
# GhostRace

**Description:** What's the latest on that massive five-year-old AT&T data breach? Who just leaked more than 340,000 Social Security numbers, Medicare data and more, and what does that mean? Are websites honoring their cookie banner notification permissions? And why do we already know the answer to that question? What surprise has the GDPR's transparency requirements just revealed? And after sharing a bit of feedback from our listeners, we're going to go deeper into raw fundamental computer science technology than we have in a long time - and it may be inadvisable to operate any heavy equipment while listening to that part.

High quality (64 kbps) mp3 audio file URL: http://media.GRC.com/sn/SN-970.mp3
Quarter size (16 kbps) mp3 audio file URL: http://media.GRC.com/sn/sn-970-lq.mp3

SHOW TEASE: It's time for Security Now!. Steve Gibson's here. I'm at, well, not here, I'm at my mom's house. But we do have a great show. It's a propeller beanie show this week. He's going to explain what race conditions are and why it contributes to problems like Spectre and Meltdown. We've got the latest numbers on the massive five-year-old AT&T data breach. You won't believe how many customers are affected. You'll also be curious to know who just leaked more than 340,000 Social Security numbers, Medicare data, and more, and what you can do about it. And GDPR transparency requirements, you know, those cookie pop-ups? Are they honored? What do you think? We've got the deets, all coming up next on Security Now!.

**Leo Laporte:** This is Security Now! with Steve Gibson, Episode 970, recorded April 16th, 2024: GhostRace.

It's time for Security Now!, the show where we cover the latest news about what's happening in the world of the Internet in security. Bad guys, good guys, black hats, white hats. I am broadcasting from the East Coast this week. I'm at Mom's house. Steve Gibson never leaves his fortress of solitude. Hello, Steve.

**Steve Gibson:** Why would I ever leave? And Leo, this is not a green screen behind me, as Andy, or was it, no, it was one of your guys on MacBreak Weekly said. No, this is not a green screen. This is, yeah...

**Leo:** Well, Jason has the same background, whether he's real or not. And I always have to ask him. But if this were a green screen, could I do this?

**Steve:** That's a very nice little bunny. That's a very - that's a very nice bunny.

**Leo:** I have lots of tchotchkes behind me.

**Steve:** Yeah, I think I grew up with one of those clocks in my grandparents' house.

**Leo:** Oh, yeah. That's a classic...

**Steve:** Yeah.

**Leo:** Isn't that neat? There's a name for it.

**Steve:** That's neat, with a big golden one that - yup.

**Leo:** Is it a steeple clock? I can't remember. There is a name for it, yeah. Lots of antiques here in beautiful Providence.

**Steve:** And in fact your own mother could be considered an antique.

**Leo:** Yes, absolutely. Although she doesn't live here anymore. She's at the home. But she's loving it. It's an assisted living facility. She loves it. She's in memory care because she has no ability to form memories at all; you know? She can't, like she won't remember that I was there today. But she remembers everything else perfectly.

**Steve:** Isn't that...

**Leo:** So it's a really interesting conversation. She's jolly and happy as ever. She's not - I said, "Sometimes people get grumpy when they can't remember from day to day." She says, "No, no, it's good. Everything's fresh and new."

**Steve:** I had one of my very good lifelong friends - who's no longer on the planet. But we had been driving to each other's place to go see a movie together for decades. And one day he said, "I've lost my maps." And I said, "What?" And he said, "Everything's fine," he said, "but apparently I had a little stroke, and I don't know where anything is anymore."

**Leo:** Interesting. Interesting.

**Steve:** And it was so selective.

**Leo:** Precise, yeah.

**Steve:** I mean, it was like, it was just his mapping center.

**Leo:** Wow, yeah.

**Steve:** Was, like, lost. And so I said, "Well, okay, I'll come, I'll pick you up instead of the reverse."

**Leo:** It happens.

**Steve:** But it was - and that's - he said, "I lost my maps." He, like, realized, he didn't know where, how to go places.

**Leo:** Isn't that interesting.

**Steve:** But everything else was still...

**Leo:** The little vessel that burst just...

**Steve:** Breaking something.

**Leo:** One little zone of the brain, and it's gone, yeah.

**Steve:** Yeah. Okay, so this week, oh, boy. Probably the most consistent feedback that we received through the life of this podcast has been that our listeners love the deep technology dives. And, oh, get your scuba equipment.

**Leo:** Oh, good.

**Steve:** Because in order for today's revelation to make sense, today's podcast is titled "GhostRace" - "race" as in race conditions, and "ghost" as in Spectre. So what has been found by a team of researchers at AU Amsterdam and IBM is yet another problem with our industry's attempt to accelerate CPU performance. But in order to understand it, it's going to be necessary for our listeners to understand about preemptive multithreading environments and thread concurrency and synchronization of shared objects. And that may sound like a lot, but everyone's going to be able to go, oh, you know, "that's so cool" by the end of this podcast.

So we're going to talk about an update on that massive five-year-old AT&T data breach. Also, who just leaked an additional 340,000 Social Security numbers, Medicare data, and more.

**Leo:** Ay ay ay.

**Steve:** And what that means. I know. It's just, gosh, really? Are websites honoring their cookie banner notification permissions? And why do we already know the answer to that question? What surprise has the GDPR's transparency requirements just revealed? And then, after sharing a bit of feedback from our listeners, as I said, oh, boy. Given the feedback that I've had about what people like when we do it on this podcast, we're going to go deeper into raw fundamental computer science technology than we have in a long time.

**Leo:** Good.

**Steve:** And I'll just caution our listeners, it may be inadvisable to operate any heavy equipment while listening to that portion of today's presentation.

**Leo:** Well, I always call these, and I think you do, too, the propeller hat episodes.

**Steve:** Yep.

**Leo:** And we love the propeller hat episodes. So get your beanie ready. We're going to talk about some deep stuff. And of course our Picture of the Week this week, which I have not seen. So you'll get a clean take from me on this one. Always a fun part of the show.

**Steve:** Okay. So this is just a wonderful picture. I gave it the headline "This wonderful concept suffers from being a bit of an inside joke, but for those who understand it..."

**Leo:** I get it.

**Steve:** "...well, it's quite wonderful." What we're seeing is the result from some random guy in Nebraska no longer supporting his 20-year-old, you know, thankless maintenance of some random piece of open source software upon which the entire Internet's infrastructure has been erected.

**Leo:** It's a sequel to a famous xkcd cartoon, yeah.

**Steve:** Yes, yes, xkcd.com/2347. I put below the picture, I said, "If you need a hint, see xkcd.com/2347," where...

**Leo:** Which is a perfect little sandcastle of blocks and then giant things, all relying on one little tiny brick.

**Steve:** Yup, yup. And so anyway, today's picture is what happens if the brick gets pulled.

**Leo:** Or the maintainer retires, which has been a problem of late.

**Steve:** Yeah, exactly. I mean, we don't know how to launch shuttles anymore because all the shuttle guys are gone. And it's like, what do you mean, I need a different kind of epoxy for this tile on the bottom of the ship?

**Leo:** We don't know how to build spacesuits. Did you see that?

**Steve:** Yeah.

**Leo:** Nobody knows how to build those original Apollo spacesuits. They're starting from scratch.

**Steve:** Wow. Okay. So I wanted to follow up on our report from two weeks ago about the massive and significant AT&T data breach. Reading BleepingComputer's update, I was put in mind of the practice known as "rolling disclosure," where the disclosing party successively denies the truth, only admitting to what is already known when evidence of that has been presented. The trouble here, of course, is that AT&T is a massive publicly owned communications carrier that holds deep details about their U.S. consumer users. And an example of this incident suggests that they could be acting far more responsibly; and, moreover, that by not having done so for the past five years, the damage inflicted upon their own customers has likely been far worse than it needed to be. I mean, it's just - it's such bad practice.

