Flip Feng Shui: Hammering a Needle in the Software Stack

Kaveh Razavi, Ben Gras, and Erik Bosman, ***Vrije Universiteit Amsterdam;***
Bart Preneel, ***Katholieke Universiteit Leuven;*** **Cristiano Giuffrida and Herbert Bos,**
Vrije Universiteit Amsterdam

https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/razavi

This paper is included in the Proceedings of the 25th USENIX Security Symposium

August 10–12, 2016 • Austin, TX

ISBN 978-1-931971-32-4

Open access to the Proceedings of the 25th USENIX Security Symposium is sponsored by USENIX

Flip Feng Shui: Hammering a Needle in the Software Stack

Kaveh Razavi*
Vrije Universiteit Amsterdam

Ben Gras*
Vrije Universiteit Amsterdam

Erik Bosman
Vrije Universiteit Amsterdam

Bart Preneel
Katholieke Universiteit Leuven

Cristiano Giuffrida
Vrije Universiteit Amsterdam

Herbert Bos
Vrije Universiteit Amsterdam

* Equal contribution joint first authors

Abstract

We introduce Flip Feng Shui (FFS), a new exploitation vector which allows an attacker to induce bit flips over *arbitrary* physical memory in a *fully controlled way*. FFS relies on hardware bugs to induce bit flips over memory and on the ability to surgically control the physical memory layout to corrupt attacker-targeted data anywhere in the software stack. We show FFS is possible today with very few constraints on the target data, by implementing an instance using the *Rowhammer bug* and *memory deduplication* (an OS feature widely deployed in production). Memory deduplication allows an attacker to reverse-map any physical page into a virtual page she owns as long as the page's contents are known. Rowhammer, in turn, allows an attacker to flip bits in controlled (initially unknown) locations in the target page.

We show FFS is extremely powerful: a malicious VM in a practical cloud setting can gain unauthorized access to a co-hosted victim VM running OpenSSH. Using FFS, we exemplify end-to-end attacks breaking OpenSSH public-key authentication, and forging GPG signatures from trusted keys, thereby compromising the Ubuntu/Debian update mechanism. We conclude by discussing mitigations and future directions for FFS attacks.

1 Introduction

The demand for high-performance and low-cost computing translates to increasing complexity in hardware and software. On the hardware side, the semiconductor industry packs more and more transistors into chips that serve as a foundation for our modern computing infrastructure. On the software side, modern operating systems are packed with complex features to support efficient resource management in cloud and other performance-sensitive settings.

Both trends come at the price of reliability and, inevitably, security. On the hardware side, components are increasingly prone to failures. For example, a large fraction of the DRAM chips produced in recent years are prone to bit flips [34, 51], and hardware errors in CPUs are expected to become mainstream in the near future [10, 16, 37, 53]. On the software side, widespread features such as memory or storage deduplication may serve as side channels for attackers [8, 12, 31]. Recent work analyzes some of the security implications of both trends, but so far the attacks that abuse these hardware/software features have been fairly limited—probabilistic privilege escalation [51], in-browser exploitation [12, 30], and selective information disclosure [8, 12, 31].

In this paper, we show that an attacker abusing modern hardware/software properties can mount much more sophisticated and powerful attacks than previously believed possible. We describe Flip Feng Shui (FFS), a new exploitation vector that allows an attacker to induce bit flips over *arbitrary* physical memory in a *fully controlled way*. FFS relies on two underlying primitives: (i) the ability to induce bit flips in controlled (but not predetermined) physical memory pages; (ii) the ability to control the physical memory layout to reverse-map a target physical page into a virtual memory address under attacker control. While we believe the general vector will be increasingly common and relevant in the future, we show that an instance of FFS, which we term dFFS (i.e, deduplication-based FFS), can already be implemented on today's hardware/software platforms with very few constraints. In particular, we show that by abusing Linux' memory deduplication system (KSM) [6] which is very popular in production clouds [8], and the widespread Rowhammer DRAM bug [34], an attacker can *reliably* flip a single bit in *any* physical page in the software stack with known contents.

Despite the complete absence of software vulnerabilities, we show that a practical Flip Feng Shui attack can have devastating consequences in a common cloud setting. An attacker controlling a cloud VM can abuse

memory deduplication to seize control of a target physical page in a co-hosted victim VM and then exploit the Rowhammer bug to flip a particular bit in the target page in a fully controlled and reliable way without writing to that bit. We use dFFS to mount end-to-end corruption attacks against OpenSSH public keys, and Debian/Ubuntu update URLs and trusted public keys, all residing within the page cache of the victim VM. We find that, while dFFS is surprisingly practical and effective, existing cryptographic software is wholly unequipped to counter it, given that "*bit flipping is not part of their threat model*". Our end-to-end attacks completely compromise widespread cryptographic primitives, allowing an attacker to gain full control over the victim VM.

Summarizing, we make the following contributions:

- We present FFS, a new exploitation vector to induce hardware bit flips over arbitrary physical memory in a controlled fashion (Section 2).
- We present dFFS, an implementation instance of FFS that exploits KSM and the Rowhammer bug and we use it to bit-flip RSA public keys (Section 3) and compromise authentication and update systems of a co-hosted victim VM, granting the attacker unauthorized access and privileged code execution (Section 4).
- We use dFFS to evaluate the time requirements and success rates of our proposed attacks (Section 5) and discuss mitigations (Section 6).

The videos demonstrating dFFS attacks can be found in the following URL:

`https://vusec.net/projects/flip-feng-shui`

2 Flip Feng Shui

To implement an FFS attack, an attacker requires a *physical memory massaging primitive* and a *hardware vulnerability* that allows her to flip bits on certain locations on the medium that stores the users' data. Physical memory massaging is analogous to virtual memory massaging where attackers bring the virtual memory into an exploitable state [23, 24, 55], but instead performed on physical memory. Physical memory massaging (or simply *memory massaging*, hereafter) allows the attacker to steer victim's sensitive data towards those physical memory locations that are amenable to bit flips. Once the target data land on the intended vulnerable locations, the attacker can trigger the hardware vulnerability and corrupt the data via a controlled bit flip. The end-to-end attack allows the attacker to flip *a bit of choice* in *data of choice anywhere* in the software stack in a controlled fashion. With some constraints, this is similar to a typical arbitrary memory write primitive used for software exploitation [15], with two key differences: (i) the end-to-end attack requires no software vulnerability; (ii) the attacker can overwrite arbitrary physical (not just virtual) memory on the running system. In effect, FFS transforms an underlying hardware vulnerability into a very powerful software-like vulnerability via three fundamental steps:

1. *Memory templating*: identifying physical memory locations in which an attacker can induce a bit flip using a given hardware vulnerability.
2. *Memory massaging*: steering targeted sensitive data towards the vulnerable physical memory locations.
3. *Exploitation*: triggering the hardware vulnerability to corrupt the intended data for exploitation.

In the remainder of this section, we detail each of these steps and outline FFS's end-to-end attack strategy.

2.1 Memory Templating

The goal of the memory templating step is to fingerprint the hardware bit-flip patterns on the running system. This is necessary, since the locations of hardware bit flips are generally unknown in advance. This is specifically true in the case of Rowhammer; every (vulnerable) DRAM module is unique in terms of physical memory offsets with bit flips. In this step, the attacker triggers the hardware-specific vulnerability to determine which physical pages, and which offsets within those pages are vulnerable to bit flips. We call the combination of a vulnerable page and the offset a *template*.

Probing for templates provides the attacker with knowledge of *usable* bit flips. Thanks to Flip Feng Shui, any template can potentially allow the attacker to exploit the hardware vulnerability over physical memory in a controlled way. The usefulness of such an exploit, however, depends on the direction of the bit flip (i.e., one-to-zero or zero-to-one), the page offset, and the contents of the target victim page. For each available template, the attacker can only craft a Flip Feng Shui primitive that corrupts the target data page with the given *flip* and *offset*. Hence, to surgically target the victim's sensitive data of interest, the attacker needs to probe for matching templates by repeatedly exploiting the hardware vulnerability over a controlled physical page (i.e., mapped in her virtual address space). To perform this step efficiently, our own dFFS implementation relies on a variant of double-sided Rowhammer [51]. Rowhammer allows an attacker to induce bit flips in vulnerable memory locations by repeatedly reading from memory pages located in adjacent rows. We discuss the low-level details

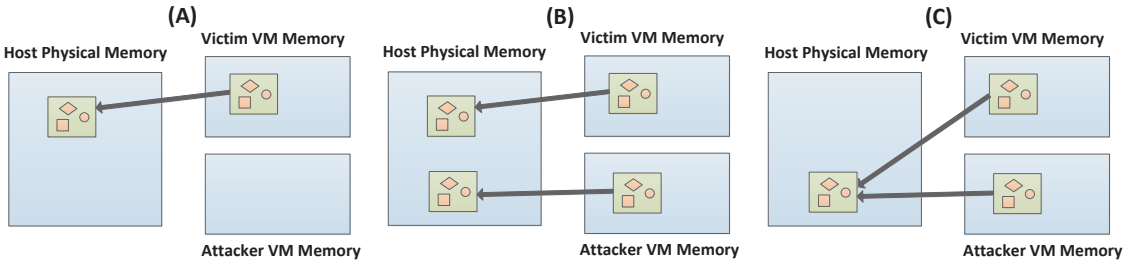

Figure 1: Memory deduplication can provide an attacker control over the layout of physical memory.

of the Rowhammer vulnerability and our implementation in Section 4.2.

