Security Now! #691 - 11-27-18

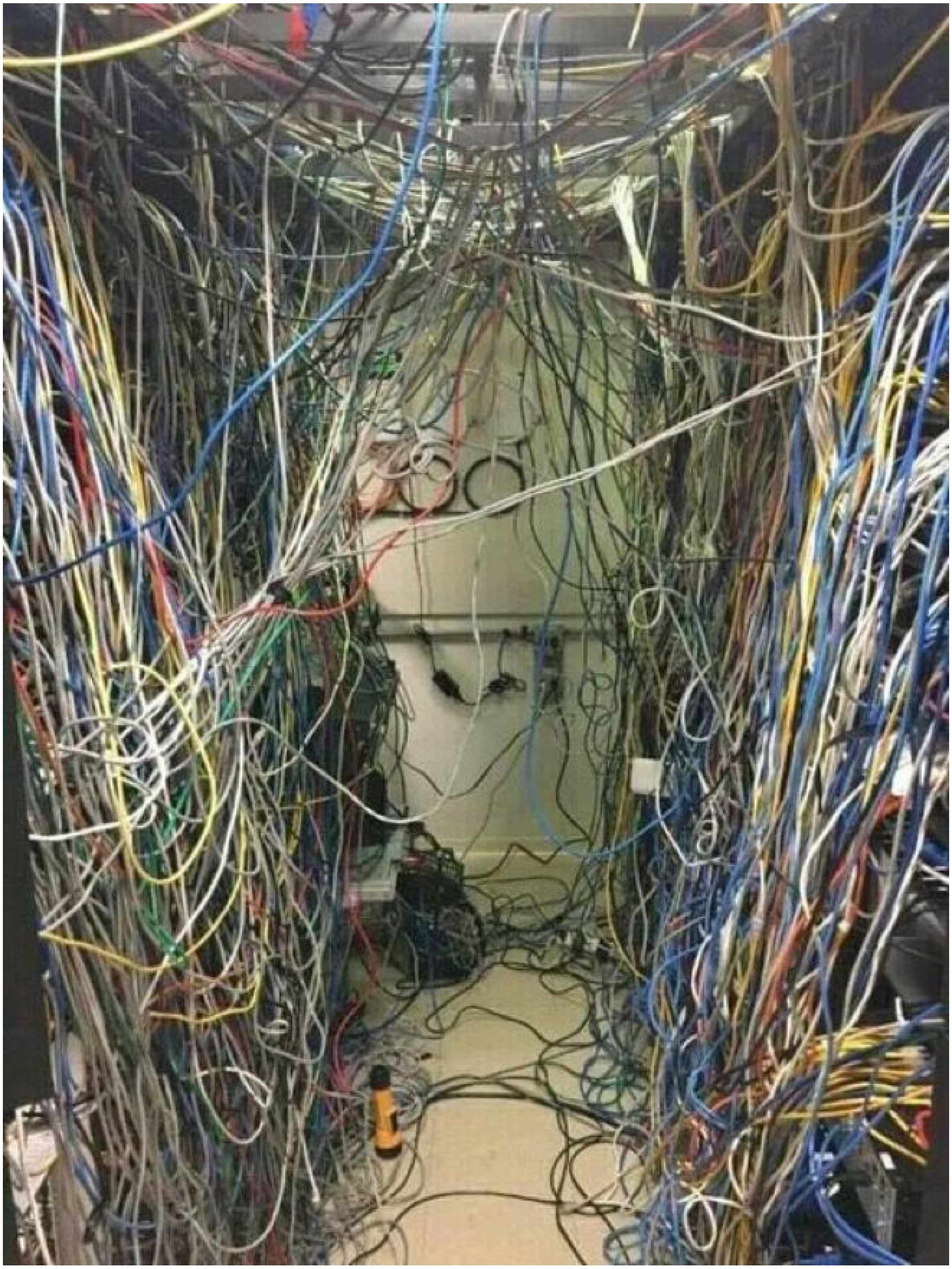
"ECCploit"

This week on Security Now!

