicate pages between virtual machines. When this is accomplished by finding pairs of memory pages with the same content, and an attacker can predict the content of a target memory page, the attacker can gain the ability to run a FLUSH+RELOAD attack on the target page by creating a duplicate of it in memory that they control. It's an open question if this leads to better privacy-compromising attacks than spying on the program code alone.

Aside from the possibility of developing better attacks, we've left two questions unexplored. The first is whether or not these attacks can be made to work across virtual machine boundaries, and the second is whether or not they can be made to work on processors with exclusive cache architectures, like AMD. Lipp et al.'s work on ARM processors suggests that it is possible [24].

9 Conclusion

Classically, side-channel attacks are used to extract encryption keys from software and hardware implementations of cryptography. More recently, side-channel attacks are being used to compromise privacy in more general settings. We presented three attacks that extend this work, along with accompanying tools and an automation framework.

Our attacks let an attacker (1) determine which of the top 100 Wikipedia pages a victim visited with the Links web browser, (2) determine which of the 127 debates in the 2014 Canadian parliament a victim transcoded with the `pdftops` command, and (3) determine whether a TrueCrypt volume contains a hidden volume when it is mounted.

Because we attacked relatively obscure programs, we expect the set of vulnerable users to be small. However, our attacks are part of a growing body of work that applies side-channel analysis to more than just breaking cryptography; we hope others will build better attacks on top of our work.

Acknowledgment

I would like to thank Prof. John Aycock for supervising and assisting with this project when I was working on it for my undergraduate thesis at the University of Calgary. Our discussions helped guide the project towards the results presented here. He also contributed edits and clarifications to an earlier version of this paper, for which I am grateful. John Aycock's research is supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada.

I would also like to thank the three anonymous reviewers who provided valuable feedback when we submitted this paper to USENIX WOOT 2015.

References

[1] Debates (Hansard). `http://www.parl.gc.ca/HouseChamberBusiness/ChamberSittings.aspx?View=H&Language=E`. Accessed: 2015-05-15.

[2] Most viewed articles on Wikipedia 2013. `https://web.archive.org/web/20141215053145/http://tools.wmflabs.org/wikitrends/2013.html`. Accessed: 2015-01-19.

[3] ANDRYSCO, M., KOHLBRENNER, D., MOWERY, K., JHALA, R., LERNER, S., AND SHACHAM, H. On subnormal floating point and abnormal timing. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 623–639.

[4] BACKES, M., DÜRMUTH, M., GERLING, S., PINKAL, M., AND SPORLEDER, C. Acoustic side-channel attacks on printers. In *USENIX Security Symposium* (2010), pp. 307–322.

[5] BENGER, N., VAN DE POL, J., SMART, N. P., AND YAROM, Y. "Ooh aah... just a little bit": A small amount of side channel can go a long way. *IACR Cryptology ePrint Archive 2014* (2014), 161.

[6] BORTZ, A., AND BONEH, D. Exposing private information by timing web applications. In *Proceedings of the 16th international conference on World Wide Web* (2007), ACM, pp. 621–628.

[7] BOSMAN, E., RAZAVI, K., BOS, H., , AND GIUFFRIDA, C. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)* (San Jose, CA, USA, May 2016), IEEE.

[8] BRUINDERINK, L. G., HÜLSING, A., LANGE, T., AND YAROM, Y. Flush, gauss, and reload–a cache attack on the bliss lattice-based signature scheme. *exchange 6*, 18 (2016), 24.

[9] CAI, L., AND CHEN, H. On the practicality of motion based keystroke inference attack. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing* (Berlin, Heidelberg, 2012), TRUST'12, Springer-Verlag, pp. 273–290.

[10] CHEN, Q. A., QIAN, Z., AND MAO, Z. M. Peeking into your app without actually seeing it: UI state inference and novel Android attacks. In *Proceedings of the 23rd USENIX conference on Security Symposium* (2014), USENIX Association, pp. 1037–1052.

[11] CHEN, S., WANG, R., WANG, X., AND ZHANG, K. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *2010 IEEE Symposium on Security and Privacy* (2010), IEEE, pp. 191–206.

[12] DIAO, W., LIU, X., LI, Z., AND ZHANG, K. No pardon for the interruption: New inference attacks on android through interrupt timing analysis.

[13] FUTORANSKY, A., SAURA, D., AND WAISSBEIN, A. The ND2DB attack: Database content extraction using timing attacks on the indexing algorithms. In *WOOT* (2007).