2.2 Memory Massaging

To achieve bit flips over arbitrary contents of the victim's physical memory, FFS abuses modern memory management patterns and features to craft a memory massaging primitive. Memory massaging allows the attacker to map a desired victim's physical memory page into her own virtual memory address space in a controllable way.

Given a set of templates and the memory massaging primitive, an ideal version of FFS can corrupt any of the victim's memory pages at an offset determined by the selected template.

While memory massaging may be nontrivial in the general case, it is surprisingly easy to abuse widely deployed memory deduplication features to craft practical FFS attacks that corrupt any of the victim's memory pages with *known contents* (similar to our dFFS implementation). Intuitively, since memory deduplication merges system-wide physical memory pages with the same contents, an attacker able to craft the contents of any of the victim's memory pages can obtain a memory massaging primitive and map the target page into her address space.

Figure 1 shows how an attacker can control the physical memory location of a victim VM's memory page. At first, the attacker needs to predict the contents of the victim VM's page that she wants to control (Figure 1-A). Once the target page is identified, the attacker VM creates a memory page with the same contents as the victim VM's memory page and waits for the memory deduplication system to scan both pages (Figure 1-B). Once the two physical pages (i.e., the attacker's and the victim's pages) are identified, the memory deduplication system returns one of the two pages back to the system, and the other physical page is used to back both the attacker and the victim's (virtual) pages. If the attacker's page is used to back the memory of the victim page, then, in effect, the attacker controls the physical memory location of the victim page (Figure 1-C).

There are additional details necessary to craft a memory massaging primitive using a real-world implementation of memory deduplication (e.g., KSM). Section 4.1 elaborates on such details and presents our implementation of memory massaging on Linux.

2.3 Exploitation

At this stage, FFS already provides the attacker with templated bit flips over the victim's physical memory pages with known (or predictable) contents. The exploitation surface is only subject to the available templates and their ability to reach interesting locations for the attacker. As we will see, the options are abundant.

While corrupting the memory state of running software of the victim is certainly possible, we have opted for a more straightforward, yet extremely powerful exploitation strategy. We consider an attacker running in a cloud VM and seeking to corrupt interesting contents in the page cache of a co-hosted victim VM. In particular, our dFFS implementation includes two exploits that corrupt sensitive file contents in the page cache in complete absence of software vulnerabilities:

1. Flipping SSH's `authorized_keys`: assuming the RSA public keys of the individuals accessing the victim VM are known, an attacker can use dFFS to induce an exploitable flip in their public keys, making them prone to factorization and breaking the authentication system.

2. Flipping apt's `sources.list` and `trusted.gpg`: Debian/Ubuntu's `apt` package management system relies on the `sources.list` file to operate daily updates and on the `trusted.gpg` file to check the authenticity of the updates via RSA public keys. Compromising these files allows an attacker to make a victim VM download and install arbitrary attacker-generated packages.

In preliminary experiments, we also attempted to craft an exploit to bit-flip SSH's `moduli` file containing Diffie-Hellman group parameters and eavesdrop on the victim

VM's SSH traffic. The maximum group size on current distributions of OpenSSH is 1536. When we realized that an exploit targeting such 1536-bit parameters would require a nontrivial computational effort (see Appendix A for a formal analysis), we turned our attention to the two more practical and powerful exploits above.

In Section 3, we present a cryptanalysis of RSA moduli with a bit flip as a result of our attacks. In Section 4, we elaborate on the internals of the exploits, and finally, in Section 5, we evaluate their success rate and time requirements in a typical cloud setting.

3 Cryptanalysis of RSA with Bit Flips

RSA [49] is a public-key cryptosystem: the sender encrypts the message with the public key of the recipient (consisting of an exponent e and a modulus n) and the recipient decrypts the ciphertext with her private key (consisting of an exponent d and a modulus n). This way RSA can solve the key distribution problem that is inherent to symmetric encryption. RSA can also be used to digitally sign messages for data or user authentication: the signing operation is performed using the private key, while the verification operation employs the public key.

Public-key cryptography relies on the assumption that it is computationally infeasible to derive the private key from the public key. For RSA, computing the private exponent d from the public exponent e is believed to require the factorization of the modulus n. If n is the product of two large primes of approximately the same size, factorizing n is not feasible. Common sizes for n today are 1024 to 2048 bits.

In this paper we implement a fault attack on the modulus n of the victim: we corrupt a single bit of n, resulting in n'. We show that with high probability n' will be easy to factorize. We can then compute from e the corresponding value of d', the private key, that allows us to forge signatures or to decrypt. We provide a detailed analysis of the expected computational complexity of factorizing n' in the following[1].

RSA perform computations modulo n, where t is the bitlength of n ($t = 1 + \lfloor \log_2 n \rfloor$). Typical values of t lie between 512 (export control) and 8192, with 1024 and 2048 the most common values. We denote the ith bit of n with $n[i]$ ($0 \le i < t$), with the least significant bit (LSB) corresponding to $n[0]$. The unit vector is written as e_i , that is $e_i[i] = 1$ and $e_i[j] = 0$, for $j \neq i$. The operation of flipping the ith bit of n results in n', or $n' = n \oplus e_i$. Any integer can be written as the product of primes, hence $n = \prod_{j=1}^{s} p_i^{\gamma_i}$, where p_i are the prime factors of n, γ_i is the multiplicity of p_i and s is the number of distinct prime factors. W.l.o.g. we assume that $p_1 > p_2 > \cdots p_s$.

In the RSA cryptosystem, the modulus n is the product of two odd primes p_1, p_2 of approximate equal size, hence $s = 2$, and $\gamma_1 = \gamma_2 = 1$. The encryption operation is computed as $c = m^e \bmod n$, with e the public exponent, and $m, c \in [0, n-1]$) the plaintext respectively the ciphertext. The private exponent d can be computed as $d = e^{-1} \bmod \lambda(n)$, with $\lambda(n)$ the Carmichael function, given by $\mathrm{lcm}(p_1, p_2)$. The best known algorithm to recover the private key is to factorize n using the General Number Field Sieve (GNFS) (see e.g. [42]), which has complexity $O(L_n[1/3, 1.92])$, with

$$L_n[a,b] = \exp\left((b+o(1))(\ln n)^a(\ln\ln n)^{1-a}\right).$$

For a 512-bit modulus n, Adrian et al. estimate that the cost is about 1 core-year [3]. The current record is 768 bits [35], but it is clear that 1024 bits is within reach of intelligence agencies [3].

If we flip the LSB of n, we obtain $n' = n - 1$, which is even hence $n' = 2 \cdot n''$ with n'' a $t-1$-bit integer. If we flip the most significant bit of n, we obtain the odd $t-1$-bit integer n'. In all the other cases we obtain an odd t-bit integer n'. We conjecture that the integer n'' (for the LSB case) and the integers n' (for the other cases) have the same distribution of prime factors as a random odd integer. To simplify the notation, we omit in the following the LSB case, but the equations apply with n' replaced by n''.

Assume that an attacker can introduce a bit flip to change n into n' with as factorization $n = \prod_{j=1}^{s'} {p_i'}^{\tilde{\gamma}_i}$. Then $c' = m'^e \bmod n'$. The Carmichael function can be computed as

$$\lambda(n') = \mathrm{lcm}\left(\left\{ {p_i'}^{\tilde{\gamma}_i - 1} \cdot (p_i' - 1) \right\}\right).$$

If $\gcd(e, \lambda(n')) = 1$, the private exponent d' can be found as $d' = e^{-1} \bmod \lambda(n')$. For prime exponents e, the probability that $\gcd(e, \lambda(n')) > 1$ equals $1/e$. For $e = 3$, this means that 1 in 3 attacks fails, but for the widely used value $e = 2^{16} + 1$, this is not a concern. With the private exponent d' we can decrypt or sign any message. Hence the question remains how to factorize n'. As it is very likely that n' is not the product of two primes of almost equal size, we can expect that factorizing n' is much easier than factorizing n.

Our conjecture implies that with probability $2/\ln n'$, n' is prime and in that case the factorization is trivial. If n' is composite, the best approach is to find small factors (say up to 16 bits) using a greatest common divisor operation with the product of the first primes. The next step is to use Pollard's ρ algorithm (or Brent's variant) [42]: this algorithm can easily find factors up to 40...60 bits. A third step consist of Lenstra's Elliptic Curve factorization Method (ECM) [38]: ECM can quickly find factors up to 60...128 bits (the record is a factor of about

[1] A similar analysis for Diffie-Hellman group parameters with bit flips can be found in Appendix A.

270 bits[2]). Its complexity to find the smallest prime factor p'_s is equal to $O(L_{p'_s}[1/2, \sqrt{2}])$. While ECM is asymptotically less efficient than GNFS (because of the parameter 1/2 rather than 1/3), the complexity of ECM depends on the size of the smallest prime factor p'_s rather than on the size of the integer n' to factorize. Once a prime factor p'_i is found, n' is divided by it, the result is tested for primality and if the result is composite, ECM is restarted with as argument n'/p'_i.

The complexity analysis of ECM depends on the number of prime factors and the distribution of the size of the second largest prime factor p'_2: it is known that its expected valued is $0.210 \cdot t$ [36]. The Erdös–Kac theorem [22] states that the number $\omega(n')$ of distinct prime factors of n' is normally distributed with mean and variance $\ln\ln n'$: for $t = 1024$ the mean is about 6.56, with standard deviation 2.56. Hence it is unlikely that we have exactly two prime factors (probability 3.5%), and even less likely that they are of approximate equal size. The probability that n' is prime is equal to 0.28%. The expected size of the second largest prime factor p'_2 is 215 bits and the probability that it has less than 128 bits is 0.26 [36]. In this case ECM should be very efficient. For $t = 2048$, the probability that n' is prime equals 0.14%. The expected size of the second largest prime factor p'_2 is 430 bits; the probability that p'_2 has less than 228 bits is 0.22 and the probability that it has less than 128 bits is about 0.12. Similarly, for $t = 4096$, the expected size of the second largest prime factor p'_2 is 860 bits. The probability that p'_2 has less than 455 bits is 0.22.