Hackers and attackers apparently enjoyed their Thanksgiving, since this week we have very little news to report. But what we do have to discuss should be entertaining and engaging: Yesterday the US Supreme Court heard Apple's argument about why a class action lawsuit against their monopoly App Store should not be allowed to proceed; Google and Mozilla are looking to remove support for FTP from their browsers; and from our "what could possibly go wrong" department we have browsers asking for explicit permission to leave their sandboxes. We also have some interesting post-Troy Hunt "Are Password Immortal?" listener feedback from last week's topic. Then we will discuss the next step in the evolution of RowHammer attacks which do, as Bruce Schneier once opined, only get better… of in this case worse!

This week's Security Now! Picture of the Week Needed a page all to itself, so...

"The best laid plans"…



Security News

The supreme court and Apple's App store
https://www.supremecourt.gov/DocketPDF/17/17-204/46060/20180508135603183_17-204%20Apple%20v.%20Pepper%20%20-%20AC%20Pet.pdf

At issue is whether Apple's App store, through which it alone can offer and sell applications for its mobile devices, constitutes a monopoly and gives it monopoly power over all sales of iOS platform apps.

The lawsuit, known as Apple Inc. vs Robert Pepper, et al, has been around for almost a decade. It argues that Apple's 30% cut of all App Store sales revenue is excessive and anti-competitive and amounts to price gouging since consumers have no alternative other than to purchase apps for their devices at Apple's <quote> inflated <unquote> prices.

This issue is of interest to this podcast since we have often noted that consumers can act unwisely and may be poorly informed about the true dangers of obtaining applications which have not been rigorously scrutinized and vetted and curated.

As someone who very much wants to be able to trust the apps I use in my devices -- as much as it's possible to do so -- I, for one, have no complaint with the low, or no, cost of Apple's apps.

But as is often done in the law, Apple is not defending against that allegation, but rather whether consumers have any grounds for their complaint in the first place. In other words, at issue before the Supreme Court is whether the class of end-user consumers who are taking this action has "standing" to sue Apple over any of this... before even taking up the issue of whether Apple is overstepping the law by creating a fully captive store.

Apple has taken the position that only its developers, who are being charged a 30% sales commission on the retail sales of their apps, would be damaged if Apple's conduct was found to be unlawful... and that it would be up to them to complain, not the end user consumer.

The precedent the Supreme Court is revisiting was set back in 1977 in Illinois Brick Co. v. Illinois, a dispute in which the court ruled in favor of concrete brick manufacturers. The state of Illinois sued the brickmakers for allegedly inflating their prices, causing an increase in the the cost of public building projects.

The court ruled that even though the increased brick costs might hurt Illinois indirectly, only the contractors who actually bought the bricks had standing to sue. That established the so-called "Illinois brick doctrine," which says that only the direct purchaser of a good can collect damages from a monopoly holder.

The US Department of Justice Court of Appeals has already ruled that the Illinois Brick case was controlling here, whereas the plaintiffs have argued that the Court of Appeals misaplied the Illinois Brick Doctrine.

Apple argued yesterday that it is not directly selling apps to iPhone users. Rather, Apple said that it is acting as an agent for app developers, who ultimately are selling their wares to consumers. In exchange for the commission Apple takes on app sales, the company provides access to its vast user base and performs other services, such as malware detection.

What I believe that that Apple needs to remain the sole source and curator of their App Store apps.

File Transfer Protocol (FTP) browser support endangered
... does anyone care?

Bleeping Computer had an interesting posting about the endangered status of web browser support for the age-old FTP protocol.
https://www.bleepingcomputer.com/news/google/chrome-and-firefox-developers-aim-to-remove-support-for-ftp/

The trouble is that FTP is barely and rarely used by the mass of Chrome and Firefox users, the various attempts to secure FTP in the way that HTTP was secured have never taken hold, and there are some technical challenges:

I recall talking about the NAT traversal challenges of supporting FTP many years ago on this podcast. The trouble is, unlike HTTP, standard FTP uses a pair of TCP connections. One for control and a second connection for data. The second channel is allocated dynamically where the answering server telling the requesting client which service port to connect to for its data link. This created a problem for many firewall and NAT routers which adopted the practice -- which remains in use today -- of snooping on the FTP protocol control messages in order to determine which secondary data connections they need to allow inbound.

