THEME ARTICLE: TOP PICKS

Leaking Secrets Through Compressed Caches

Po-An Tsai, *NVIDIA, Santa Clara, CA, 95051, USA*

Andres Sanchez, *École Polytechnique Fédérale de Lausanne, 27218, Lausanne, Switzerland*

Christopher W. Fletcher, *University of Illinois at Urbana-Champaign, Champaign, IL,14589, USA*

Daniel Sanchez, *Massachusetts Institute of Technology, Cambridge, MA, 02139, USA*

We offer the first security analysis of cache compression, a promising architectural technique that is likely to appear in future mainstream processors. We find that cache compression has novel security implications because the compressibility of a cache line reveals information about its contents. Compressed caches introduce a new side channel that is especially insidious, as simply storing data transmits information about the data. We present two techniques that make attacks on compressed caches practical. Pack+Probe allows an attacker to learn the compressibility of victim cache lines, and Safecracker leaks secret data efficiently by strategically changing the values of nearby data. Our evaluation on a proof-of-concept application shows that, on a representative compressed cache architecture, Safecracker lets an attacker compromise an 8-byte secret key in under 10 ms. Even worse, Safecracker can be combined with latent memory safety vulnerabilities to leak a large fraction of program memory.

Over the past three years, computer architecture has suffered a major security crisis. Researchers have uncovered critical security flaws in billions of deployed processors related to speculative execution, starting with Spectre[1] and Meltdown,[2] generating significant interest and progress in microarchitectural side and covert channel research.

While microarchitectural side channel attacks have been around for over a decade, speculative execution attacks are significantly more dangerous because of their ability *to leak program data directly*. In the worst case, these attacks let the attacker construct a *universal read gadget*,[3] capable of leaking data at attacker-specified addresses. For example, the Spectre V1 attack—`if (i < N) { B[A[i]]; }`—exploits branch misprediction to leak the data at address `&A + i` given an attacker-controlled `i`.

Yet, speculative execution is only one optimization in modern microprocessors. It is critical to ask: Are there other microarchitectural optimizations that enable a similarly large amount of data leakage? In this work, we provide an answer in the affirmative by analyzing the security of *memory hierarchy compression*, specifically *cache compression*.

Compression is an attractive technique to improve memory performance and has received intense development from both academia and industry. Some early adopter processors already feature memory-hierarchy compression, including IBM's z15,[4] Qualcomm's Centriq, and NVIDIA's A100. As data movement becomes increasingly critical, we expect to see general-purpose cache compression become widely used. Nonetheless, despite strong interest from both academia and industry, prior research in this area has focused on performance and *ignored security*.

We present the first security analysis of cache compression. The key insight that our analysis builds on is that the compressibility of data reveals information about the data itself. Similar to speculative execution attacks, we show how this allows an attacker to *leak program data directly* and, in the worst case, create a new universal read gadget that can leak large portions of program memory. In short, we show that cache compression—without speculative execution—can leak as much program privacy as speculative execution.

0272-1732 © 2021 IEEE
Digital Object Identifier 10.1109/MM.2021.3069158
Date of publication 26 March 2021; date of current version 25 May 2021.