The main conclusion is that *if n has 1024-2048 bits, we can expect to factorize n' efficiently with a probability of $12-22\%$ for an arbitrary bit flip, but larger moduli should also be feasible.* As we show in Section 5, given a few dozen templates, we can easily factorize any 1024 bit to 4096 bit modulus with one (or more) of the available templates.

4 Implementation

To implement dFFS reliably on Linux, we need to understand the internals of two kernel subsystems, kernel same-page merging [6] (KSM) and transparent huge pages [5], and the way they interact with each other. After discussing them and our implementation of the Rowhammer exploit (Sections 4.1, 4.2, and 4.3), we show how we factorized corrupted RSA moduli in Section 4.4 before summarizing our end-to-end attacks in Section 4.5.

[2] `https://en.wikipedia.org/wiki/Lenstra_elliptic_curve_factorization`

4.1 Kernel Same-page Merging

KSM, the Linux implementation of memory deduplication, uses a kernel thread that periodically scans memory to find memory pages with the same contents that are candidates for merging. It then keeps a single physical copy of a set of candidate pages, marks it read-only, and updates the page-table entries of all the other copies to point to it before releasing their physical pages to the system.

KSM keeps two red-black trees, termed "stable" and "unstable", to keep track of the merged and candidate pages. The merged pages reside in the stable tree while the candidate contents that are not yet merged are in the unstable tree. KSM keeps a list of memory areas that are registered for deduplication and goes through the pages in these areas in the order in which they were registered. For each page that it scans, it checks if the stable tree already contains a page with the same contents. If so, it updates the page-table entry for that page to have it point to the physical page in the stable tree and releases the backing physical page to the system. Otherwise, it searches the unstable tree for a match and if it finds one, promotes the page to the stable tree and updates *the page-table entry of the match to make it point to this page*. If no match is found in either one of the trees, the page is added to the unstable tree. After going through all memory areas, KSM dumps the unstable tree before starting again. Further details on the internals of KSM can be found in [6].

In the current implementation of KSM, during a merge, the physical page in either the stable tree or the unstable tree is always preferred. This means that during a merge with a page in the stable tree, the physical location of the page in the stable tree is chosen. Similarly, the physical memory of the page in the unstable tree is chosen to back both pages. KSM scans the memory of the VMs in order that they have been registered (i.e., their starting time). This means that to control the location of the target data on physical memory using the *unstable tree* the attacker VM should have been started *before* the victim VM. Hence, the longer the attacker VM waits, the larger the chance of physical memory massaging through the unstable tree.

The better physical memory massaging possibility is through the stable tree. An attacker VM can upgrade a desired physical memory location to the stable tree by creating two copies of the target data and placing one copy in the desired physical memory location and another copy in a different memory location. By ensuring that the other copy comes after the desired physical memory location in the physical address-space, KSM merges the two pages and creates a stable tree node using the desired physical memory location. At this point, any other

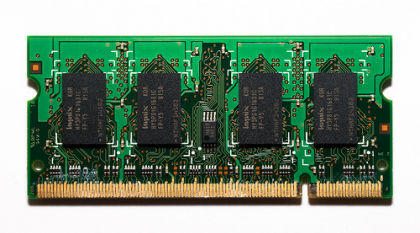

Figure 2: A SO-DIMM with its memory chips.

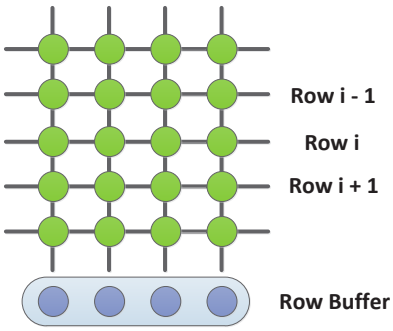

Figure 3: DRAM's internal organization.

page with the same contents will assume the same physical memory location desired by the attacker VM. For this to work, however, the attacker needs to control when the memory page with the target contents is created in the victim VM. In the case of our OpenSSH attack, for example, the attacker can control when the target page is created in the victim VM by starting an SSH connection using an invalid key with the target username.

For simplicity, the current version of dFFS implements the memory massaging using the unstable tree by assuming that the attacker VM has started first, but it is trivial to add support for memory massaging with stable tree. Using either the stable or unstable KSM trees for memory massaging, all dFFS needs to do is crafting a page with the same contents as the victim page and place it at the desired physical memory page; KSM will then perform the necessary page-table updates on dFFS's behalf! In other words, KSM inadvertently provides us with exactly the kind of memory massaging we need for successful Flip Feng Shui.

4.2 Rowhammer inside KVM

Internally, DRAM is organized in rows. Each row provides a number of physical cells that store memory bits. For example, in an x86 machine with a single DIMM, each row contains 1,048,576 cells that can store 128 kB of data. Each row is internally mapped to a number of chips on the DIMM as shown in Figure 2.

Figure 3 shows a simple organization of a DRAM chip. When the processor reads a physical memory location, the address is translated to an offset on row i of the DRAM. Depending on the offset, the DRAM selects the proper chip. The selected chip then copies the contents of its row i to the row buffer. The contents at the correct offset within the row buffer is then sent on the bus to the processor. The row buffer acts as a cache: if the selected row is already in the row buffer, there is no need to read from the row again.

Each DRAM cell is built using a transistor and a capacitor. The transistor controls whether the contents of the cell is accessible, while the capacitor can hold a charge which signifies whether the stored content is a high or low bit. Since capacitors leak charge over time, the processor sends refresh commands to DIMM rows in order to recharge their contents. On top of the refresh commands, every time a row is read by the processor, the chip also recharges its cells.

As DRAM components have become smaller, they keep a smaller charge to signify stored contents. With a smaller charge, the error margin for identifying whether the capacitor is charged (i.e., the stored value) is also smaller. Kim et al. [34] showed that the smaller error margin, in combination with unexpected charge exchange between cells of different rows, can result in the cell to "lose" its content. To trigger this DRAM reliability issue, an attacker needs fast activations of DRAM rows which causes a cell in adjacent rows to lose enough charge so that its content is cleared. Note that due to the row buffer, at least two rows need to activate one after the other in a tight loop for Rowhammer to trigger. If only one row is read from, the reads can be satisfied continually from the row buffer, without affecting the row charges in the DRAM cells.

Double-sided Rowhammer. Previous work [51] reported that if these two "aggressor" rows are selected in a way that they are one row apart (e.g., row $i-1$ and $i+1$ in Figure 3), the chances of charge interaction between these rows and the row in the middle (i.e., row i) increases, resulting in potential bit flips in that row. This variant of Rowhammer is named double-sided Rowhammer. Apart from additional speed for achieving bit flips, it provides additional reliability by isolating the location of most bit flips to a certain (victim) row.

To perform double-sided Rowhammer inside KVM, we need to know the host physical addresses inside the VM. This information is, however, not available in the guest: guest physical addresses are mapped to host virtual addresses which can be mapped to any physical page by the Linux kernel. Similar to [30], we rely on transparent huge pages [5] (THP). THP is a Linux kernel feature

that runs in the background and merges virtually contiguous normal pages (i.e., 4 kB pages) into huge pages (i.e., 2 MB pages) that rely on contiguous pieces of physical memory. THP greatly reduces the number of page-table entries in suitable processes, resulting in fewer TLB[3] entries. This improves performance for some workloads.

THP is another (weak) form of memory massaging: it transparently allows the attacker control over how the system maps guest physical memory to host physical memory. Once the VM is started and a certain amount of time has passed, THP will transform most of the VM's memory into huge pages. Our current implementation of dFFS runs entirely in the userspace of the guest and relies on the default-on THP feature of both the host and the guest. As soon as the guest boots, dFFS allocates a large buffer with (almost) the same size as the available memory in the guest. The THP in the host then converts guest physical addresses into huge pages and the THP in the guest turns the guest virtual pages backing dFFS's buffer into huge pages as well. As a result, dFFS's buffer will largely be backed by huge pages all the way down to host physical memory.

To make sure that the dFFS's buffer is backed by huge pages, we request the guest kernel to align the buffer at a 2 MB boundary. This ensures that if the buffer is backed by huge pages, it starts with one: on the `x86_64` architecture, the virtual and physical huge pages share the lowest 20 bits, which are zero. The same applies when transitioning from the guest physical addresses to host physical addresses. With this knowledge, dFFS can assume that the start of the allocated buffer is the start of a memory row, and since multiple rows fit into a huge page, it can successively perform double-sided Rowhammer on these rows. To speed up our search for bit flips during double-sided Rowhammer on each two rows, we rely on the row-conflict side channel for picking the hammering addresses within each row [44]. We further employed multiple threads to amplify the Rowhammer effect.

While THP provides us with the ability to efficiently and reliably induce Rowhammer bit flips, it has unexpected interactions with KSM that we will explore in the next section.

4.3 Memory Massaging with KSM

In Section 2.2, we discussed the operational semantics of KSM. Here we detail some of its implementation features that are important for dFFS.