Anyway, so BleepingComputer's reporting was headlined: "AT&T now says data breach impacted 51 million customers," and the annoyance in this piece's author is readily apparent. I've edited it just a bit to be more clear for the podcast, but BleepingComputer posted this.

They said: "AT&T is notifying 51 million former and current customers, warning them of a data breach that exposed their personal information on a hacking forum. However, the company has still not disclosed how the data was obtained. These notifications are related to the recent leak of a massive amount of AT&T customer data on the breach hacking forums that was first offered for sale for $1 million back in 2021.

"When threat actor ShinyHunters first listed the AT&T data for sale in 2021, AT&T told BleepingComputer that the collection did not belong to them, and that their systems had not been breached. Last month, when another threat actor known as MajorNelson" - I guess he's an "I Dream of Jeannie" fan - "MajorNelson leaked the entire dataset on the hacking forum, AT&T once again told BleepingComputer that the data did not originate from them, and that their systems had not been breached.

"After BleepingComputer confirmed that the data did belong to AT&T and DirecTV accounts, and TechCrunch reported" - as we've reported two weeks ago - "that AT&T's logon passcodes were part of the data dump" - a little hard to deny that one - "AT&T finally confirmed that the data did belong to them." You know, it's like, "Oh, that data." Uh-huh. Right.

Anyway, "While the leak contained information for more than 70 million people, AT&T now says that it impacted a total of 51,226,382 customers." So apparently they've been doing some research into that data. "AT&T's most recent notification states: 'The exposed information varied by individual and account, but may have included full name, email address, mailing address, phone number, social security number, date of birth, AT&T account number, and AT&T passcode. To the best of our knowledge, personal financial information and call history were not included.'" Oh, goodie. So call history not there, but

who cares? They said: "'Based on our investigation to date, the data appears to be from June 2019 or earlier.'

"BleepingComputer contacted AT&T, asking why there is such a large difference in impacted customers. They said: 'We are sending a communication to each person'" - that is, again, 51,226,382 people.

**Leo:** What?

**Steve:** Yeah.

**Leo:** 51 million?

**Steve:** 226,382.

**Leo:** Wasn't it, like, eight or nine million when this all began?

**Steve:** Yes. That's right. They've done a little more research and said, well, it's a little worse than we first told you.

**Leo:** 52 million?

**Steve:** That's the definition of a rolling disclosure.

**Leo:** Yes. Holy cow.

**Steve:** So, yeah, it's like, oh, that data. Oh.

**Leo:** Yeah, yeah, the ones with the PINs. Oh, yeah.

**Steve:** That's right.

**Leo:** Yeah.

**Steve:** Got a little hard to deny that one now.

**Leo:** Well, they think maybe a contractor - now, that's the thing. They don't know.

**Steve:** So, oh, but they said: "We're sending a communication to each person whose sensitive personal information was included. Some people had more than one account in the dataset, and others did not have sensitive personal information." Because

BleepingComputer said, wait a minute, you said more than 70 million, but now you're sending notices to 51,226,382.7. Anyway, "The company has still," writes BleepingComputer, "not disclosed how the data was stolen, and why it took them almost five years to confirm that it belonged to them, and to alert their customers."

So okay. I can understand that today, five years downstream, they may not know how it was stolen. It's at least believable that, if they didn't ever look, then they would have never found out; right? But the denial of the evidence they were shown many years ago is, you know, is difficult to excuse. They were first shown this in 2001. And it does appear that it's not going to be excused.

BleepingComputer said: "Furthermore, the company told" - meaning AT&T told - "the Maine Attorney General's Office that they first learned of the breach on March 26, 2024, yet BleepingComputer first contacted AT&T about it on March 17th and the information was for sale first in 2021.

"While it is likely too late, as the data has been privately circulating for years, AT&T is offering" - this is so big of them - "one year of identity theft protection" - whatever that means - "and credit monitoring services through Experian, with instructions enclosed in the notices. The enrollment deadline was set to August 30th of this year." So a few months from now. "But exposed people should move faster" - yeah, all 51 million of you - "to protect themselves.

"Recipients are urged to stay vigilant, monitor their accounts and credit reports for suspicious activity, and treat unsolicited communications with elevated caution. For the admitted security lapse and the massive delay in verifying the data breach claims and informing affected customers accordingly" - not surprisingly - "AT&T is facing multiple class-action lawsuits in the United States.

"Considering that the data was stolen in 2021, cybercriminals had ample opportunity to exploit the dataset and launch targeted attacks against exposed AT&T customers. However," finishes Bleeping Computer, "the dataset has now been leaked to the broader cybercrime community." That is, no longer privately circulated.

**Leo:** Oh, good.

**Steve:** Now everybody has it.

**Leo:** The broader cybercrime community.

**Steve:** That's right, "exponentially increasing the risk for former and current AT&T customers."

Okay. So just so we're clear here, and so that all of our listeners new and old understand the nature of the risk this potentially presents to AT&T's customers: Armed with the personal data that has been confirmed and admitted to having been disclosed - specifically Social Security numbers, dates of birth, physical addresses, and of course names - that is all that's needed to empower bad guys to apply for and establish new credit accounts in the names of creditworthy individuals. This is one of the most severe consequences of what is commonly known as "identity theft," and it is a true nightmare.

The way this is done is that an individual's private information is used to impersonate them when a bad guy applies for credit in their name, using what amounts to their

identity. That bad guy then drains that freshly established credit account and disappears, leaving the individual on the hook for the debt that has just been incurred. Since anyone might do this themselves, then claim that it really wasn't them, and that "It must have been identity theft, your Honor," proving this wasn't really them running, like, a scam of their own can be nearly impossible. And among other things, it can mess up someone's credit for the rest of their lives.

There's only one way to stop this, which is to prevent anyone, including ourselves, from applying for and receiving any new credit by preemptively freezing our credit reports at each of the three major credit reporting agencies. I know we've covered this before, at the beginning of the year, in fact. But it just bears repeating. And we may have some new listeners who haven't already heard this, or existing listeners who intended to take this action before, but never got around to it.

So here's my point: In this day and age of repeated, almost constant, inadvertent online information disclosure, it is no longer safe or practical for our personal credit reporting to ever be left unfrozen by default. Rather than freezing our credit if we receive notification of a breach that might affect us - and after all it could take five years, you know, and 51,226,382 of AT&T's current and former customers are now being informed five years after the fact, everyone should have their credit always frozen by default.

And in fact I can see some legislation in the future where somehow this is the policy because it's still wrong that by default it is open. But of course all the bureaus want it to be because that's how they make money is by selling access to our credit, which we never gave them permission to do, and they just took it. And then, once our credit is frozen by default, only briefly and selectively unfreezing it when a credit report does actually need to be made available for some entity to whom we wish and need to prove our credit worthiness.

Anyway, the last time I talked about this I created a GRC shortcut link for that podcast, which was #956. But I want to make it even easier to get to this page. So you can get all the details about how to freeze your credit reports by going to grc.sc/credit. So just put into your browser grc.sc, as in shortcut, grc.sc/credit. That will bounce your browser over to a terrific article at Investopedia that I verified is still there, still current, and still great information.