Naturally, this becomes impossible if the FTP control connection is encrypted using TLS since then the man-in-the-middle firewall cannot determine the TCP port number for the data connection negotiated between the client and FTP server. Consequently, in existing firewalled networks, a secure FTP ("FTPS") deployment would fail when an unencrypted FTP deployment would work.

And there's the whole issue of the web browser's attack surface being kept open and expanded for the use of a protocol that virtually no one uses through their browser any longer.

It's not that FTP isn't a useful application protocol for cross-Internet file transfer, but standalone full-featured FTP clients are available for every platform. Windows has long had an FTP client command line. Open an command line and enter "ftp" and you're at an FTP client command prompt. (Type "help" to obtain a large list of available commands and "quit" the client and return to the Windows command prompt.

And, moreover, insecure (and insecurable) FTP has become the lowest common denominator for Internet file transfer. There are many more modern and very secure means for transferring files across the Internet such as SCP (secure copy which uses SSH), SFTP which is another use of SSH for file transfer, FISH (File Transfer over SHell protocol).
So this is just a heads-up that we can eventually expect to be discussing the announcement that

version ‘X’ of Chrome and ‘Y’ of Firefox will be removing their native support for the venerable and increasingly obsolete FTP.

From our "What could possibly go wrong" department...
https://developers.google.com/web/updates/2018/11/writable-files

The Writable Files API: Simplifying local file access

What is the Writable Files API

Today, if a user wants to edit a local file in a web app, the web app needs to ask the user to open the file. Then, after editing the file, the only way to save changes is by downloading the file to the Downloads folder, or having to replace the original file by navigating the directory structure to find the original folder and file. This user experience leaves a lot to be desired, and makes it hard to build web apps that access user files.

The writable files API is designed to increase interoperability of web applications with native applications, making it possible for users to choose files or directories that a web app can interact with on the native file system, and without having to use a native wrapper like Electron to ship your web app.

With the Writable Files API, you could create a simple, single file editor that opens a file, allows the user to edit it, and save the changes back to the same file. Or a multi-file editor like an IDE or CAD style application where the user opens a project containing multiple files, usually together in the same directory. And there are plenty more.

Rather than filing this in the "What could possibly go wrong?" department, this should really have been in the "We should have seen this coming" department.

Our web browsers first began pulling crap from all over the Internet to fully monetize the pages we visited by mistake. Then they got us hooked on JavaScript so that nothing works anymore without it. But compared to native OS apps, JavaScript performance sucked. So first we got NaCL (native client) which was very clever but never achieved critical mass. Then consensus **was** reached on Web Assembly, so that code downloaded from random sites on the Internet could much more efficiently mine Monero on our machines while we were scratching ourselves and reading the "Okay, yeah, I understand that you'll be leaving cookies behind on my machine after I close this page I went to by mistake" acknowledgement banner.

Given this history of desperation to become a first-class native-born citizen, we **know** that the whole “sandboxing thing” had to really be chafing the web browser guys. They want to be real apps on our machines. They don't want to be stuck behind any safe enclosures. They have the speed... Now they want access! Maybe "What could possibly go wrong?" was the right place for this, after all!

<<Continuing from Google>>

Security considerations

The primary entry point for this API is a file picker, which ensures that the user is always in full control over what files and directories a website has access to. Every access to a user selected file (either reading or writing) is done through an asynchronous API, allowing the browser to potentially include additional prompting and/or permission checks.

The Writable Files API provides web developers with significant access to user data and has potential to be abused. There are both privacy risks, for example websites getting access to private data they weren't supposed to have access to, as well as security risks, for example websites able to modify executables, encrypt user data, and so forth. The Writable Files API must be designed in such a way as to limit how much damage a website can do, and make sure that the user understands what they're giving the site access to.

https://github.com/WICG/writable-files/blob/master/EXPLAINER.md

The following is a bit redundant, but it also elaborates their thinking…

Proposed security models

By far the hardest part for this API is of course going to be the security model to use. The API provides a lot of scary power to websites that could be abused in many terrible ways. There are both major privacy risks (websites getting access to private data they weren't supposed to have access to) as well as security risks (websites modifying executables, installing viruses, encrypting the users data and demanding ransoms, etc). So great care will have to be taken to limit how much damage a website can do, and make sure a user understands what they are giving a website access to. Persistent access to a file could also be used as some form of super-cookie (but of course all access to files should be revoked when cookies/storage are cleared, so this shouldn't be too bad).