[3]TLB or translation lookaside buffer is a general term for processor caches for page-table entries to speed up the virtual to physical memory translation

4.3.1 Interaction with THP

As we discussed earlier, KSM deduplicates memory pages with the same contents as soon as it finds them. KSM currently does not support deduplication of huge pages, but what happens when KSM finds matching contents within huge pages?

A careful study of the KSM shows that KSM always prefers reducing memory footprint over reducing TLB entries; that is, KSM breaks down huge pages into smaller pages if there is a small page inside with similar contents to another page.

This specific feature is important for an efficient and reliable implementation of dFFS, but has to be treated with care. More specifically, we can use huge pages as we discussed in the previous section for efficient and reliable double-sided Rowhammer, while retaining control over which victim page we should map in the middle of our target (vulnerable) huge page.

KSM, however, can have undesired interactions with THP from dFFS's point of view. If KSM finds pages in the attacker VM's memory that have matching contents, it merges them with each other or with a page from a previously started VM. In these situations, KSM breaks THP by releasing one of its smaller pages to the system. To avoid this, dFFS uses a technique to avoid KSM during its templating phase. KSM takes a few tens of seconds to mark the pages of dFFS's VM as candidates for deduplication. This gives dFFS enough time to allocate a large buffer with the same size as VM's available memory (as mentioned earlier) and write unique integers at a pre-determined location within each (small) page of this buffer as soon as its VM boots. The entropy present within dFFS's pages then prohibits KSM to merge these pages which in turn avoids breaking THP.

4.3.2 On dFFS Chaining

Initially, we planned on chaining memory massaging primitive and FFS to induce an arbitrary number of bit flips at many desired locations of the victim's memory page. After using the first template for the first bit flip, the attacker can write to the merged memory page to trigger a copy-on-write event that ultimately unmerges the two pages (i.e., the attacker page from the victim page). At this stage, the attacker can use dFFS again with a new template to induce another bit flip.

However, the implementation of KSM does not allow this to happen. During the copy-on-write event, the victim's page remains in the stable tree, even if it is the only remaining page. This means that subsequent attempts for memory massaging results in the victim page to control the location of physical memory, disabling the attacker's ability for chaining FFS attacks.

Even so, based on our single bit flip cryptanalysis on public keys and our evaluation in Section 5, chaining is not necessary for performing successful end-to-end attacks with dFFS.

4.4 Attacking Weakened RSA

For the two attacks in this paper, we generate RSA private keys, i.e., the private exponents d' corresponding to corrupted moduli n' (as described in Section 3). We use d' to compromise two applications: OpenSSH and GPG.

Although the specifics of the applications are very different, the pattern to demonstrate each attack is the same and as follows:

1. Obtain the file containing the RSA public key (n,e). This is application-specific, but due to the nature of public key cryptosystems, generally unprotected. We call this the input file.

2. Using the memory templating step of Section 2.1 we obtain a list of templates that we are able to flip within a physical page. We flip bits according to the target templates to obtain corrupted keys. For every single bitflip, we save a new file. We call these files the corrupted files. According to the templating step, dFFS has the ability to create any of these corrupted files in the victim by flipping a bit in the page cache.

3. One by one, we now read the (corrupted) public keys for each corrupted file. If the corrupted file is parsed correctly *and* the public key has a changed modulus $n' \neq n$ and the same e, this n' is a candidate for factorization.

4. We start factorizations of all n' candidates found in the previous step. As we described in Section 3, the best known algorithm for our scenario is ECM that finds increasingly large factor in an iterative fashion. We use the Sage [19] implementation of ECM for factorizing n'. We invoke an instance of ECM per available core for each corrupted key with a 1 hour timeout (all available implementations of ECM run with a single thread).

5. For all successful factorizations, we compute the private exponent d' corresponding to (n',e) and generate the corresponding private key to the corrupted public key. How to compute d' based on the factorization of n' is described in Section 3. We can then use the private key with the unmodified application. This step is application-specific and we will discuss it for our case studies shortly.

We now describe our end-to-end attacks that put all the pieces of dFFS together using two target applications: OpenSSH and GPG.

4.5 End-to-end Attacks

Attacker model. The attacker owns a VM co-hosted with a victim VM on a host with DIMMs susceptible to Rowhammer. We further assume that memory deduplication is turned on—as is common practice in public cloud settings [8]. The attacker has the ability to use the memory deduplication side-channel to fingerprint low-entropy information, such as the IP address of the victim VM, OS/library versions, and the usernames on the system (e.g., through `/etc/passwd` file in the page cache) as shown by previous work [32, 43, 56]. The attacker's goal is to compromise the victim VM without relying on any software vulnerability. We now describe how this model applies with dFFS in two important and widely popular applications.

4.5.1 OpenSSH

One of the most commonly used authentication mechanisms allowed by the OpenSSH daemon is an RSA public key. By adding a user's RSA public key to the SSH `authorized_keys` file, the corresponding private key will allow login of that user without any other authentication (such as a password) in a default setting. The public key by default includes a 2048 bit modulus n. The complete key is a 372-byte long base64 encoding of (n,e).

The attacker can initiate an SSH connection to the victim with a correct victim username and an arbitrary private key. This interaction forces OpenSSH to read the `authorized_keys` file, resulting in this file's contents getting copied into the page cache at the right time as we discussed in Section 4.1. Public key cryptosystems by definition do not require public keys to be secret, therefore we assume an attacker can obtain a victim public key. For instance, GitHub makes the users' submitted SSH public keys publicly available [27].

With the victim's public key known and in the page cache, we can initiate dFFS for inducing a bit flip. We cannot flip just any bit in the memory page caching the `authorized_keys`; some templates *will* break the base64 encoding, resulting in a corrupted file that OpenSSH does not recognize. Some flips, however, decode to a valid (n',e) key that we can factorize. We report in Section 5 how many templates are available on average for a target public key.

Next, we use a script with the PyCrypto RSA cryptographic library [39] to operate on the corrupted public keys. This library is able to read and parse OpenSSH public key files, and extract the RSA parameters (n,e).

It can also generate RSA keys with specific parameters and export them as OpenSSH public (n', e) and private (n', d') keys again. All the attacker needs to do is factorize n' as we discussed in Section 4.4.

Once we know the factors of n', we generate the private key (n', d') that can be used to login to the victim VM using an unmodified OpenSSH client.

4.5.2 GPG

The GNU Privacy Guard, or GPG, is a sophisticated implementation of, among others, the RSA cryptosystem. It has many applications in security, one of which is the verification of software distributions by verifying signatures using trusted public keys. This is the larger application we intend to subvert with this attack.

Specifically, we target the `apt` package distribution system employed by Debian and Ubuntu distribution for software installation and updates. `apt` verifies package signatures after download using GPG and trusted public keys stored in `trusted.gpg`. It fetches the package index from sources in `sources.list`.

Our attack first steers the victim to our malicious repository. The attacker can use dFFS to achieve this goal by inducing a bit flip in the `sources.list` file that is present in the page cache after an update. `sources.list` holds the URL of the repositories that are used for package installation and update. By using a correct template, the attacker can flip a bit that results in a URL that she controls. Now, the victim will seek the package index and packages at an attacker-controlled repository.

Next, we use our exploit to target the GPG trusted keys database. As this file is part of the software distribution, the stock contents of this file is well-known and we assume this file is unchanged or we can guess the new changes. (Only the pages containing the keys we depend on need be either unchanged or guessed.) This file resides in the page cache every time the system tries to update as a result of a daily cron job, so in this attack, no direct interaction with the victim is necessary for bringing the file in the page cache. Our implicit assumption is that this file remains in the page cache for the next update iteration.

Similar to OpenSSH, we apply bit flip mutations in locations where we can induce bit flips according to the memory templating step. As a result, we obtain the corrupted versions of this file, and each time check whether GPG will still accept this file as a valid keyring and that one of the RSA key moduli has changed as a result of our bit flip. Extracting the key data is done with the GPG `--list-keys --with-key-data` options.

For every bitflip location corresponding to a corrupted modulus that we can factorize, we pick one of these mutations and generate the corresponding (n', d') RSA private key, again using PyCrypto. We export this private key using PyCrypto as PEM formatted key and use `pem2openpgp` [26] to convert this PEM private key to the GPG format. Here we specify the usage flags to include signing and the same generation timestamp as the original public key. We can then import this private key for use for signing using an unmodified GPG.

It is important that the Key ID in the private keyring match with the Key ID in the `trusted.gpg` file. This Key ID is not static but is based on a hash computed from the public key data, a key generation timestamp, and several other fields. In order for the Key ID in the private keyring to match with the Key ID in the public keyring, these fields have to be identical and so the setting of the creation timestamp is significant.

One significant remark about the Key ID changing (as a result of a bit flip) is that this caused the self-signature on the public keyring to be ignored by GPG! The signature contains the original Key ID, but it is now attached to a key with a different ID due to the public key mutation. As a result, *GPG ignores the attached signature as an integrity check of the bit-flipped public key and the self-signing mechanism fails to catch our bit flip*. The only side-effect is harmless to our attack – GPG reports that the trusted key is not signed. `apt` ignores this without even showing a warning. After factorizing the corrupted public key modulus, we successfully verified that the corresponding private key can generate a signature that verifies against the bit-flipped public key stored in the original `trusted.gpg`.

We can now sign our malicious package with the new private key and the victim will download and install the new package without a warning.

5 Evaluation

We evaluated dFFS to answer the following three key questions:

- What is the success probability of the dFFS attack?
- How long does the dFFS attack take?
- How much computation power is necessary for a successful dFFS attack?