And on the heels of that, just to put an exclamation mark at the end of that news, another disclosure was just made and reported under the headline "Hackers Siphon 340,000 Social Security Numbers From U.S. Consulting Firm." CySecurity wrote: "Greylock McKinnon Associates (GMA) has discovered a data breach in which hackers gained access to 341,650 Social Security numbers. The data breach was disclosed last week on Friday on Maine's government website, where the state issues data breach notifications. In its data breach warning mailed to impacted individuals, GMA stated that it was targeted by an undisclosed cyberattack in May of 2023 and 'promptly took steps to mitigate the incident.'" Unfortunately, they didn't promptly apparently disclose it until now, almost a year later.

"GMA provides economic and litigation support to companies and government agencies in the U.S., including the DOJ, that are involved in civil action. According to their data breach notification, GMA informed affected individuals" - okay, they were informed - "that their personal information 'was obtained by the U.S. Department of Justice as part of a civil litigation matter' which was supported by GMA. The purpose and target of the DOJ's civil litigation are unknown. A Justice Department representative did not return a request for comment.

"GMA stated that individuals that were notified of the data breach are 'not the subject of this investigation or the associated litigation matters,' adding that the cyberattack 'does

not impact your current Medicare benefits or coverage. We consulted with third-party cybersecurity specialists to assist in our response to the incident, and we notified law enforcement and the DOJ. We received confirmation of which individuals' information was affected, and obtained their contact addresses on February 7, 2024.'" So, whoops, it was almost a year before people were notified.

"GMA notified victims that" - here it is - "'Your private and Medicare data was likely affected in this incident.'" Now, you've got to love the choice of the word "affected." You mean as in "obtained by malicious hackers?" Yeah. Anyway, they said: "...which included names, dates of birth, home addresses, some medical and health insurance information, Medicare claim numbers, and Social Security numbers.

"Finally, it remains unknown why GMA took nine months to discover the scope of the incident and notify victims. GMA and its outside legal counsel" - been very active lately - "Linn Freedman of Robinson & Cole LLP, did not immediately respond to a request for comment."

So as I noted above, this is now happening all the time. It is no longer safe to leave one's credit report unfrozen, and freezing it everywhere will only take a few minutes. Since you won't want to be locked out of your own credit reporting afterwards, however, be sure to securely record the details you'll need when it comes time to briefly and selectively unfreeze your credit in the future.

And Leo, let's take a break before we talk about cookie notice compliance or lack thereof.

**Leo:** By the way, you mentioned in your credit report you didn't give them permission. You did. But it was in the very, very fine print of that credit card agreement of every agreement you make. They put it in the fine print that we will submit information to the credit reporting bureaus.

**Steve:** Okay, thank you. I'm glad you caught that.

**Leo:** Yeah. So you did agree to it. But, you know, I mean, the truth is it's how the world works because it is a sensible system for having somebody who wants to lend you money have some way of verifying that you're a good prospect.

**Steve:** Yup.

**Leo:** So every time you buy a car, rent a house, get a new phone, cell phone, set up a cell phone carrier, they do that. They pull those credit reports. I freeze mine, though, and I think you're absolutely right. Everything I have is frozen. And thanks to a federal law, they can't charge you to unfreeze it. They used to charge you 35 bucks. In some states it was more to unfreeze it. They can't do that anymore.

**Steve:** Yeah. Yeah, and to add confusion, there's also the term "lock," which does not mean the same as "freeze."

**Leo:** Yeah, not the same, yeah.

**Steve:** So it's freezing your report, not locking your report.

**Leo:** Freeze it, don't lock it.

**Steve:** Yup.

**Leo:** Now, there were some good show titles in there, but I know you already have one, so I won't belabor it. But "Freeze It, Don't Lock It" would be a good show title.

**Steve:** Okay. So the following very interesting research was originally slated to be this week's major discussion topic.

**Leo:** Ah.

**Steve:** Uh-huh. But after I spent some time looking into the recently revealed GhostRace problem, I wanted to talk about that instead since it brings in some very cool fundamental computer science about the problem of "race conditions" which, interestingly, in our 20 years of this podcast - well, we're in our 20th year - we've never touched on race conditions. So we're going to resolve that today.

But what this team of five guys from ETH Zurich discovered was very interesting, too. So here's that. They began asking themselves: "When we go to a website that presents us with what has now become the rather generic and GDPR-required Cookie Permission pop-up, do sites where permission is not explicitly granted and cookie use is explicitly denied actually honor that denial?" And with that open question on the table, the follow-up question was: "Can we arrange to create an automated process to obtain demographic information about the cookie permission handling of the top 100,000 websites?"

Well, we already know the answer to the second question. That's "yes." You know? They figured out how to automate this data collection. And the results of their research will be presented during the 33rd USENIX Security Symposium, which will take place this coming August 14th through the 16th in Philadelphia. Their paper is titled "Automated Large-Scale Analysis of Cookie Notice Compliance." And for anyone who's interested, I have a link to their PDF research and the presentation prepub in the show notes.

Perhaps it won't come as any shock that what they found was somewhat disappointing. Did they find that 5% of the 97,090 websites they surveyed did not obey the sites' visitors' explicit denial of permission to store privacy invading cookies? No. Was the number 10%? Nope, not that either. How about 15? Nope, still too low. Believe it or not, 65.4% of all websites tested, so just shy of fully two thirds of all websites tested - two thirds - do not obey their visitors' explicit privacy requests. They wrote: "We find that 65.4% of websites do not respect users' negative consent, and that top-ranked websites are more likely to ignore users' choice, despite having seemingly more compliant cookie notices."

The Abstract of their paper says: "Privacy regulations such as the General Data Protection Regulation (GDPR) require websites to inform EU-based users about non-essential data collection and to request their consent to this practice. Previous studies have documented widespread violations of these regulations. However, these studies provide a limited view of the general compliance picture. They're either restricted to a subset of notice types, detect only simple violations using prescribed patterns, or analyze

notices manually. Thus, they're restricted both in their scope and in their ability to analyze violations at scale.

"We present the first general, automated, large-scale analysis of cookie notice compliance. Our method interacts with cookie notices, in other words, by navigating through their settings. It observes declared processing purposes and available consent options using Natural Language Processing and compares them to the actual use of cookies. By virtue of the generality and scale of our analysis, we correct for the selection bias present in previous studies focusing on specific Content Management Platforms. We also provide a more general view of the overall compliance picture using a set of 97,000 websites popular in the EU. We report, in particular, that 65.4% of websites offering a cookie rejection option likely collect user data despite explicit negative consent."

Okay. So I suppose we should not be surprised to learn that what's hiding behind the curtain is not what we would hope. Two thirds of companies in general - the larger they are, the worse is their behavior - are simply blowing off their visitors' explicit requests for privacy. And this is in the EU, where the regulation is present and would be more likely to be strongly enforced. Except who would know if the regulation was being ignored unless you checked. So not only are we now being hassled by the presence of these cookie permission banners, but two out of every three required "do nothing clicks." When we say we want you not to do this, they have no actual effect.

The researchers discovered additional unwanted behavior which they termed "dark patterns." Specifically, 32% are missing the notice entirely. 56.7% don't have a reject button. So they display the notice, but they don't give you the required option to opt out. And then there's those 65.4% that do have the button, but they ignore it. Then there's 73.4% with implicit consent prior to interaction, meaning they assume you are consenting, and they do things in 73.4% of the cases prior to giving you the option to not have them do that. There is a category, implicit consent after close, that's there in 77.5% of the instances. Undeclared purposes in 26.1, so they don't tell you what's going on. There's interface interference in two thirds of the cases, 67.7%; and a forced action in almost half, 46.5. So bottom line, you know, this is all a mess.