[14] GENKIN, D., SHAMIR, A., AND TROMER, E. RSA key extraction via low-bandwidth acoustic cryptanalysis. *IACR Cryptology ePrint Archive 2013* (2013), 857.

[15] GRUSS, D., BIDNER, D., AND MANGARD, S. Practical memory deduplication attacks in sandboxed javascript. In *European Symposium on Research in Computer Security* (2015), Springer, pp. 108–122.

[16] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium (USENIX Security 15)* (2015), pp. 897–912.

[17] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache games–bringing access-based cache attacks on AES to practice. In *2011 IEEE Symposium on Security and Privacy* (2011), IEEE, pp. 490–505.

[18] HUND, R., WILLEMS, C., AND HOLZ, T. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 191–205.

[19] INTEL. Intel Core2 Duo Processor P8700 (3M Cache, 2.53 GHz, 1066 MHz FSB). `http://ark.intel.com/products/37006/Intel-Core2-Duo-Processor-P8700-3M-Cache-2_53-GHz-1066-MHz-FSB`. Accessed: 2016-07-19.

[20] INTEL. Intel Xeon Processor E3-1245 v2 (8M Cache, 3.40 GHz). `http://ark.intel.com/products/65729/Intel-Xeon-Processor-E3-1245-v2-8M-Cache-3_40-GHz`. Accessed: 2016-07-19.

[21] KADLOOR, S., GONG, X., KIYAVASH, N., TEZCAN, T., AND BORISOV, N. Low-cost side channel remote traffic analysis attack in packet networks. In *2010 IEEE International Conference on Communications (ICC)* (2010), IEEE, pp. 1–5.

[22] KIM, Y., DALY, R., KIM, J., FALLIN, C., LEE, J. H., LEE, D., WILKERSON, C., LAI, K., AND MUTLU, O. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *ACM SIGARCH Computer Architecture News* (2014), vol. 42, IEEE Press, pp. 361–372.

[23] LEVENSHTEIN, V. I. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet physics doklady* (1966), vol. 10, p. 707.

[24] LIPP, M., GRUSS, D., SPREITZER, R., AND MANGARD, S. Armageddon: Last-level cache attacks on mobile devices. *arXiv preprint arXiv:1511.04897* (2015).

[25] LIU, F., GE, Q., YAROM, Y., MCKEEN, F., ROZAS, C., HEISER, G., AND LEE, R. B. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2016), IEEE, pp. 406–418.

[26] MESSERGES, T. S., DABBISH, E. A., AND SLOAN, R. H. Power analysis attacks of modular exponentiation in smartcards. In *Cryptographic Hardware and Embedded Systems* (1999), Springer, pp. 144–157.

[27] OREN, Y., KEMERLIS, V. P., SETHUMADHAVAN, S., AND KEROMYTIS, A. D. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 1406–1418.

[28] OWUSU, E., HAN, J., DAS, S., PERRIG, A., AND ZHANG, J. ACCessory: password inference using accelerometers on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications* (2012), ACM, p. 9.

[29] STONE, P. Pixel perfect timing attacks with HTML5. `http://www.contextis.com/documents/2/Browser_Timing_Attacks.pdf`, 2013. Accessed: 2015-05-19.

[30] WHITE, A. M., MATTHEWS, A. R., SNOW, K. Z., AND MONROSE, F. Phonotactic reconstruction of encrypted VoIP conversations: Hookt on fon-iks. In *2011 IEEE Symposium on Security and Privacy (SP)* (2011), IEEE, pp. 3–18.

[31] XU, Y., BAILEY, M., JAHANIAN, F., JOSHI, K., HILTUNEN, M., AND SCHLICHTING, R. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop* (2011), ACM, pp. 29–40.

[32] YAROM, Y., AND BENGER, N. Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. *IACR Cryptology ePrint Archive 2014* (2014), 140.

[33] YAROM, Y., AND FALKNER, K. E. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. *IACR Cryptology ePrint Archive 2013* (2013), 448.

[34] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-tenant side-channel attacks in PaaS clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 990–1003.

A Reproducing this Work

The attack tools, experiment implementations, and experiment data are all available for download on the author's website: `https://defuse.ca/BH2016`.

Throughout this paper we have referred to experiment runs by the name of the experiment followed by a four-digit run number. The experiment name corresponds to a directory name in the archive, and the run number corresponds to a subdirectory of that directory. For example, the truecrypt/0003 data can be found in `experiments/truecrypt/runs/0003`.