The primary entry point for this API is a file picker (i.e. a chooser). As such the user always is in full control over what files and directories a website has access to. Furthermore every access to the file (either reading or writing) after a website has somehow gotten a handle is done through an asynchronous API, so browser could include more prompting and/or permission checking at those points. This last bit is particularly important when it comes to persisting handles in IndexedDB. When a handle is retrieved later a user agent might want to re-prompt to allow access to the file or directory.

Other parts that can contribute to making this API as safe as possible for users include:

- Limiting access to certain directories
  For example it is probably a good idea for a user agent to not allow the user to select things like the root of a filesystem, certain system directories, the users entire home directory, or even their entire downloads directory.

- Limiting write access to certain file types

Not allowing websites to write to certain file types such as executables will limit the possible attack surface.

- Other things user agents come up with.

We've often observed here that just because it's possible to do something doesn't automatically mean that it SHOULD be done. A better name for this proposal might be the *"Security Now four-digit podcast numbering assurance act of 2018"*... Since this guarantees a long future for this podcast!

Further discussion of this looming disaster can be found here:
https://discourse.wicg.io/t/writable-file-api/1433

If you care and can spend some time, you might want to put your two-cents in.

SQRL Bits

Dan Stevens @dan_iamai
Hi Steve. I listened to Security Now! episode 690 where you discuss Troy Hunt's blog post about passwords with great interest. I think good points were made on both sides. But I'm optimistic SQRL will eventually become widely used. When credit cards came out in the 1970s (before my time) I bet everyone was saying "they'll never replace cash", and they'd be right - cash is still widely used even in developed nations. Yet over the last few years (perhaps since contactless payments became available) I've needed cash less and less, so much so that now I leave my wallet at home and just carry my payment card! I think this shows if people are able to perceive the benefits of a new system over the status quo, a large portion will eventually switch to it. It's just a matter of time. Yes, passwords will never die, but they don't need to. If SQRL becomes widely available enough for me to 'leave my password vault at home', so to speak, that's good enough for me. I'm really looking forward to the official release of SQRL and excited to see what becomes of it.

Also... on the issue of friction:
Unlike **any** of the other multifactor systems, SQRL's setup is needed exactly once. After that, it's lower friction than anything else could be. Even setting up OTP codes is not zero-friction every time.

Don Williams @DonWmsGSO
Does SQRL have an option for the following scenario?
My employer contracts with a third party for services to be used by all employees. The third party permits access to only those users who can authenticate to the employer's security system. Therefore the third party is depending on the employer's authentication system.

James @Jr_McG1

Hi Steve,

I was delighted to hear my feedback being discussed regarding delegation of access on SQRL, and even more delighted to hear that it has been addressed already.

Listening to SN690, I was enjoying your discussion of Troy Hunts blog post. You mentioned the importance of a frictionless process being key for the adoption of any potential successor to the current security model of username/password. In particular you articulated the often encountered situation where people will refrain from leaving a comment on a blog post or site simply because they do not have an account, and the process of creating one falls into the 'too hard' drawer. Consequently, people just move along, and their insightful comment or reply is lost to the world.

Which got me thinking (again)…

The beauty of SQRL, as you have explained before is that it allows a person to anonymously identify AND assert said (anonymous) identity with a site. It allows them to do so quickly and easily. BUT - many many MANY sites base their entire business model on knowing something tangible about their visitors. Their name, e-mail address, a human friendly username etc. In the case of e-commerce then delivery addresses, billing addresses and credit card information would also be required in due course.

In short, most sites probably will not accept the creation of an account with just one anonymous handle. They will want more. My question (finally) is - is there a provision in the SQRL spec to allow this to be handled in a 'frictionless' manner.

I envision / propose something along the following lines:

1) On connecting with a site for the first time, the recognition is made that the user is unknown.

2) The site passes it's challenge to you, asking for you to provide and assert your identity.

3) Could (at this moment) the protocol ALSO allow the site (for the first time) to request other information about you that the site would want to know? Your preferred handle, your e-mail, your real name, your billing address and even (cringe) your CC details!