What this means is that the true enforcement of our privacy cannot be left to lawmakers and their legislation, nor even to websites. New technology needs to be brought to bear to take this out of the hands entirely of third parties. And I know it seems insane for me to be saying this, but this really is what Google's Privacy Sandbox has been designed to do. And as with all change, advertisers and websites are going to be kicking and screaming, and they're not going to go down without a fight.

But the advertisers are also, reluctantly, currently in the process of upgrading their ad delivery architectures for the Privacy Sandbox because they know that, with Chrome commanding two thirds market share, they're going to have to work within this brave new world that's coming soon, you know, and real soon, as in later this year. Google's not messing around this time. They've given everyone years' of notice. Change is coming. And I say thank goodness because it's obvious that just saying, oh, you know, you've got to get permission, well, looks like websites, most of them, are asking for permission. It turns out two thirds of them are ignoring it when we say no.

And speaking of the GDPR, which created this accursed cookie pop-up legislation which currently plagues the Internet, holding to the "there's two sides to every coin" rule, the requirements for disclosure that's also part of the GDPR legislation occasionally produces some startling revelations when conduct that parties would have doubtless much preferred to remain off the radar are required to be seen. Get a load of this one.

In this case, we have the new Outlook app for Windows. Thanks to the GDPR, users in Europe who download the Outlook for Windows app will be greeted with a modal pop-up

dialog that displays a user agreement and requires its users' consent. So far, so good. The breathtaking aspect of this is that the dialogue starts right out stating: "We and 772 third parties process data to store and/or access information on your device."

**Leo:** Oh, only 700, huh?

**Steve:** 772 third parties that Microsoft is sharing their users' data with.

**Leo:** More computers is slow, but lord.

**Steve:** Wow.

**Leo:** Wow.

**Steve:** "To develop and improve products, personalize ads and content, measure ads and content, derive audience insights, obtain precise geolocation data" - yes, we know right where you are, and we're telling everyone - "and identify users through device scanning." And, no, that was not a typo. It says "We and 772 third parties."

**Leo:** At least they're honest. I mean...

**Steve:** Well, they have no choice; right? In the EU.

**Leo:** Yeah, that's awesome.

**Steve:** You know, I don't know what to think about that. It's mindboggling. For one thing, it's disappointing that only those in the EU get to see this. U.S. regulators are not forcing the same transparency requirements upon Microsoft, and Microsoft certainly doesn't want to show any more of this than they're forced to.

**Leo:** No. Yeah.

**Steve:** You know, earlier we were talking about the confidentiality of our data and the challenge that presents. Having Microsoft profiting from the sale of the contents of its users' email seems annoying enough. At the same time, you know, there's Gmail, and they're doing some of that, too. But when there are, by Microsoft's own forced admission, 772 such third parties who all, presumably, receive paid access to this data, doesn't it feel as though Microsoft should be paying us for the privilege of access to our private communications, which is apparently so valuable to them? Instead, we get free email. Whoopee. Wow.

**Leo:** Does Google do anything like that with Gmail? It's got to be - that's wild.

**Steve:** No. I've never - whoa. Oh, you mean in the EU. That's a good question, whether they have had to start that, too. Yeah.

**Leo:** I think it's the case, although I'm not a lawyer, that this is in the EU, all of this; that even though we all in the U.S. and everywhere else in the world see these cookie monster announcements, they don't - they're not technically required to do them for us. It's only for EU people.

**Steve:** Well, as I understand it, if an EU person visits your website...

**Leo:** No, it's still not liable because it has to be a company doing - well, I don't know, that's a good question. There was a very interesting piece which I've mentioned on other shows. And of course the guy's not in [indiscernible] either. Blog piece that said no one has to do this except EU companies. They're the only companies liable for this. And he points out Amazon does not. Right?

**Steve:** Interesting.

**Leo:** You've never seen it on Amazon.

**Steve:** Interesting.

**Leo:** You don't see it on a lot of sites.

**Steve:** Not mine.

**Leo:** I don't see cookies on Google sites, come to think of it.

**Steve:** GRC doesn't have it, although I'm not collecting any information and using cookies that way.

**Leo:** Well, we do it on TWiT because there's a whole secondary business of people who go around saying, "You'd better do this for compliance," and "Pay me, and I'll do it for you." And they said otherwise you're going to be in trouble. So we have a lot of privacy stuff on our site. We don't - you can't log into our site. But we, as every site does, we have cookies. We have loggers, you know, I mean, I do it on my personal website. I say, "Look, the only cookie I have is whether you like the light mode or dark mode. So suck it. If you don't like that, too bad." I mean, that's why this whole thing is so silly. I just, it's so silly.

**Steve:** Well, it's probably going to go away because if Google has their way...

**Leo:** Maybe.

**Steve:** ...and Google generally gets their way, you know, they've got the demographics, the market share. They really are terminating third-party cookies.

**Leo:** We've done this story before because remember you talked about Do Not Track, which is in the spec. Everybody has it. Nobody honors it.

**Steve:** It's coming back. It's on its way back.

**Leo:** I think it would be nice to see, yeah.

**Steve:** Yeah, yeah. Well, and actually, again, what's happening is that the next-generation technology will not track.

**Leo:** Right.

**Steve:** By siloing each site, the tracking becomes, or the tracking happens because traditionally browsers have had one big cookie jar. As soon as you start creating individual per-domain cookie jars, then each domain is welcome to have its own local set of cookies, but they're not visible from any other domain. And that terminates tracking.

**Leo:** Right.

**Steve:** Unfortunately, that's why we're now doing the, oh, please join our email, you know, join our community, give us your email. And so basically first-party cookies are being used to track through the browser.

So, okay. We have a bit of feedback from our listeners. And then, boy, have we got a fun techie episode. Okay. Let's see. I got: "@SGgrc. Been hearing you talk about the physical input requirement for security setting modifications," meaning like push the button on the browser if you want to make a security change, and that keeps somebody in Russia from being able to push a button on your browser. This person said: "Just wanted to mention Fritz and Fritz!Box, a common German home router manufacturer who's been doing this for a while." And he said: "Annoying, yes. Effective, also yes."

So, you know, when we've talked about this previously I haven't mentioned that many SoHo routers, probably everyone's router if it's at all recent, have that dedicated WPS hardware button that is a physical button for essentially the same reason. It allows a network device to be persistently connected to a router without the need to provide the device with the router's WiFi SSID name and password. That button being physical requires someone to prove their presence at the router by pressing the button. So the notion of doing something similar to similarly protect configuration changes makes a lot of sense.

But then another listener suggested, this was Rob Woodruff who tweeted. He said: "I'm listening to the part of SN-969 where you're talking about the five security best practices. And it occurs to me that, regarding the requirement for a physical button press, it won't be long before we start seeing cheap, Chinese-made, WiFi-enabled IoT button pressers on Amazon. What could possibly go wrong?"

**Leo:** They exist, by the way. Stacey Higginbotham used to talk about them.

**Steve:** A little thing that presses a button?

**Leo:** Yeah. It's a WiFi-enabled little finger that goes like this.

**Steve:** Oh.

**Leo:** And for lights that don't - you can't change or whatever the - yeah, yeah. So it exists.

**Steve:** Too funny.

**Leo:** I don't know if it would work with that WPS-type button, but...

**Steve:** Yeah. And actually I think I remember watching that This Week in Google when you guys talked about that.

**Leo:** Yeah, yeah, yeah.

**Steve:** And I thought that was...

**Leo:** Stacey used them.

**Steve:** Wow.

**Leo:** She liked them.

**Steve:** Remember that thing when we were growing up, the little black box that you would stick a penny on, and the hand would reach out and grab the penny?