We used the following methodology for our evaluation. We first used a Rowhammer testbed to measure how many templates are available in a given segment of memory and how long it takes us to find a certain template. We then executed the end-to-end attacks discussed in Section 4.5 and report on their success rate and their start-to-finish execution time. We then performed an analytical large-scale study of the factorization time, success probability, and computation requirements of 200

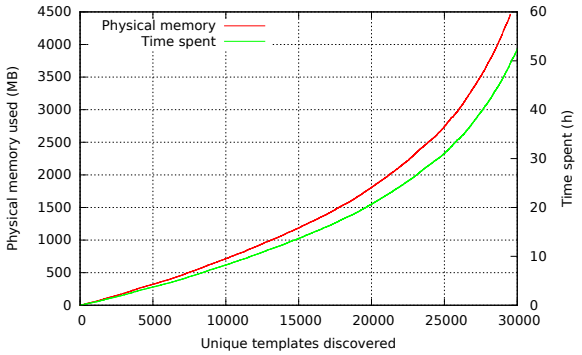

Figure 4: Required time and memory for templating.

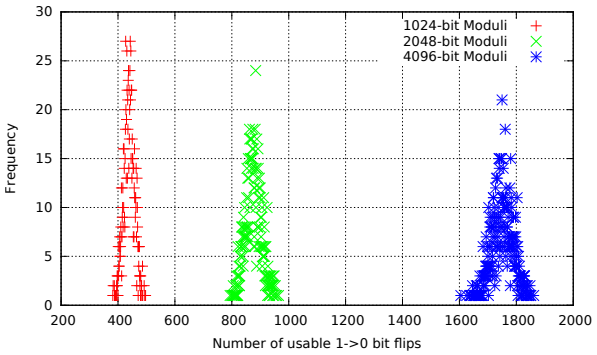

Figure 5: Number of usable 1-to-0 bit flips usable in the SSH `authorized_keys` file for various modulus sizes.

RSA public keys for each of the 1024, 2048 and 4096-bit moduli with 50 bit flips at random locations (i.e., 30,000 bit flipped public keys in total).

We used the following hardware for our Rowhammer testbed and for the cluster that we used to conduct our factorization study:

Rowhammer testbed. Intel Haswell i7-4790 4-core processor on an Asus H97-Pro mainboard with 8 GB of DDR3 memory.

Factorization cluster. Up to 60 nodes, each with two Intel Xeon E5-2630 8-core processors with 64 GB of memory.

5.1 dFFS on the Rowhammer Testbed

Memory templating. Our current implementation of Rowhammer takes an average of 10.58 seconds to complete double-sided Rowhammer for each target row. Figure 4 shows the amount of time and physical memory that is necessary for discovering a certain number of templates. Note that, in our testbed, we could discover templates for almost any bit offset (i.e., 29,524 out of 32,768 possible templates). Later, we will show that we only need a very small fraction of these templates to successfully exploit our two target programs.

Memory massaging. dFFS needs to wait for a certain amount of time for KSM to merge memory pages. KSM scans a certain number of pages in each waking period. On the default version of Ubuntu, for example, KSM scans 100 pages every 20 milliseconds (i.e., 20 MB). Recent work [12] shows that it is possible to easily detect when a deduplication pass happens, hence dFFS needs to wait at most the sum of memory allocated to each co-hosted VM. For example, in our experiments with one attacker VM and one victim VM each with 2 GB of memory, KSM takes at most around 200 seconds for a complete pass.

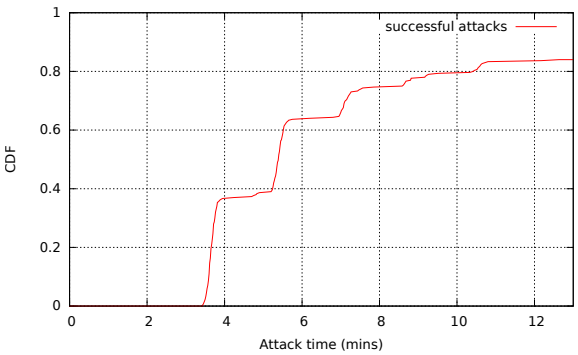

Figure 6: CDF of successful automatic SSH attacks.

5.2 The SSH Public Key Attack

Figure 5 shows the number of possible templates to perform the dFFS attack on the SSH `authorized_keys` file with a single randomly selected RSA public key, for 1024, 2048 and 4096-bit public keys. For this experiment, we assumed 1-to-0 bit flips since they are more common in our testbed. For DRAM chips that are susceptible to frequent 0-to-1 bit flips, these numbers double. For our experiment we focused on 2048-bit public keys as they are the default length as generated by the `ssh-keygen` command.

To demonstrate the working end-to-end attack, measure its reliability, and measure the elapsed time distribution, we automatically performed the SSH attack 300 times from an attacker VM on a victim VM, creating the keys and VM's from scratch each time. Figure 6 shows the CDF of successful attacks with respect to the time they took. In 29 cases (9.6%), the Rowhammer operation did not change the modulus at all (the attacker needs

Table 1: Examples of domains that are one bit flip away from ubuntu.com that we purchased.

ubuftu.com	ubunt5.com	ubunte.com
ubunuu.com	ubunvu.com	ubunpu.com
ubun4u.com	ubuntw.com	ubuntt.com

to retry). In 19 cases (6.3%), the Rowhammer operation changed the modulus other than planned. The remaining 252 (84.1%) were successful the first time. All the attacks finished within 12.6 minutes with a median of 5.3 minutes.

5.3 The Ubuntu/Debian Update Attack

We tried factorizing the two bit-flipped 4096 bit **Ubuntu Archive Automatic Signing** RSA keys found in the `trusted.gpg` file. Out of the 8,192 trials (we tried both 1-to-0 and 0-to-1 flips), we could factorize 344 templates. We also need to find a bit flip in the URL of the Ubuntu or Debian update servers (depending on the target VM's distribution) in the page cache entry for apt's `sources.list` file. For `ubuntu.com`, 29 templates result in a valid domain name, and for `debian.org`, 26 templates result in a valid domain name. Table 5.2 shows examples of domains that are one bit flip away from ubuntu.com.

Performing the update attack on our Rowhammer testbed, we could trigger a bit flip in the page cache entry of sources `sources.list` in 212 seconds, converting `ubuntu.com` to `ubunvu.com`, a domain which we control. Further, we could trigger a bit flip in the page cache entry of `trusted.gpg` that changed one of the RSA public keys to one that we had pre-computed a factorization in 352 seconds. At this point, we manually sign the malicious package with our GPG private key that corresponds to the mutated public key. When the victim then updates the package database and upgrades the packages, the malicious package is downloaded and installed without warning. Since the current version of dFFS runs these steps sequentially, the entire end-to-end attack took 566 seconds. We have prepared a video of this attack which is available at: `https://vusec.net/projects/flip-feng-shui`

Growingly concerned about the impact of such practical attacks, we conservatively registered all the possible domains from our Ubuntu/Debian list.

5.4 RSA Modulus Factorization

Figure 7 shows the average probability of successful factorizations based on the amount of available compute

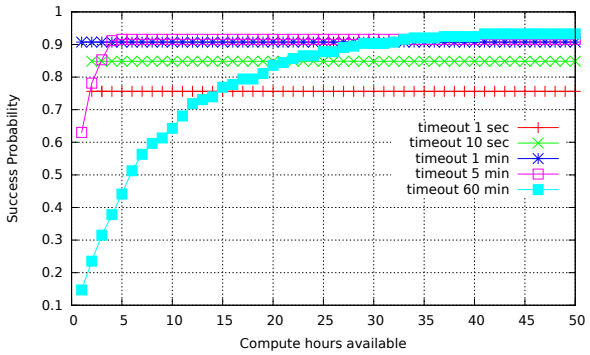

Figure 7: Compute power and factorization timeout tradeoff for 2048-bit RSA keys.

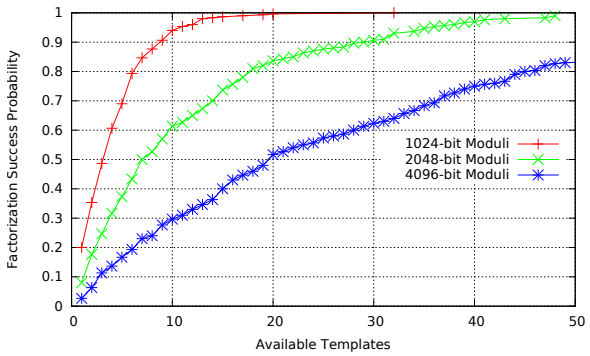

Figure 8: CDF of success rate with increasing templates.

hours. We generated this graph using 200 randomly generated 2048-bit RSA keys, each with a bit flip in 50 distinct trials (i.e., 10,000 keys, each with a bit flip). For this experiment, we relied on the ECM factorization tool, discussed in Section 3, and varied its user-controlled timeout parameter between one second and one hour. For example, with a timeout of one second for a key with a bit flip, we either timeout or the factorization succeeds immediately. In both cases, we move on to the next trial of the same key with a different bit flip.

This graph shows that, with 50 bit flips, the average factorization success probability is between 0.76 for a timeout of one second and 0.93 for a timeout of one hour. Note that, for example, with a timeout of one second, we can try 50 templates in less than 50 seconds, while achieving a successful factorization in as many as 76% of the public keys. A timeout of one minute provides a reasonable tradeoff and can achieve a success rate of 91% for 2048-bit RSA keys.