4) The SQRL application could then display (one time only) a list of the details that are requested by the site and which of those are considered by the site to be 'mandatory conditions of account creation', much the same way that Android requests app permissions at install time. The user could then choose what they wish to disclose to that site at that point, and those they wish to withold. Or, if they feel that the site is being 'too nosey' they can withdraw and the identity creation process is aborted. This would also likely assist in compliance with GDPR, as the user is being made explicitly aware of what information they are being asked to disclose and providing tacit consent in the process.

Personal details could be securely maintained in the SQRL application, with a provision to update personal details within the app, and have those changes pushed to relevant sites automatically upon subsequent login to that service. Users could also have information profiles, with a full set of details they provide to trusted e-commerce applications and other relatively sparse or anonymous profiles that they choose to disclose to less trusted sites or blogs.

This ensures that sites get the information they desire with maybe only one or two quick clicks being added to the user journey at the front end, rather than the annoying process of '20 questions' that currently has to be undertaken when we register with somebody new.

I have been listening to the development of SQRL for a number of years now and like everyone else listening want it to be a roaring success. I wonder what you think about this, and quietly hope that it's either already been solved, or can be quickly because I am looking forward to being able to use SQRL to replace all of my P@55w0rds!

Best Regards,

James.

Closing The Loop

Joe Petrakovich @makingloops
Shout out to @SGgrc Steve Gibson who I think gave me the idea to use Amazon S3 instead of Dropbox. Folder auto-sync job set up. Monthly bill reduced from $10 to $0.10

ECCploit: Rowhammer attacks against ECC protected memory

https://cs.vu.nl/~lcr220/ecc/ecc-rh-paper-sp2019-cr.pdf

As we'll recall, Professor Herbert Bos and team of researchers at VU Amsterdam were the first to demonstrate how potent and potentially practical RowHammer attacks could be.

At the time everyone took some relief from the observation that parity-protected RAM would tend to catch and thwart any single bit-flips, though not dual-bit flips. But that at least the presence of full ECC error-correcting RAM would provide much stronger protection.

Well... now, Professor Bos and a different team of researchers are back with the results of their examination of just how safe we should feel about ECC's protection from RowHammer attacks...

Their paper is titled: "Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks"

To give everyone a hint about what they found, the headlines in the technical press include:

"Rowhammer Data Hacks Are More Dangerous Than Anyone Feared"
"Not Even ECC Memory Is Safe Against Rowhammer Attacks"
"Potentially disastrous Rowhammer bitflips can bypass ECC"

First of all, let's remind ourselves about RowHammer...

- A matrix grid of cells
- Rows are read as one
- "Destructive Read" and rewrite.
- Refresh rate / higher is better, but lowers performance.

Abstract

Given the increasing impact of Rowhammer, and the dearth of other adequate hardware defenses, many in the security community have pinned their hopes on error-correcting code (ECC) memory as one of the few practical defenses against Rowhammer attacks. Specifically, the expectation is that the ECC algorithm will correct or detect any bits they manage to flip in memory in real-world settings. However, the extent to which ECC really protects against Rowhammer is an open research question, due to two key challenges. First, the details of the ECC implementations in commodity systems are not known. Second, existing Rowhammer exploitation techniques cannot yield reliable attacks in presence of ECC memory.

In this paper, we address both challenges and provide concrete evidence of the susceptibility of ECC memory to Rowhammer attacks. To address the first challenge, we describe a novel approach that combines a custom-made hardware probe, Rowhammer bit flips, and a cold-boot attack to reverse engineer ECC functions on commodity AMD and Intel processors. To address the second challenge, we present ECCploit, a new Rowhammer attack based on composable, data-controlled bit flips and a novel side channel in the ECC memory controller. We show that, while ECC memory does reduce the attack surface for Rowhammer, ECCploit still allows an attacker to mount reliable Rowhammer attacks against vulnerable ECC memory on a variety of systems and configurations. In addition, we show that, despite the non-trivial constraints imposed by ECC, ECCploit can still be powerful in practice and mimic the behavior of prior Rowhammer exploits.