**Leo:** Right. But my favorite one, though, there's varieties of these, you'd flip the switch, the hand comes out and turns the switch off.

**Steve:** Yes, yes.

**Leo:** Loved those.

**Steve:** Okay. So the downside of the physical button press requirement, you know, obviously is its inherent inconvenience. You know? But that's also the source of its

security. It would be nice if we could have the security without the inconvenience, but it's unclear how we could go about doing that. But to Rob's point, even if a user chose to employ a third-party remote control button presser, that would still provide greater security than having no button at all. For one thing, it would be necessary to hack two very different systems, kind of like having multifactor authentication.

But the most interesting observation, I think, is that generic remote attacks against an entire class of devices that were known to be "button protected" would never even be launched in the first place because it would be known that they could not succeed. So even a router that technically breaks the rules with an automated button presser gains the benefit of what is essentially herd immunity because, you know, attackers wouldn't bother attacking that model router because they're unattackable, thanks to having the physical button.

Rami Vaspami tweeted: "@SGgrc I don't see the benefit if NetSecFish kept quiet." So this person's talking about that premature, well, the disclosure of the incredible amount of old D-Link NAS devices. So they write: "This would have been quietly exploited in perpetuity. Instead, now, everyone knows to deprecate those devices, and the issue has an end in sight. Better to know."

Okay. If, indeed, current D-Link NAS owners could have been notified as a consequence of NetSecFish's disclosure on GitHub, then okay, maybe. But they still would have had to act quickly since the attacks against these D-Link NAS boxes required less than two weeks to begin. But the problem is that nothing about NetSecFish's disclosure on GitHub suggests that even a single end-user of these older but still in use D-Link NAS devices would have ever received notification. Notification from whom? You know, he posted this on GitHub. That's not notifying all those D-Link NAS end-users. Why would any random user know what's been posted in someone's GitHub account? D-Link said, "Sorry, but those are old devices for which we no longer accept any responsibility."

But we know who does know what's been posted to someone's random GitHub account, since it took less than two weeks for the attacks to begin. So I would argue that it is not "better to know" when it's by far predominantly bad guys who will be paying attention and who will be doing the knowing.

And finally, John Moriarty wins the "feedback of the week award." He tweeted: "Catching up on last week's Security Now! and hearing @SGgrc lament the fans spinning up and resources being swamped by Google's Chrome bloat while it was apparently 'doing nothing.' Perhaps it was busy running and completing ad auctions."

**Leo:** Oh.

**Steve:** So yes, indeed. It's quite true that our browsers are going to be much busier once responsibility for ad selection has been turned over to them. Although, you know, our computers are so overpowered now it's ridiculous. You know, using a browser shouldn't even wake the computer up.

I don't have any big SpinRite news this week. Everything continues to go well, and I'm working to update GRC's SpinRite pages just enough to hold us while I get email running, and then back over to work on SpinRite's documentation. I've been seeing more good reports of SSD original performance recovery. While catching up in Twitter, I did run across a good question about SpinRite. Paul, tweeting from @masonpaulak, he said: "Hi, Steve. Long-time podcast listener and SpinRite owner. Thanks for the many gems of info over the years." And boy, have we got a good one for this podcast coming up. "Over the weekend," he said, "I saw the RAM test in SpinRite 6.1. Please, would you give some

insight into how long I should let it run to test 1GB? And am I wearing it out as writing to an SSD would? Thanks, Paul."

I don't think I remembered to respond to the wearing it out question. No. DRAM absolutely, thank goodness, does not wear out when it's being read and/or written. Okay. So but to his question of how long you should run it, the best answer to this is really quite frustrating because the best answer is "the longer the better." You know, now, that said, I can clarify a couple of points. The total amount of RAM in a system has no bearing on this, since SpinRite is only testing the specific 54MB that it will be using once the testing is over. While the testing is underway, a test counter is spinning upwards on the screen, and a total elapsed testing time is shown. For each test it fills 54MB with a random pattern, then comes back to reread and verify that memory was stored correctly. So the longer the testing runs, the greater the likelihood that it will find a pattern that causes some trouble.

The confusion, and Paul's question, comes from the fact that the new startup RAM test can be interrupted. It doesn't stop by itself. It just goes. And eventually the user's going to interrupt it in order to get going with SpinRite. So when should that be? Best practice, if it's feasible, would be for any new machine where SpinRite 6.1 has never been run before, to start it up and allow it to just run on that machine overnight, doing nothing but testing its RAM. In the morning, the screen will almost certainly still be happy and still showing zero errors. I mean, that's by far the most likely case. But the screen might have turned red, indicating that one or more verification passes failed.

After I asked that first guy who was having weird problems that made no sense to try running MemTest86, the venerable, you know, full system RAM test that's been around forever, it reported errors. So I decided I needed to build that into SpinRite, just to be safe. Users do not need to use it at all, but it's there. And several other of SpinRite's early testers were surprised to have SpinRite detect previously unsuspected memory errors on some of their machines. So SpinRite's built-in RAM memory test is sort of useful all by itself, just for that. But once it's been run for a few hours on any given machine, then it can just be bypassed quickly from then on, and you don't need to worry about it.

Okay. So if anybody is doing brain surgery right now...

**Leo:** Driving a large truck...

**Steve:** Yes, close the skull. Put down your tools.

**Leo:** Pull over.

**Steve:** Yes. The first question to answer is "What's a Race Condition?" A race condition is any situation which might occur in hardware or software systems where outcome uncertainty or an outright bug is created when the sequence of closely timed events, in other words a race among events, determines the effects of those events. For example, in digital electronics, it is often the case that the state of a set of data lines is captured when a clock signal occurs. This is typically known as "latching" the data lines on a clock. But for this to work reliably, the data lines need to be stable and settled when the clocking occurs. If the data lines are still changing state when the clock occurs, or if the clock occurs too soon, a race condition can occur where the data that's latched is incorrect.

Here's how Wikipedia defines the term. They write: "A race condition or race hazard is the condition of an electronics, software, or other system where the system's substantive behavior is dependent on the sequence or timing of other uncontrollable events, leading to unexpected or inconsistent results. It becomes a bug when one or more of the possible behaviors is undesirable. The term 'race condition' was already in use by 1954, for example in David A. Huffman's doctoral thesis 'The synthesis of sequential switching circuits.' Race conditions can occur especially in logic circuits, multithreaded, or distributed software programs."

Okay. Race conditions are also a huge issue with modern software systems because our machines now have so many things going on at once where the independent actions of individual threads of execution must be synchronized. Any failure in that synchronization can result in bugs or even serious security vulnerabilities.

To understand this fully, we need to switch into computer science mode. And as I said at the top of the show, I often receive feedback from our listeners wishing that I did more of this, but I don't often encounter such a perfect opportunity as we have today. Back when we were talking about how processors work during one of our early sets of podcasts on that topic, we talked about the concepts of multithreading and multitasking.

In a preemptive multithreaded environment, "multithreaded" meaning multiple threads of execution happening simultaneously, the concept of a thread of execution is an abstraction. The examples I'm going to draw assume a single core system, but they apply equally to multicore systems. When a processor is busily executing code, performing some task, we call that a "thread of execution" since the processor is threading its way through the system's code, executing one instruction after another. After a certain amount of time, a hardware interrupt event occurs when a system clock ticks. This interrupts that thread of execution, causing it to be suspended and control to be returned to the operating system.