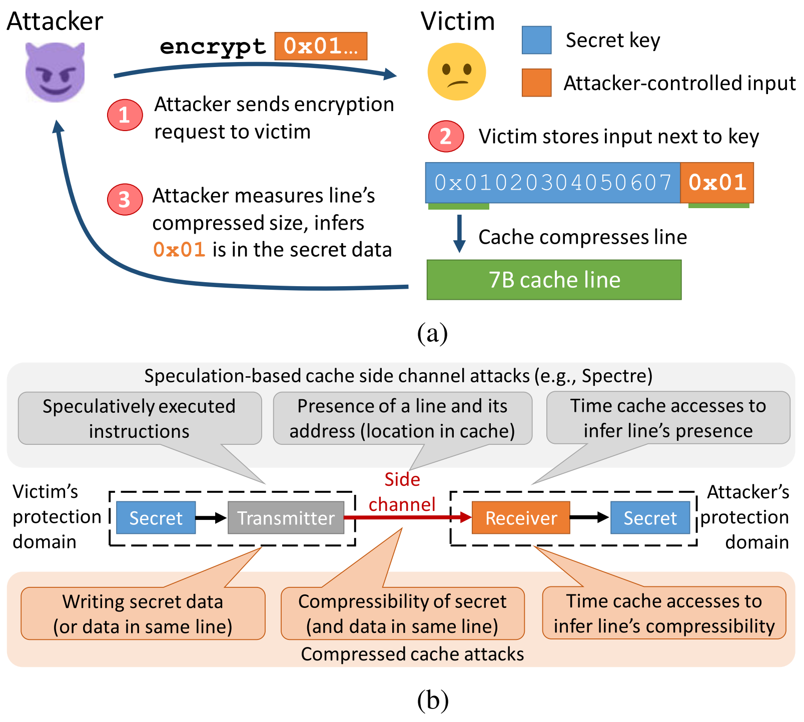

Attacker
encrypt 0x01...
Victim
🟦 Secret key
🟧 Attacker-controlled input
1 Attacker sends encryption request to victim
2 Victim stores input next to key
0x01020304050607 0x01
Cache compresses line
7B cache line
3 Attacker measures line's compressed size, infers 0x01 is in the secret data
(a)
Speculation-based cache side channel attacks (e.g., Spectre)
Speculatively executed instructions
Presence of a line and its address (location in cache)
Time cache accesses to infer line's presence
Victim's protection domain
Secret
Transmitter
Side channel
Receiver
Secret
Attacker's protection domain
Writing secret data (or data in same line)
Compressibility of secret (and data in same line)
Time cache accesses to infer line's compressibility
Compressed cache attacks
(b)

FIGURE 1. Overview of our compressed cache attacks. (a) Simple attack on a compressed cache, where the attacker exploits colocation with secret data to leak it. (b) Comparison between Spectre and our proposed attacks.

CACHE COMPRESSION INTRODUCES A NEW CHANNEL

Figure 1(a) shows a simple attack on compressed caches. The attacker seeks to steal the victim's encryption key and can submit encryption requests to the victim. On each request, the victim's encryption function stores the key and the attacker's plaintext consecutively, so they fall on the same cache line.

THE KEY INSIGHT THAT OUR ANALYSIS BUILDS ON IS THAT THE COMPRESSIBILITY OF DATA REVEALS INFORMATION ABOUT THE DATA ITSELF.

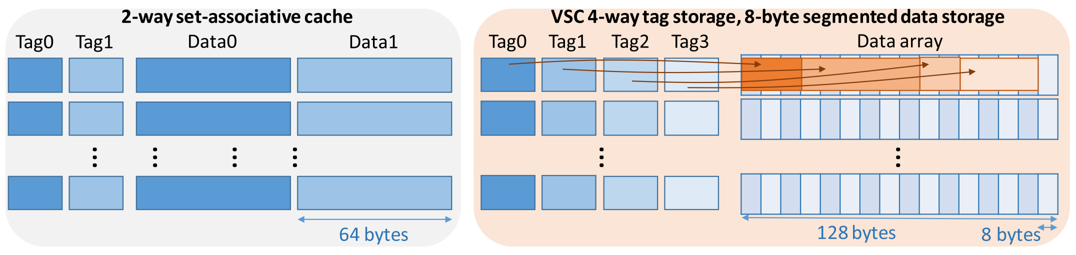
Colocating secret data and attacker-controlled data is safe with conventional caches, but it is unsafe with a compressed cache. Suppose we run this program on a system with a compressed cache that tries to shrink each cache line by removing duplicate bytes. If the attacker can observe the line's size, it can leak all individual bytes of the key by trying different chosen plaintexts, as the compressed line's size changes when a byte of the key matches a byte of the plaintext.

The general principle in the above mentioned example is that when the attacker is able to *colocate* its own data alongside secret data, it can learn the secret data. Beyond cases where the victim itself facilitates colocation (e.g., by pushing arguments onto the stack), we observe that latent security vulnerabilities related to memory safety, such as buffer overflows, heap spraying, and uninitialized memory, further enable the attacker to colocate its data with secret data. Combined with the new side channel in compressed caches, these can enable a read gadget that leaks a significant amount of data, as we will show later.