Figure 8 shows the cumulative success probability of factorization as more templates become available for 1024-bit, 2048-bit and 4096-bit keys. For 4096-bit keys, we need around 50 templates to be able to factorize a key with high probability (0.85) with a 1-hour timeout.

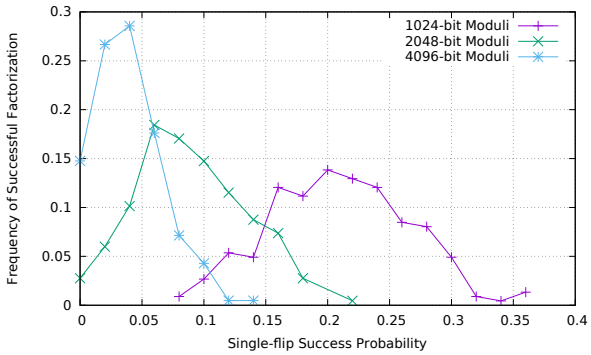

Figure 9: Probability mass function of successful factorizations with one flip.

With bit-flipped 2048-bit RSA public keys, with only 48 templates, we achieved a success probability of 0.99 with a 1 hour timeout. This proves that for 2048-bit keys (`ssh-keygen`'s default), *only a very small fraction of the templates from our testbed is necessary for a successful factorization.* For 1024-bit keys, we found a successful factorization for all keys after just 32 templates.

Some DRAM modules may only have a small number of bit flips [34], so an interesting question is: what is the chance of achieving a factorization using only a single template? Figure 9 answers this question for 1024-bit, 2048-bit and 4096-bit moduli separately. To interpret the figure, fix a point on the horizontal axis: this is the probability of a successful factorization using a single bit flip within 1 hour. Now read the corresponding value on the vertical axis, which shows the probability that a public key follows this success rate. For example, on average, 15% of 2048-bit RSA public keys can be factored using only a single bit flip with probability 0.1. As is expected, the probability to factor 4096-bit keys with the same 1-hour timeout is lower, and for 1024-bit keys higher. The fact that the distributions are centered around roughly 0.22, 0.11, and 0.055 are consistent with our analytical results in 3, which predict the factorization cost is linear in the bitlength of the modulus.

6 Mitigations

Mitigating Flip Feng Shui is not straightforward as hardware reliability bugs become prevalent. While there is obviously need for new testing methods and certification on the hardware manufacturer's side [4], software needs to adapt to fit Flip Feng Shui in its threat model. In this section, we first discuss concrete mitigations against dFFS before suggesting how to improve software to counter FFS attacks.

Table 2: Memory savings with different dedup strategies.

Strategy	Required memory	Savings
No dedup	506 GB	0%
Zero-page dedup	271 GB	46%
Full dedup	108 GB	79%

6.1 Defending against dFFS

We discuss both hardware and software solutions for defending against dFFS.

6.1.1 In Hardware

We recommend DRAM consumers perform extensive Rowhammer testing [2] to identify vulnerable DRAM modules. These DRAM modules should be replaced, but if this is not possible, reducing DRAM refresh intervals (e.g., by half) may be sufficient to protect against Rowhammer [51]. However, this also reduces DRAM performance and consumes additional power.

Another option is to rely on memory with error-correcting codes (ECC) to protect against single bit flips. Unfortunately, we have observed that Rowhammer can occasionally induce multiple flips in a single 64-bit word confirming the findings of the original Rowhammer paper [34]. These multi-flips can cause corruption even in presence of ECC. More expensive multi-ECC DIMMs can protect against multiple bit flips, but it is still unclear whether they can completely mitigate Rowhammer.

A more promising technology is *directed row refresh*, which is implemented in low-power DDR4 [7] (LPDDR4) and some DDR4 implementations. LPDDR4 counts the number of activations of each row and, when this number grows beyond a particular threshold, it refreshes the adjunct rows, preventing cell charges from falling below the error margin. Newer Intel processors support a similar feature for DDR3, but require compliant DIMMs. While these fixes mitigate Rowhammer, replacing most of current DDR3 deployments with LPDDR4 or secure DDR4 DIMMs (some DDR4 DIMMs are reported to be vulnerable to Rowhammer [1]), is not economically feasible as it requires compatible mainboards and processors. As a result, a software solution is necessary for mitigating Rowhammer in current deployments.

6.1.2 In Software

The most obvious mitigation against dFFS is disabling memory deduplication. In fact, this is what we recommend in security-sensitive environments. Disabling memory deduplication completely, however, wastes a

substantial amount of physical memory that can be saved otherwise [6, 46, 54].

Previous work [12] showed that deduplicating zero pages alone can retain between 84% and 93% of the benefits of full deduplication in a browser setting. Limiting deduplication to zero pages and isolating their Rowhammer-prone surrounding rows was our first mitigation attempt. To understand whether zero-page deduplication retains sufficient memory saving benefits in a cloud setting, we performed a large-scale memory deduplication study using 1,011 memory snapshots of *different* VMs from community VM images of Windows Azure [48]. Table 2 presents our results. Unfortunately, zero-page deduplication only saves 46% of the potential 79%. This suggests that deduplicating zero pages alone is insufficient to eradicate the wasteful redundancy in current cloud deployments. Hence, we need a better strategy that can retain the benefits of full memory deduplication without resulting in a memory massaging primitive for the attackers.

A strawman design A possible solution is to rely on a deduplication design that, for every merge operation, randomly allocates a new physical page to back the existing duplicate pages. When merge operations with existing shared pages occur, such design would need to randomly select a new physical page and update all the page-table mappings for all the sharing parties.

This strawman design eliminates the memory massaging primitive that is necessary for dFFS under normal circumstances. However, this may be insufficient if an attacker can find different primitives to control the physical memory layout. For example, the attacker's VM can corner the kernel's page allocator into allocating pages with predictable patterns if it can force the host kernel into an out-of-memory (OOM) situation. This is not difficult if the host relies on over-committed memory to pack more VMs than available RAM, a practice which is common in cloud settings and naturally enabled by memory deduplication. For example, the attacker can trigger a massive number of unmerge operations and cause the host kernel to approach an OOM situation. At this point, the attacker can release vulnerable memory pages to the allocator, craft a page with the same contents as the victim page, and wait for a merge operation. Due to the near-OOM situation, the merge operation happens almost instantly, forcing the host kernel to predictably pick one of the previously released vulnerable memory pages (i.e., templates) to back the existing duplicate pages (the crafted page and the victim page). At this stage, the attacker has again, in effect, a memory massaging primitive.

A better design To improve on the strawman design, the host needs to ensure enough memory is available not to get cornered into predictable physical memory reuse patterns. Given a desired level of entropy h, and the number of merged pages m_i for for the ith VM, the host needs to ensure $A = 2^h + Max(m_i)$ memory pages are available or can easily become available (e.g., page cache) to the kernel's page allocator at all times. With an adequate choice of h, it may become difficult for an attacker to control the behavior of the memory deduplication system. We have left the study of the right parameters for h and the projected A for real systems to future work. We also note that balancing entropy, memory, and performance when supporting a truly random and deduplication-enabled physical memory allocator is challenging, and a promising direction for future work.

6.2 Mitigating FFS at the Software Layer

The attacks presented in this paper provide worrisome evidence that even the most security-sensitive software packages used in production account for no attacker-controlled bit flips as part of their threat model. While there is certainly room for further research in this direction, based on our experience, we formulate a number of suggestions to improve current practices:

- Security-sensitive information needs to be checked for integrity in software right before use to ensure the window of corruption is small. In all the cases we analyzed, such integrity checks would be placed on a slow path with essentially no application performance impact.

  Certificate chain formats such as X.509 are automatically integrity checked as certificates are always signed [17]. This is a significant side benefit of a certification chain with self-signatures.

- The file system, due to the presence of the page cache, should not be trusted. Sensitive information on stable storage should include integrity or authenticity information (i.e., a security signature) for verification purposes. In fact, this simple defense mechanism would stop the two dFFS attacks that we presented in this paper.

- Low-level operating system optimizations should be included with extra care. Much recent work [11, 12, 29, 40, 58] shows that benign kernel optimizations such as transparent huge pages, vsyscall pages, and memory deduplication can become dangerous tools in the hands of a capable attacker. In the case of FFS, any feature that allows an untrusted entity to control the layout or reuse of data in physical memory may provide an attacker with a memory massaging primitive to mount our attacks.

7 Related Work

We categorize related work into three distinct groups discussed below.

7.1 Rowhammer Exploitation

Pioneering work on the Rowhammer bug already warned about its potential security implications [34]. One year later, Seaborn published the first two concrete Rowhammer exploits, in the form of escaping the Google Native Client (NaCl) sandbox and escalating local privileges on Linux [51]. Interestingly, Seaborn's privilege escalation exploit relies on a weak form of memory massaging by probabilistically forcing a OOMing kernel to reuse physical pages released from user space. dFFS, in contrast, relies on a deterministic memory massaging primitive to map pages from co-hosted VMs and mount fully reliable attacks. In addition, while mapping pages from kernel space for local privilege escalation is possible, dFFS enables a much broader range of attacks over nearly arbitrary physical memory.