If the operating system sees that this thread has been running long enough, and that there are other waiting threads, the OS decides to give some other thread a turn to run. So it does something known as a "context switch." A thread's context is the entire state of execution at any moment in time. So it's all of the registers the thread is working with and any other thread-specific volatile information. The operating system saves all of that thread context, then reloads another thread's saved context, which is to say everything that had been previously saved, you know, from other thread when it was suspended. The OS then jumps to the point in code where that other thread was rudely interrupted the last time it was preempted, and that other thread resumes running just as though nothing had ever happened.

And that's a critical point. From the viewpoint of the many threads all running in the system, the threads themselves are completely oblivious to all of this going on above them. As far as they're concerned, they're just running and running and running without interruption, as if each of them were all alone in the system. The fact that there's an overlord frantically switching among them is unseen by the threads. I just opened Task Manager on my Windows 10 machine. It shows that there are 2,836 individual threads that have been created and are in some state of execution right at this moment. That's a busy system.

Okay. So now let's dive deeper into very cool and very important coding technology. The reason for this dive will immediately become clear once we look at the vulnerabilities that the researchers from VU Amsterdam and IBM have uncovered in all contemporary processor architectures - not just Intel this time, or AMD, also ARM and RISC-V - any architecture that incorporates - get ready for it - speculative execution technology, as all modern processor architectures do.

Okay. Imagine that two different threads running in the same system share access to some object. Anyone who's been around computing for long will have heard the term "object oriented programming." An "object" is another abstraction that can mean different things in different contexts. But what's important is that it's a way of thinking about and conceptually organizing complex systems that often have many moving parts. So when I use the term "object," I'm just referring to something as a unit. It might be a single word in memory, or it might be a large allocation of memory, or it could be a network interface, whatever. In the examples we'll be using, our object of interest is just a word of RAM memory that stores a count of events. So this object is a simple counter.

And in our system, various threads might, among many other things, have the task of incrementing this counter from time to time. So let's closely examine two of those threads. Imagine that the first thread decides it wishes to increment the count. So it reads the counter's current value from memory into a register. It might examine it to make sure that it has not hit some maximum count value. Then it increments the value in the register and stores it back out into the counter memory. No problem. The shared value in memory was incremented. But remember that, in our hypothetical system, there are at least two separate threads that might be able to do this.

So now imagine a different scenario. Just like the first time, the first thread reads the current counter value from main memory into one of its registers. It examines its value and sees that it's okay to increment the count. But just at that instance, at that instant, the system's thread interrupt occurs; and that first thread's operation is paused to give other threads the opportunity to run. And as it happens, that other thread that may also decide to increment that counter decides that it wishes to do so. So it reads that shared value from memory, examines it, increments its register, and saves it back out to memory. And as luck would have it, for whatever reason, while that second thread is still running, it does that four more times. So the counter in memory was counted up by a total of five times. Then after a while that second thread's time is up, and the operating system suspends it to give other threads a chance to run.

Eventually, the operating system resumes our first thread from the exact point where it left off. Remember that before it was paused by the operating system, it had already read the counter's value from memory, examined it, and decided that it should be incremented. So it increments the value in its register and stores it back out to memory, just as it did the first time.

But look what just happened. Those five counter increments that were performed by the second thread, which shares access to the counter with the first thread, were all lost. The first thread had read the counter's value into a register. Then, while it was suspended by the operating system, the second thread came along and incremented that shared storage five times. But when the first thread was resumed, it stored the incremented counter value that it had previously saved back out, thus overwriting the changes made by the second thread.

This is a classic and clear example of a race condition in software. It occurred because of the one in a million chance that the first thread would be preempted at the exact instant, between instructions, when it had made a copy of some shared data that another thread might also modify, and it hadn't yet written it back to the shared memory location. The picture I painted was deliberately clear because I wanted to highlight the problem. But in the real world it doesn't happen that way, and these sorts of race condition problems are so easy to miss.

Imagine that two different programmers working on the same project were each responsible for their piece of code, and each of them increments that counter. If they didn't realize that the other had code that might also choose to increment that counter, this would introduce an incredibly subtle and difficult to ever find bug. Normally,

everything would work perfectly. The system's tested. Everything's great. But then every once in a blue moon the code, which looks perfect, would not do what it was supposed to. And the nature of the bug is so subtle that coders could stare at their code endlessly without ever realizing what was going on.

Last Tuesday was Patch Tuesday, and Microsoft delighted us all by repairing another 149 bugs, which included two that created vulnerabilities which were under active exploitation at the time. Microsoft's software does not appear to be in any danger of running out of bugs to fix, and race conditions are one of the major reasons for that. How many times have we talked about so-called "use-after-free" bugs, where malicious code retains a pointer to some memory that was released back to the system and is then able to use that pointer for malicious purposes? More often than not, those are deliberately leveraged race condition bugs.

Okay. So now the question is, as computer science people, how do we solve this race condition problem? The answer is both tricky and gratifyingly elegant. We first introduce the concept of ownership, where an object can only be owned by one thread at a time, and only the current owner of an object has permission to modify its value. In the example I've painted, the race condition occurred because two threads were both attempting to alter the same object, the shared counter, at the same time. If either thread were forced to obtain exclusive ownership of the counter until it was finished with it, the competing thread might need to wait until the other thread had released its ownership, but this would completely eliminate the race condition bug.

Okay. So at the code level, how do we handle ownership? We could observe that modern operating systems typically provide a group of operations known as synchronization primitives. They're called this because they can be used to synchronize the execution of multiple threads that are competing for access to shared resources. The primitive that would be used to solve a problem like this one is known as a "mutex," which is short for "mutual exclusion," meaning that threads are mutually excluded from having simultaneous access to a shared object.

So by using an operating system's built-in mutex function, a thread would ask the operating system to give it exclusive ownership of an object and be willing to wait for it to become available. If the object were not currently owned, then it would become owned by that first requesting thread, and that owning thread could do whatever it wished with the object. Once it was finished, it would release its ownership, thus making the object available to another thread that may have been waiting for it.

But I said that what actually happens is both tricky and gratifyingly elegant, and we haven't seen that part yet. How is a mutex actually implemented? Our threads are living in an environment where anything and everything that they are doing might be preempted at any time, without any warning. This literally happens in between instructions, between one instruction and the next instruction. And in between any two instructions, anything else might have transpired.

If you think about that for a moment, you'll realize that this means that acquiring ownership of an object cannot require the use of multiple instructions that might be interrupted by the operating system at any point. What we need is a single indivisible instruction that allows our threads to attempt to acquire ownership of a shared object, while also determining whether they succeeded. And that has to happen all at once.

It is so cool that an instruction as simple as "exchange" can provide this entire service. The exchange instruction does what its name suggests. It exchanges the contents of two "things." These might be two registers whose contents are exchanged, or it might be a location in memory whose contents are exchanged with a register. We already have a counter, the ownership of which is the entire issue. So we create another variable in

memory to manage that counter's ownership, to represent whether or not the counter is currently owned by any thread. We don't need to care which thread owns the counter because, if any thread does, no other thread can touch it. All they can do is wait for the object's ownership to be released.

So we just need some binary, you know, a one or a zero. If the value of this counter ownership variable is zero, then the counter is not owned by any thread at the moment. And if the counter ownership variable is not zero, if it's a one, then some thread currently owns the counter.

Okay. So how do we work with this and the exchange instruction? When a thread wants to take ownership of the counter, it places a one in a register and exchanges that register's value with the current value of the ownership variable in memory. If, after the exchange, its register, which received the previous value of the ownership variable, is a zero, then that means that until the exchange was made, the counter was not owned by any thread. And since this exchange simultaneously placed a one into the ownership variable at the same time, with the same instruction, our thread is now the exclusive owner of the counter and is free to do anything it wishes without fear of any collision.