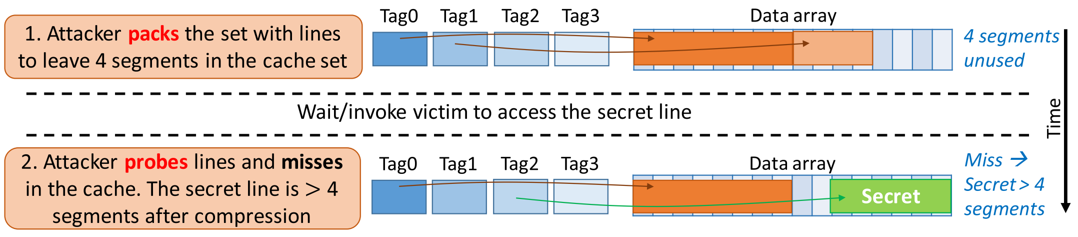
CACHE COMPRESSION ENABLES A NEW TYPE OF ACTIVE ATTACK

All compression techniques seek to store data efficiently by using a *variable-length code*, where the length of the encoded message approaches the *information content*, or *entropy*, of the data being encoded. It trivially follows that the *compressibility* of a data chunk, i.e., the compression ratio achieved, reveals information about the data.

Hence, compressed caches introduce a new, fundamentally different type of side channel. As a point


2-way set-associative cache
Tag0
Tag1
Data0
Data1
64 bytes
VSC 4-way tag storage, 8-byte segmented data storage
Tag0
Tag1
Tag2
Tag3
Data array
128 bytes
8 bytes

(a)


1. Attacker packs the set with lines to leave 4 segments in the cache set
Tag0
Tag1
Tag2
Tag3
Data array
4 segments unused
Wait/invoke victim to access the secret line
Time
2. Attacker probes lines and misses in the cache. The secret line is > 4 segments after compression
Tag0
Tag1
Tag2
Tag3
Data array
Secret
Miss →
Secret > 4 segments

(b)

FIGURE 2. Comparison of VSC and conventional caches and example showing how Pack+Probe measures compressibility on VSC. (a) Comparison of VSC (right) versus an uncompressed set-associative cache (left). VSC divides each set of the data array into small *segments* (8 bytes in this example), stores each variable-sized line as a contiguous run of segments in the data array, modifies tags to point to the blocks' data segments, and increases the number of tags per set relative to the uncompressed cache (by $2\times$ in this example) to allow tracking more, smaller lines per set. (b) Simplified example of the Pack+Probe attack on VSC. The attacker first packs the target set to leave exactly S segments left. After the victim accesses the secret, the attacker probes the set to see if the compressed size of the secret is larger than S segments.

of comparison, consider conventional cache-based side channels. Conventional cache channels are based on the *presence or absence* of a line in the cache. Thus, conventional attacks make a strong assumption, namely that the victim is written in a way that encodes the secret as a load address.

Compressed cache attacks relax this assumption. Compressed cache channels are based on *data compressibility* in the cache. Data can leak regardless of how the program is written, just based on what data is written to memory. In this sense, our attacks on compressed caches are more similar to Spectre and the recent Rambleed[5] attacks than conventional side-channel attacks.

Figure 1(b) compares Spectre and the new attacks in this article, using an abstract view of a side-channel attack from prior work.[6] Attacker and victim reside in different protection domains, so the attacker resorts to a *side channel* to extract the secret from the victim. To exploit the side channel, a *transmitter* in the victim's protection domain encodes the secret into the *channel*, which is read and interpreted by a *receiver* in the attacker's protection domain.

Spectre attacks allow the attacker to create different transmitters by arranging various sequences of mispredicted speculations. Analogously, attacks based on cache compression allow the attacker to create different transmitters by writing different data into the cache. Exploiting the compressed cache side channel requires a new *receiver*. In Spectre, the receiver uses techniques like Prime+Probe to detect the timing difference due to a line's presence. In compressed cache attacks, the receiver has to detect the compressibility information from the channel. To this end, we present Pack+Probe, a general technique that leverages the timing difference due to a line's presence to also infer its compressibility.