Introduction

Originally designed to handle accidental and rare occurrences of data corruption in DRAM chips due to cosmic rays or electrical interference, Error-Correcting Code (ECC) memory is also perceived as one of the few effective bulwarks against Rowhammer attacks. These attacks exploit a vulnerability in DRAM hardware that allows attackers to flip bits in memory that should not be accessible to them. Since the discovery of the Rowhammer vulnerability in 2014, the security community has devised ever more worrying exploitation techniques. Starting with fairly simple, probabilistic corruption of page tables from native x86 code, researchers have extended the Rowhammer attack surface across all sorts of computing systems (including PCs, clouds and mobile devices), launching exploits from different environments (such as native C binaries and browser-based JavaScript), using a variety of processors (notably x86, ARM, and GPU), against a variety of targets (page tables, encryption keys, object pointers, repository URLs, and opcodes), in different types of memory (DDR3 and DDR4). As a result, Rowhammer has grown into a major security concern in real-world settings.

Not surprisingly, there has been much speculation on the effectiveness of ECC memory in deterring real-world Rowhammer attacks, often hypothesizing ECC memory would reduce Rowhammer to a denial-of-service vulnerability. As a result, practical Rowhammer exploits have thus far only targeted non-ECC-equipped platforms. However, once the uncommon case, ECC-equipped platforms are now on the rise, from large cloud providers (e.g., Amazon EC2) to high-end consumer platforms. In addition, ECC memory is increasingly deployed on low-power platforms such as mobile and IoT devices to drop the DRAM refresh rate below "safe" values and save power. It has therefore become important to quantitatively assess the effectiveness of ECC memory as a Rowhammer mitigation.

ECC is able to correct n bit errors (with $n \geq 1$) and detect cases where more than n bits have flipped, up to some maximum. For this purpose, ECC adds redundant ECC bits to every data word that "check" the other bits. The combination of the data bits and the ECC bits is known as a code word. ECC ensures that if any bit in a valid code word changes, it is no longer a valid code word. Thus, in a chipset with ECC memory, attackers may still use Rowhammer to cause a bit flip in physical memory, but the ECC mechanism immediately catches it on the first subsequent access, and flips it back. Since the probability of flipping exactly the right set of bits to turn one valid code word into a new valid code word using Rowhammer is extremely low, state-of-the-art Rowhammer attacks either fail, or trigger uncorrectable errors, leading to denial of service. Better still, modern processors apply additional memory reliability measures such as data masking (scrambling) to turn the data that the CPU really writes to main memory into pseudo-random patterns—making it even harder for an attacker to flip the right bits. The research question in this paper is whether the assumption is true that Rowhammer attacks are really not practical on ECC memory. In particular, we examine the strength of ECC in several modern chipsets and show that this is not the case: reliable attacks in real-world settings are harder, but still possible.

[[snip down to an exploration of just how mind-boggling this work is]]

To determine the exact protection offered by ECC, we must know the details of the ECC algorithms. Unfortunately, vendors such as Intel and AMD do not release these details. Moreover, to the best of our knowledge, no prior work has managed to reverse engineer the ECC functions. Important contributions of this paper are therefore the recovered ECC computation for popular chipsets and a detailed description of the techniques to reverse engineer other ECC algorithms.

A major challenge in examining a DRAM's susceptibility to Rowhammer on ECC memory, both for us and for attackers, is detecting the bit flips in the first place. How do we even know that we flipped a bit using Rowhammer, if the hardware automatically flips it back when we try to read it? Phrased differently, observing ECC errors is hard, precisely because the hardware is designed to hide them. To solve this problem, we describe a novel side channel that allows us to observe bit flips even when the error correction functionality flips them back when we read the corresponding memory location.

Armed with the ability to detect (correctable) bit flips and knowledge of a fully reverse engineered ECC algorithm, another challenge towards reliable attacks is to surgically trigger the “right” combination of bit flips in a single code word to bypass ECC. An invalid combination may be corrected or, worse, trigger uncorrectable errors and crash the system. To address this challenge, we develop a new Rowhammer attack technique based on composable, data-controlled bit flips. The key insight is that Rowhammer bit flips are data-dependent and, if we study how specific data patterns determine the triggering of individual bit flips, we can then reliably isolate/compose multiple bit flips by placing the “right” data patterns in memory. Our attack, termed ECCploit, relies on such insight to incrementally find an exploitable combination of bit flips in a code word and bypass ECC memory.