If another thread comes along and wants to take ownership of the counter, it also places a one in a register and exchanges that with the counter's ownership variable. But since the counter is still owned by the first thread, this second thread will get a one back in its exchanged register. That tells it that the counter was already owned by some other thread, and that it's going to need to wait and try again later. And notice that since what it swapped into the ownership variable was also a one, this exchange instruction did not change the ownership of the counter. It was owned by someone else before, and it still is. What the exchange instruction did in this instance was to allow a thread to query the ownership of the counter. If it was not previously owned, it now would be by that query. But if it was previously owned by some other thread, it would still be.

And finally, once the thread that had first obtained exclusive ownership of the counter is finished with the counter and is ready to release its ownership, it simply stores a zero into the counter's ownership variable in memory, thus marking the counter as currently unowned. This means that the next thread to come along and swap a one into the ownership variable will get a zero back and know that it now owns the counter. We call the exchange instruction when it's used like this an "atomic" operation because, like an atom, it is indivisible. With that single instruction, we both acquire ownership if the object was free, while also obtaining the object's previous ownership status. Okay. I suppose I'm weird, but this is why I love coding so much.

The holistic way of viewing this is to appreciate that only atomic operations are safe in a preemptively multithreaded environment. In our example, if the counter in question only needed to be incremented by multiple threads, and that incrementation could be performed by a single indivisible instruction, then that would have been "thread safe," which is the term used. And there are instructions that increment memory in a single instruction. But in our example, the threads needed to examine the current value of the counter to make a decision about whether it had hit its maximum value yet, and only then increment it. That required multiple instructions and was not thread safe. So what do we do? We protect a collection of non-thread safe operations with a thread-safe operation. We create the abstraction of ownership and use an exchange instruction, which being one instruction is thread safe, to protect a collection of operations that are not thread safe.

And this brings us to "GhostRace," the discovery made by a team of researchers, and the question, what would it mean if it were discovered that these critical, supposedly atomic and indivisible synchronization operations were not, after all, guaranteed to do what we

thought? And Leo, let's tell our listeners of our last sponsor, and then we're going to answer that question because the bottom just fell out of computer security.

**Leo:** I guess you could have a race for the lock; right? That would be part of it? That's all you need to make them atomic operations; right? You can have races all the way down.

**Steve:** Well, you can't have a race for the lock or for the ownership because only actually one instruction is executing at any time.

**Leo:** Because it's atomic; right.

**Steve:** It feels like multiple things are going on. But actually, as we know, the processor is just switching among threads.

**Leo:** Right.

**Steve:** So at any given time there's only one thing happening. And if that one thing grabs the lock, if it grabs ownership, then it has it.

**Leo:** Right.

**Steve:** And that's the elegance of being able to do something in a single instruction.

**Leo:** Right. If it's multiple instructions, then you've got a problem.

**Steve:** Right.

**Leo:** If it was a resource that takes, you know, three cycles to poll, then you have the problem of you could have it interrupted.

**Steve:** Exactly.

**Leo:** It's really a fascinating ecosystem that's evolved. And this actually, this happened even before there were multithreaded computers; right? I mean, you could have a race condition anytime you have a branch, I guess.

**Steve:** Well, anytime you have timesharing - basically it's a matter of contention whenever there are multiple things contending for a single resource.

**Leo:** Right.

**Steve:** They're all free to read it at once; right? Everybody can read.

**Leo:** It's writing it that's the problem.

**Steve:** But if anybody modifies it, then you're in trouble.

**Leo:** Right.

**Steve:** Okay. So for their paper's Abstract, the researchers wrote: "Race conditions arise when multiple threads attempt to access a shared resource without proper synchronization, often leading to vulnerabilities such as concurrent use-after-free. To mitigate their occurrence, operating systems rely on synchronization primitives such as mutexes, spinlocks, et cetera.

"In this paper, we present GhostRace, the first security analysis of these primitives on speculatively executed code paths. Our key finding is that all of the common synchronization primitives can be microarchitecturally bypassed on speculative paths, turning all architecturally race-free critical regions into Speculative Race Conditions, or SRCs.

"To study the severity of SRCs, these Speculative Race Conditions," they wrote, "we focus on Speculative Concurrent Use-After-Free and uncover 1,283 potentially exploitable instances in the Linux kernel. Moreover, we demonstrate that" - they abbreviate it SCUAF, Speculative Concurrent Use-After-Free - "information disclosure attacks against the kernel are not only practical, but that their reliability can closely match that of traditional Spectre attacks, with our proof of concept leaking kernel memory at 12KB per second.

"Crucially, we develop a new technique to create an unbounded race window, accommodating an arbitrary number of Speculative Concurrent Use-After-Free invocations required by an end-to-end attack in a single race window. To address the new attack surface, we propose a generic SRC mitigation" - and we'll see about how expensive that is in a minute - "to harden all the affected synchronization primitives on Linux. Our mitigation requires minimal kernel changes, but incurs around a 5% geometric mean performance overhead on LMbench."

And then they finish their Abstract by quoting Linus Torvalds on the topic of Speculative Race Conditions, saying: "There's security, and then there's just being ridiculous." Okay. Well, they may not be so ridiculous after all. What this comes down to is a mistake that's been made in the implementation of current operating system synchronization primitives in an environment where processors are speculating about their own future. Somewhere in their implementations is a conditional branch that's responsible for making the synchronization decision. And in speculatively executing processors it's possible to deliberately mistrain the speculation's branch predictor to effectively neuter the synchronization primitive. As they say in their paper, to turn it into a no-op, which is the programmer shorthand for no operation. And if that can be done, it's possible to wreak havoc.

Here's what they explained. They said: "Since the discovery of Spectre, security researchers have been scrambling to locate all the exploitable snippets or gadgets in victim software. Particularly insidious is the first Spectre variant, exploiting conditional branch misprediction, since any victim code path guarded by a source 'if' statement may result in a gadget. To identify practical Spectre-v1 gadgets, previous research has

focused on speculative memory safety vulnerabilities, use-after-free, and type confusion. However, much less attention has been devoted to other classes of normally architectural software bugs, such as concurrency bugs.

"To avoid, or at least reduce concurrency bugs, modern operating systems allow threads to safely access shared memory by means of synchronization primitives, such as mutexes and spinlocks. In the absence of such primitives, for example, due to a software bug, critical regions would not be properly guarded to enforce mutual exclusion, and race conditions would arise. While much prior work has focused on characterizing and facilitating the architectural exploitation of race conditions, very little is known about their prevalence on transiently executed code paths. To shed light on the matter, in this paper we ask the following research questions: First, how do synchronization primitives behave during speculative execution? And second, what are the security implications for modern operating systems?

"To answer these questions, we analyze the implementation of common synchronization primitives in the Linux kernel. Our key finding is that all the common primitives lack explicit serialization and guard the critical region with a conditional branch. As a result, in an adversarial speculative execution environment, i.e., with a Spectre attacker mistraining the conditional branch, these primitives essentially behave like a no-op. The security implications are significant, as an attacker can speculatively execute all the critical regions in victim software with no synchronization.