Furthermore, Seaborn's exploits relied on Intel x86's `CLFLUSH` instruction to evict a cache line from the CPU caches in order to read directly from DRAM. For mitigation purposes, `CLFLUSH` was disabled in NaCl and the same solution was suggested for native CPUs via a microcode update. In response to the local privilege exploit, Linux disabled unprivileged access to virtual-to-physical memory mapping information (i.e., `/proc/self/pagemap`) used in the exploit to perform double-sided Rowhammer. Gruss et al. [30], however, showed that it is possible to perform double-sided Rowhammer from the browser, without `CLFLUSH`, and without pagemap, using cache eviction sets and transparent huge pages (THP). dFFS relies on nested THP (both in the host and in the guest) for reliable double-sided Rowhammer. In our previous work [12], we took the next step and implemented the first reliable Rowhammer exploit in the Microsoft Edge browser. Our exploit induces a bit flip in the control structure of a JavaScript object for pivoting to an attacker-controlled counterfeit object. The counterfeit object provides the attackers with arbitrary memory read and write primitives inside the browser.

All the attacks mentioned above rely on one key assumption: the attacker already *owns* the physical memory of the victim to make Rowhammer exploitation possible. In this paper, we demonstrated that, by abusing modern memory management features, it is possible to completely lift this assumption with alarming consequences. Using FFS, an attacker can seize control of nearly arbitrary physical memory in the software stack, for example compromising co-hosted VMs in complete absence of software vulnerabilities.

7.2 Memory Massaging

Sotirov [55] demonstrates the power of controlling virtual memory allocations in JavaScript, bypassing many protections against memory errors with a technique called Heap Feng Shui. Mandt [41] demonstrates that it is possible to control reuse patterns in the Windows 7 kernel heap allocator in order to bypass the default memory protections against heap-based attacks in the kernel. Inspired by these techniques, our Flip Feng Shui demonstrates that an attacker abusing benign and widespread memory management mechanisms allows a single bit flip to become a surprisingly dangerous attack primitive over physical memory.

Memory spraying techniques [25, 33, 47, 50] allocate a large number of objects in order to make the layout of memory predictable for exploitation purposes, similar, in spirit, to FFS. Govindavajhala and Appel [28] sprayed the entire memory of a machine with specially-crafted Java objects and showed that 70% of the bit flips caused by rare events cosmic rays and such will allow them to escape the Java sandbox. This attack is by its nature probabilistic and, unlike FFS, does not allow for fully controllable exploitation.

Memory deduplication side channels have been previously abused to craft increasingly sophisticated information disclosure attacks [8, 12, 29, 32, 43, 56]. In this paper, we demonstrate that memory deduplication has even stronger security implications than previously shown. FFS can abuse memory deduplication to perform attacker-controlled page-table updates and craft a memory massaging primitive for reliable hardware bit flip exploitation.

7.3 Breaking Weakened Cryptosystems

Fault attacks have been introduced in cryptography by Boneh et al. [9]; their attack was highly effective against implementations of RSA that use the Chinese Remainder Theorem. Since then, many variants of fault attacks against cryptographic implementations have been described as well as countermeasures against these attacks. Seifert was the first to consider attacks in which faults were introduced in the RSA modulus [52]; his goal was limited to forging signatures. Brier et al. [14] have extended his work to sophisticated methods to recover the private key; they consider a setting of uncontrollable faults and require many hundreds to even tens of thousands of faults. In our attack setting, the attacker can choose the location and observe the modulus, which reduces the overhead substantially.

In the case of Diffie-Hellman, the risk of using it with moduli that are not strong primes or hard-to-factor integers was well understood and debated extensively dur-

ing the RSA versus DSA controversy in the early 1990s (e.g., in a panel at Eurocrypt'92 [18]). Van Oorschot and Wiener showed how a group order with small factors can interact badly with the use of small Diffie-Hellman exponents [57]. In 2015, the Logjam attack [3] raised new interest in the potential weaknesses of Diffie-Hellman parameters.

In this paper, we performed a formal cryptanalysis of RSA public keys in the presence of bit flips. Our evaluation of dFFS with bit-flipped default 2048-bit RSA public keys confirmed our theoretical results. dFFS can induce bit flips in RSA public keys and factorize 99% of the resulting 2048-bit keys given enough Rowhammer-induced bit flips. We further showed that we could factor 4.2% of the two 4096 bit **Ubuntu Archive Automatic Signing Keys** with a bit flip. This allowed us to generate enough templates to successfully trick a victim VM into installing our packages. For completeness, we also included a formal cryptanalysis of Diffie-Hellman exponents in the presence of bit flips in Appendix A.

8 Conclusions

Hardware bit flips are commonly perceived as a vehicle of production software failures with limited exploitation power in practice. In this paper, we challenged common belief and demonstrated that an attacker armed with Flip Feng Shui (FFS) primitives can mount devastatingly powerful end-to-end attacks even in complete absence of software vulnerabilities. Our FFS implementation (dFFS) combines hardware bit flips with novel memory templating and massaging primitives, allowing an attacker to controllably seize control of arbitrary physical memory with very few practical constraints.

We used dFFS to mount practical attacks against widely used cryptographic systems in production clouds. Our attacks allow an attacker to completely compromise co-hosted cloud VMs with relatively little effort. Even more worryingly, we believe Flip Feng Shui can be used in several more forms and applications pervasively in the software stack, urging the systems security community to devote immediate attention to this emerging threat.

Disclosure

We have cooperated with the National Cyber Security Centre in the Netherlands to coordinate disclosure of the vulnerabilities to the relevant parties.

Acknowledgements

We would like to thank our anonymous reviewers for their valuable feedback. This work was supported by Netherlands Organisation for Scientific Research through the NWO 639.023.309 VICI "Dowsing" project, Research Council KU Leuven under project C16/15/058, the FWO grant G.0130.13N, and by the European Commission through projects H2020 ICT-32-2014 "SHARCS" under Grant Agreement No. 644571 and H2020 ICT-2014-645622 "PQCRYPTO".

References

[1] DDR4 Rowhammer mitigation. `http://www.passmark.com/forum/showthread.php?5301-Rowhammer-mitigation&p=19553`. Accessed on 17.2.2016.

[2] Troubleshooting Memory Errors – MemTest86. `http://www.memtest86.com/troubleshooting.htm`. Accessed on 17.2.2016.

[3] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella Béguelin, and Paul Zimmermann. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. CCS'15, 2015.

[4] Barbara P. Aichinger. DDR Compliance Testing - Its time has come! In *JEDEC's Server Memory Forum*, 2014.

[5] Andrea Arcangeli. Transparent hugepage support. *KVM Forum*, 2010.

[6] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. OLS'09, 2009.

[7] JEDEC Solid State Technology Association. Low Power Double Data 4 (LPDDR4). JESD209-4A, Nov 2015.

[8] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. CAIN: Silently Breaking ASLR in the Cloud. WOOT'15, 2015.

[9] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of eliminating errors in cryptographic computations. *J. Cryptology*, 14(2), 2001.

[10] Shekhar Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6), 2005.

[11] Erik Bosman and Herbert Bos. Framing signals—a return to portable shellcode. SP'14.

[12] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. SP'16, 2016.

[13] Cyril Bouvier, Pierrick Gaudry, Laurent Imbert, Hamza Jeljeli, and Emmanuel Thomé. Discrete logarithms in $GF(p)$ – 180 digits. `https://listserv.nodak.edu/cgi-bin/wa.exe?A2=ind1406&L=NMBRTHRY&F=&S=&P=3161`. June 2014.

[14] Eric Brier, Benoît Chevallier-Mames, Mathieu Ciet, and Christophe Clavier. Why one should also secure RSA public key elements. CHES'06, 2006.

[15] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow Bending: On the Effectiveness of Control-flow Integrity. SEC'15, 2015.

[16] Cristian Constantinescu. Trends and Challenges in VLSI Circuit Reliability. *IEEE Micro*, 23(4), 2003.

[17] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. RFC 5280 - Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Technical report, May 2008.

[18] Yvo Desmedt, Peter Landrock, Arjen K. Lenstra, Kevin S. McCurley, Andrew M. Odlyzko, Rainer A. Rueppel, and Miles E. Smid. The Eurocrypt '92 Controversial Issue: Trapdoor Primes and Moduli (Panel). Eurocrypt'92, 1992.

[19] The Sage Developers. Sage Mathematics Software (Version). `http://www.sagemath.org`. Accessed on 17.2.2016.

[20] Karl Dickman. On the frequency of numbers containing prime factors of a certain relative magnitude. *Arkiv forr Matematik, Astronomi och Fysik*, 1930.

[21] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6), 1976.

[22] Paul Erdös and Mark Kac. The Gaussian Law of Errors in the Theory of Additive Number Theoretic Functions. *American Journal of Mathematics*, 62(1), 1940.

[23] Chris Evans. The poisoned NUL byte, 2014 edition). `http://googleprojectzero.blogspot.nl/2014/08/the-poisoned-nul-byte-2014-edition.html`. Accessed on 17.2.2016.

[24] Justin N. Ferguson. Understanding the heap by breaking it. In *Black Hat USA*, 2007.

[25] Francesco Gadaleta, Yves Younan, and Wouter Joosen. ESSoS'10, 2010.

[26] Daniel Kahn Gillmor. pem2openpgp - translate PEM-encoded RSA keys to OpenPGP certificates. Accessed on 17.2.2016.

[27] GitHub Developer – Public Keys. `https://developer.github.com/v3/users/keys/`. Accessed on 17.2.2016.

[28] Sudhakar Govindavajhala and Andrew W. Appel. Using Memory Errors to Attack a Virtual Machine. SP '03, 2003.

[29] Daniel Gruss, David Bidner, and Stefan Mangard. Practical Memory Deduplication Attacks in Sandboxed Javascript. ESORICS'15. 2015.

[30] Daniel Gruss, Clementine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. DIMVA'16, 2016.

[31] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Side Channels in Cloud Services: Deduplication in Cloud Storage. *IEEE Security and Privacy Magazine, special issue of Cloud Security*, 8, 2010.