PACK+PROBE: MEASURING COMPRESSIBILITY

Whereas conventional caches manage fixed-size cache lines, compressed caches manage variable-sized lines. Thus, compressed caches divide the data array among variable-sized blocks and track their tags in a way that 1) enables fast lookups and insertions; 2) allows high compression ratios; and 3) avoids high tag storage overheads. While prior work has proposed various compressed cache architectures, Pack+Probe is general and applies broadly. For concreteness, we explain it using a commonly used organization, Variable-Sized Cache (VSC).[7]

VSC extends a set-associative design to store compressed, variable-sized cache lines. Figure 2(a)

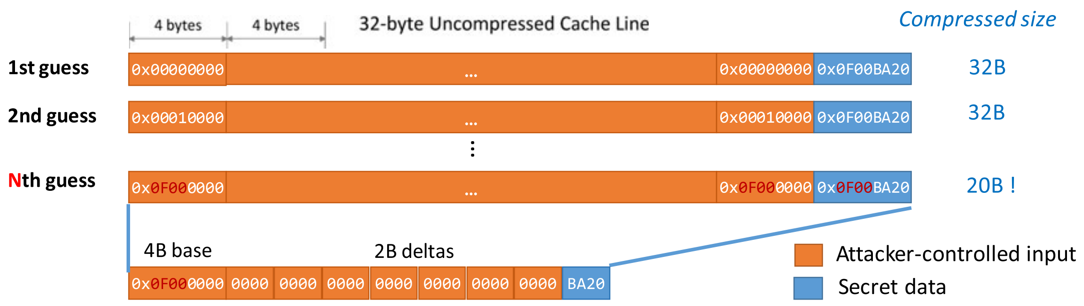


FIGURE 3. Example of the Safecracker attack on BDI. The attacker fills the cache line with attacker-controlled data and brute-forces the first 2 bytes in every 4 bytes until the guess is close to the secret value and incurs compression. The change in the line size can be observed using Pack+Probe.

illustrates VSC and compares it with a set-associative design. VSC divides each set of the data array into small *segments* [8 bytes in Figure 2(a)]. It stores each variable-size line as a contiguous run of segments in the data array. Each tag includes a pointer to identify the block's data segments within the set, and VSC increases the number of tags per set relative to the uncompressed cache [e.g., by 2× in Figure 2(a)] to support smaller lines.

Pack+Probe exploits that, in compressed caches, a victim's access to a cache line may cause different evictions of other lines depending on the compressibility (i.e., compressed size) of the victim's line, in addition to the replacement policy. Since the compressed cache can store variable-sized lines, whether a newly installed line evicts other lines is determined by the unused capacity in the compressed cache.

Figure 2(b) shows a single step of Pack+Probe. Pack+Probe first *packs* the compressed cache with attacker-controlled lines to leave exactly *S segments unused*. Once the victim accesses the secret line, if its compressed size is ≤S segments, no evictions will happen, whereas if it is larger than S segments, at least one of the attacker-controlled lines will be evicted. Finally, the attacker *probes* (accesses) the lines it inserted again, uses timing differences to infer which lines hit or miss, and thus infers whether the victim's line fits within X bytes. Repeating these steps with a simple binary search over values of X, the attacker can precisely determine the compressed size of the victim's line.

Our full paper[8] shows Pack+Probe on VSC takes less than 10 K cycles to determine the compressed size of the victim's line. Pack+Probe's efficiency allows us to further develop Safecracker, an active attack to leak secret data.

SAFECRACKER: LEARNING SECRETS THROUGH COLOCATION

Safecracker is named after the process used to crack combination locks in (classic) safes, where the attacker cycles through each digit of the combination and listens for changes in the lock that signal when the digit is correct. Similarly, Safecracker provides a guess and learns indirect outcomes (compressibility) of the guess. The attacker then uses the outcome to guide the next guess.