"Building on this finding, we present GhostRace, the first systematic analysis of Speculative Race Conditions, a new class of speculative execution vulnerabilities affecting all common synchronization primitives. SRCs are pervasive, as an attacker can turn arbitrary architecturally race-free code into race conditions exploitable on a speculative path, in fact, one originating from the synchronization primitives' conditional branch itself. While the effects of SRCs are not visible at the architectural level, for example, no crashes or deadlocks, due to the transient nature of speculative execution, a Spectre attacker can still observe their microarchitectural effects via side channels. As a result, any SRC breaking security invariants can ultimately lead to Spectre gadgets disclosing victim data to the attacker."

Okay. So essentially the specter of Spectre strikes again. Another significant exploitation of speculative execution has arisen. And need we echo once again Bruce Schneier's famous observation about attacks never getting worse and only ever getting better?

Under the paper's mitigation section they write briefly: "To mitigate the Speculative Race Condition class of vulnerabilities, we implement and evaluate the simplest, most robust, and generic one, introducing a serializing instruction into every affected synchronization primitive before it grants access to the guarded critical region, thus terminating the speculative path. This provides a baseline to evaluate any future mitigations; and, as mentioned, mitigates not only the Speculative Condition Use-After-Free vulnerabilities presented in the paper, but all other potential Speculative Race Condition vulnerabilities."

Okay. So essentially what they're saying is that, as with all anti-Spectre mitigations, the solution is to deliberately shut down speculation at the spot where it's causing trouble. In this instance, this requires the insertion of an additional so-called "serializing" instruction into the code at the point just after the synchronization instruction and before the result of the synchronization instruction is used to inform a branch instruction. This prevents any speculative execution down either path following the branch instruction. The problem with doing that is that speculation exists specifically because it's such a tremendous performance booster. And that means that even just pausing speculation will hurt performance.

To determine how much this would hurt, for example, Linux's performance, these guys tweaked Linux's source code to do exactly that, to prevent the processor from speculatively executing down both paths following a synchronization primitive's conditional branch. What they found was that the effect can be significant. It introduces a 10.82% overhead when forking a Linux process, 7.53% overhead when executing a new process, 12.35% when deleting an empty 0K file, and 11.37% when deleting an 10K file. Those are particularly bad.

But overall, as they wrote in their abstract, they measured a 5.13% performance cost overall from their mitigation. Now, 5.13%, that's huge, and it suggests just how dependent upon synchronization primitives today's modern operating systems have become. For a small change in one very specific piece of code to have that much impact means it's being used all the time.

And finally, as for the disclosure of their discovery, they wrote: "We disclosed Speculative Race Conditions to the major hardware vendors - Intel, AMD, ARM, and IBM - and the Linux kernel in late 2023. Hardware vendors have further notified other affected software - OS and hypervisor - vendors, and all parties have acknowledged the reported issue, which has been given CVE-2024-2193.

"Specifically, AMD responded with an explicit impact statement which was that existing Spectre-v1 mitigations apply, pointing to the attacks relying on conditional branch mis-speculation like Spectre-v1." In other words, they're just saying, yeah, this is more of the same nightmare. If you want to fix it, deal with it the way you do conditional branch misprediction. Which is to say block speculation before the branch. The Linux kernel developers wrote back saying they have no immediate plans to implement serialization of synchronization primitives due to performance concerns. In other words, it slows down Linux too much for them to do this, and no one has yet shown them that they absolutely must. The bottom line is, it is unclear today what the ultimate solution will be. And it's clear we don't have one yet.

Stepping back from all of this to look at the whole picture, we have a processor performance-addicted industry that several years ago awoke to the unhappy news that the extremely complex, tricky, and sophisticated way our processors had been arranging to squeeze out every last possible modicum of performance was also inherently exploitable because it opened the processor to side-channel attacks.

It turned out that performance was being enhanced by allowing the processor to dynamically learn from the execution of its past code and data. Not only could this stored knowledge reveal secrets about the data that had recently been processed, but it was possible for any other software sharing the same hardware to probe for and obtain those secrets. And that's still where we are today. No one wants to relinquish the performance we've come to need, but the way we're achieving that performance inherently comes at the cost of security and privacy.

**Leo:** Wow. Very interesting. So now you understand, everybody, why the speculative execution processors are so problematic. It's got to be similar to what Apple's experiencing with the M1, M2, and M3 processors. Same idea.

**Steve:** Yeah. ARM processors have the same problem. I mean, we have a problem, which is our DRAM is too slow. So we've had to cache it. And processors need to minimize the hits on the cache. So they desperately need to know as much as they can about what's going to happen in the future. And that means the only way they can know what's going to happen in the future is if they know what happened in the past. And knowing what happens in the past means that the past affects their future behavior. And

that's where we get the performance, but that also means that their current behavior is affected by the past, which is why they leak information.

**Leo:** Yeah. Very interesting. Wow.

**Steve:** It is. It's a Catch-22. It's a classic conundrum. And it's just not clear. The only thing I could see, and this sort of relates to what we were talking about with the recent Apple problem with their M series chips, is that the cryptographic code needs to turn off that feature...

**Leo:** Right.

**Steve:** ...while it's running so in order...

**Leo:** Selective slowness.

**Steve:** Exactly. Exactly. Selective performance retardation.

**Leo:** But in a mission-critical environment, and where you really do want to preserve your integrity. And you know what, it doesn't make that much of a difference. I mean, yeah, 5%, 10%. But these things, as you pointed out...

**Steve:** And next year we'll have processors that are 20% faster.

**Leo:** Exactly. It's not - yeah, yeah. But I think, you know, the Apple fix is very straightforward. In fact, Apple's already implemented it.

**Steve:** Yes.

**Leo:** It's not as easy to do in Spectre and Meltdown, though, in the x86.

**Steve:** No. We don't, I mean, all of Intel's cores have it, so it's not a matter of, like, running on a core that doesn't have it. You can turn it off, you know. That's what my InSpectre freeware does is allow you to globally shut it down. But, you know. But, okay. And again, we need to maintain some perspective. If the only risk is hostile software in your system...

**Leo:** On your system, exactly.

**Steve:** Yes. It is sharing - it's a processor sharing problem where you've got, what is it, the evil maid. You have...

**Leo:** Or a server where multiple processes are running in the same...

**Steve:** Well, exactly. So an end-user working at home who's got a bunch of things that it's just them, it's nothing to worry about, really.

**Leo:** Yeah.

**Steve:** But it's in the virtualized environment in the cloud, and you have to, you know, we have to admit that more and more is in the cloud. You know, we're rushing headlong into the cloud. So there it does matter.

**Leo:** It affects you if your data is encrypted with keys that are held by the server, in the processor on the server, and somebody else is using the same processor because they're also on that shared cloud server. That is a problem. Apple, I don't know if it's luck, but Apple seems to have dodged a bullet by not having its efficiency cores use this kind of speculative execution.

**Steve:** Yeah.

**Leo:** That was really - worked out well because that could say, well, if you're encryption, use the efficiency cores.

**Steve:** Yeah. It wasn't actually luck. It's efficient because it's smaller. So they ripped out a bunch of things in order to make it small and more efficient.

**Leo:** Including that.

**Steve:** Including that. And that ended up being a good thing. They're just able to move the crypto over into that core.

**Leo:** I know, you know, you're concerned that some of our audience might go, this is way over my head. But I have to tell you everybody in the Discord was really engrossed, and many of them deal with this. I mean, this is - we had one person who says, "I work with this, I have to survive this all the time, every day." Vivi, or Vivi, I'm not sure how you say it, says, "I love this topic. I deal with this all day long with concurrent network programming." And that's exactly right. You have to have locks there, too, yeah.

**Steve:** Yeah.

**Leo:** It's really great. I love this stuff. And I know our audience does, too. And thank you, Steve.