[32] Gorka Irazoqui, Mehmet Sinan IncI, Thomas Eisenbarth, and Berk Sunar. Know Thy Neighbor: Crypto Library Detection in Cloud. PETS'15, 2015.

[33] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. Ret2Dir: Rethinking Kernel Isolation. SEC'14, 2014.

[34] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. ISCA'14, 2014.

[35] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman

J. J. te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA modulus. CRYPTO'10, 2010.

[36] Donald E. Knuth and Luis Trabb-Pardo. Analysis of a Simple Factorization Algorithm. *Theoretical Computer Science*, 3(3), 1976.

[37] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. HAFT: Hardware-assisted Fault Tolerance. EuroSys'16, 2016.

[38] Hendrik W. Lenstra. Factoring Integers with Elliptic Curves. *Annals of Mathematics*, 126, 1987.

[39] Dwayne Litzenberger. PyCrypto - The Python Cryptography Toolkit). `https://www.dlitz.net/software/pycrypto/`. Accessed on 17.2.2016.

[40] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. SP'15, 2015.

[41] Tarjei Mandt. Kernel Pool Exploitation on Windows 7. In *Black Hat Europe*, 2011.

[42] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. 1996.

[43] R. Owens and Weichao Wang. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. IPCCC'11, 2011.

[44] Peter Pessl, Daniel Gruss, Clementine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. SEC'16, 2016.

[45] Stephen C. Pohlig and Martin E. Hellman. An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance (corresp.). *IEEE Transactions on Information Theory*, 24(1), 1978.

[46] Shashank Rachamalla, Dabadatta Mishra, and Purushottam Kulkarni. Share-o-meter: An empirical analysis of KSM based memory sharing in virtualized systems. HiPC'13, 2013.

[47] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. NOZZLE: A Defense Against Heap-spraying Code Injection Attacks. SEC'09, 2009.

[48] Kaveh Razavi, Gerrit van der Kolk, and Thilo Kielmann. Prebaked uVMs: Scalable, Instant VM Startup for IaaS Clouds. ICDCS '15, 2015.

[49] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2), 1978.

[50] Jurgen Schmidt. JIT Spraying: Exploits to beat DEP and ASLR. In *Black Hat Europe*, 2010.

[51] Mark Seaborn. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. In *Black Hat USA*, 2015.

[52] Jean-Pierre Seifert. On authenticated computing and RSA-based authentication. CCS'05, 2005.

[53] Noam Shalev, Eran Harpaz, Hagar Porat, Idit Keidar, and Yaron Weinsberg. CSR: Core Surprise Removal in Commodity Operating Systems. ASPLOS'16, 2016.

[54] Prateek Sharma and Purushottam Kulkarni. Singleton: System-wide Page Deduplication in Virtual Environments. HPDC'12, 2012.

[55] Alexander Sotirov. Heap Feng Shui in JavaScript. In *Black Hat Europe*, 2007.

[56] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory Deduplication As a Threat to the Guest OS. EUROSEC'11, 2011.

[57] Paul C. van Oorschot and Michael J. Wiener. On Diffie-Hellman key agreement with short exponents. Eurocrypt'96, 1996.

[58] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side-Channels for Untrusted Operating Systems. SP'15, 2015.

Appendix A Cryptanalysis of Diffie-Hellman with Bit Flips

This section describes how one can break Diffie-Hellman by flipping a bit in the modulus. Similar to RSA, Diffie-Hellman cryptosystem performs computations modulo n. In the Diffie-Hellman key agreement scheme [21], however, the modulus n is prime or $s = \gamma_1 = 1$. It is very common to choose strong primes, which means that $q = (n-1)/2$ is also prime; this is also the approach taken by OpenSSH. Subsequently a generator g is chosen of order q. In the Diffie-Hellman protocol the client

chooses a random $x \in [1, n-1]$ and computes $g^x \bmod n$ and the server chooses a random $y \in [1, n-1]$ and computes $g^y \bmod n$. After exchanging these values, both parties can compute the shared secret $g^{xy} \bmod n$. The best known algorithm to recover the shared secret is to solve the discrete logarithm problem to find x or y using the GNFS, which has complexity $O(L_n[1/3, 1.92])$. For a 512-bit modulus n, the pre-computation cost is estimated to be about 10 core-years; individual discrete logarithms $\bmod n$ can subsequently be found in 10 minutes [3]. The current record is 596 bits [13]; again 1024 bits seems to be within reach of intelligence agencies [3].

By flipping a single bit of n, the parties compute $g^x \bmod n'$ and $g^y \bmod n'$. It is likely that recovering x or y is now much easier. If we flip the LSB, $n' = n - 1 = 2q$ with q prime and g will be a generator. In the other cases n' is a t-bit or $(t-1)$-bit odd integer; we conjecture that its factorization has the same form as that of a random odd integer of the same size. It is not necessarily the case the g is a generator $\bmod n'$, but with very high probability g has large multiplicative order.

The algorithm to compute a discrete logarithm in $Z_{n'}$ to recover x from $y = g^x \bmod n'$ requires two steps.

1. Step 1 is to compute the factorization of n'. This is the same problem as the one considered in Section 3.

2. Step 2 consists in computing the discrete logarithm of $g^x \bmod n'$: this can be done efficiently by computing the discrete logarithms modulo $g^x \bmod p_i'^{\tilde{\gamma}_i}$ and by combining the result using the Chinese remainder theorem. Note that except for the small primes, the $\tilde{\gamma}_i$ are expected to be equal to 1 with high probability. Discrete logarithms $\bmod p_i'^{\tilde{\gamma}_i}$ can in turn be computed starting from discrete logarithms $\bmod p_i'$ ($\tilde{\gamma}_i$ steps are required). If $p_i' - 1$ is smooth (that is, it is of the form $p_i' = \prod_{j=1}^{r} q_j^{\delta_j}$ with q_j small), the Pohlig-Hellman algorithm [45] can solve this problem in time $O\left(\sum_{j=1}^{r} \delta_j \sqrt{q_j}\right)$. If n' has prime factors p_i' for which $p_i' - 1$ is not smooth, we have to use for those primes GNFS with complexity $O(L_{p_i'}[1/3, 1.92])$.

The analysis is very similar to that of Section 3, with as difference that for RSA we can use ECM to find all small prime factors up to the second largest one p_2'. With a simple primality test we verify that the remaining integer is prime and if so the factorization is complete. However, in the case of the discrete logarithm algorithm we have to perform in Step 2 discrete logarithm computations modulo the largest prime p_1'. This means that if n' would prime (or a small multiple of a prime), Step 1 would be easy but we have not gained anything with the bit flip operation. It is known that the expected bitlength of the largest prime factor p_1' of n' is $0.624 \cdot t$ [36] (0.624 is known as the Golomb–Dickman constant). A second number theoretic result by Dickman shows that the probability that all the prime factors p_i' of an integer n' are smaller than $n'^{1/u}$ has asymptotic probability u^{-u} [20].

For $t = 1024$, the expected size of the largest prime factor p_1' of n' is 639 bits and in turn the largest prime factor of $p_1' - 1$ is expected to be 399 bits ($1024 \cdot 0.624^2$). Note that $p_1' - 1$ can be factored efficiently using ECM as in the RSA case. If $p_1' - 1$ has 639 bits, the probability that it is smooth (say has factors less than 80 bits) is $8^{-8} = 2^{-24}$, hence Pohlig-Hellman cannot be applied. We have to revert to GNFS for a 399-bit integer. However, with probability $2^{-2} = 1/4$ all the factors of n' are smaller than 512 bits: in that case the largest prime factor of $p_1' - 1$ is expected to be 319 bits, but again with probability 1/4 all prime factors are smaller than 256 bits. Hence with probability 1/16 GNFS could solve the discrete logarithm in less than 1 core hour.

For $t = 2048$, the expected size of the largest prime factor p_1' of n' is 1278 bits and the largest prime factor of $p_1' - 1$ is expected to be 797 bits – this is well beyond the current GNFS record. However, with probability $3 \cdot 10^{-3} = 0.037$ all prime factors of n' are smaller than 638 bits. Factoring $p_1' - 1$ is feasible using ECM, given that the its second largest prime factor is expected to be 134 bits. The largest prime factor of $p_1' - 1$ is expected to be 398 bits. The discrete logarithm problem modulo the largest factor can be solved using GNFS in about 1 core-month. With probability $4 \cdot 10^{-4} = 3.9 \cdot 10^{-3}$ all prime factors of n' are smaller than 512 bits, and in that case the largest prime factor of $p_1' - 1$ is expected to be 319 bits, which means that GNFS would require a few core-hours.

Even if it would not be feasible to compute the complete discrete logarithm there are special cases: if x or y have substantially fewer than t bits, it is sufficient to recover only some of the discrete logarithms $\bmod p_i'$ and the hardest discrete logarithm p_1 can perhaps be skipped; for more details, see [3, 57].

The main conclusion is that breaking discrete logarithms with the bit flip attack is more difficult than factorizing, but for 1024 bits an inexpensive attack is feasible, while for 2048 bits the attack would require a moderate computational effort, the results of which are widely applicable. It is worth noting that this analysis is applicable to the DH key agreement algorithm in use by OpenSSH, defaulting to 1536-bit DH group moduli in the current OpenSSH (7.2), bitflipped variants of which can be precomputed by a moderately equipped attacker, and applied to all OpenSSH server installations. The consequences of such an attack are decryption of a session, including the password if used, adding another attractive facet to attacks already demonstrated in this paper.