In the context of compressed caches, the attacker first makes a guess about the secret data and then builds a data pattern that, when colocated with the secret data, will cause the cache line to be compressed in a particular way if the guess is correct [like in Figure 1(a)]. By measuring the line's compressibility using Pack+Probe, the attacker knows whether the guess was correct.

Depending on the compression algorithm used, Safecracker needs different search strategies. We showcase and implement the Safecracker attack on the base-delta-immediate (BDI) compression algorithm,[9] a simple and common baseline algorithm that relies on delta encoding. The BDI algorithm performs intra-cache-line compression by storing a common base value and small deltas. For example, for eight 4-byte integer values ranging from 1,280 to 1,287, BDI will store a 4-byte base of 1,280, and eight 1-byte values from 0 to 7, compressing a 32-byte cache line into a 12-byte compressed line. Depending on the base value and the ranges of the deltas, BDI compresses a cache line into eight different sizes. For example, a 4-byte base and 16 2-byte values result in a 20-byte compressed line.

Figure 3 shows how our Safecracker attack exploits BDI. Assume the attacker wishes to steal a 4-byte secret, located in a cache line where all other data, 28 bytes in this example, is attacker controlled [e.g., it is part of a request buffer like in Figure 1(a)]. Safecracker starts by targeting a compressed pattern of a 4-byte base and 2-bytes values (a 20-byte line). It brute-forces the first 2 bytes of the victim's word by changing the first 2 bytes of attacker-controlled words (i.e., trying patterns `0x00000000,0x00010000,{...}0xFFFF0000`, at most 2^{16} guesses), and uses Pack+Probe to see if the cache line compresses to the size of a 20-byte line. Since the secret in this

TABLE 1. Worst-case time for Safecracker to leak secrets of different sizes.

	Time (ms) to leak secret of size							
Attack	**1 bytes**	**2 bytes**	**3 bytes**	**4 bytes**	**5 bytes**	**6 bytes**	**7 bytes**	**8 bytes**
Safecracker	0.11	0.30	1.05	54.0	54.4	256	-	-
Safecracker+ buffer overflow	0.76	1.53	2.30	3.11	3.91	4.32	5.61	6.46

example is `0x0F00BA20`, when the attacker guesses `0x0F000000`, the line compresses into a 20-byte line, and the attacker records the 2 bytes it tried as part of the secret. The attacker targets the next compressed pattern, a 4-byte base and 1-byte values, and brute-forces the third byte (taking at most 2^8 guesses), and then brute-forces the last byte to learn all 4 bytes of the secret.

Our ASPLOS'20 paper[8] details the algorithm to steal secrets in various sizes on BDI, and we also discuss how to extend the idea to other cache compression algorithms. Safecracker on BDI can leak up to 8 bytes of secret data, which can be devastating. For example, a 128-bit AES key is considered secure, but leaking 64 bits degrades it to 64-bit protection, which is insecure in many contexts.

Enhancing Safecracker With Latent Memory-Safety Violations

The previous example assumes that the attacker-controlled data (OATD) is located right next to the secret in the same cache line. This limits the attacker to only leak contents contiguous to attacker-controlled data. However, if the attacker can find other latent memory-safety violations in the victim program, then this vulnerability can significantly increase the amount of data leaked and enhance the efficiency of Safecracker.

For example, combining Safecracker with buffer overflows, the attacker can control *where* the attacker-controlled OATD is located. In the worst case, if the compression algorithm allows leaking up to X bytes per line, memory-safety violations let the attacker leak X bytes from *every cache line*. That is, if the victim has a memory footprint of M, then Safecracker with a buffer overflow can leak $O(M)$ bytes of memory, where different compression algorithms have different constant factors (e.g., $8/64 = 1/8$th of program memory for BDI).

Moreover, buffer overflows also allow the attacker to control *how many bytes* the attacker-controlled data consists of. Leveraging this, the attacker can leak the secret much more efficiently by making all partial guesses at byte granularity. The attacker can allocate a buffer that leaves only one byte of secret data in the line. By brute-force, the attacker quickly learns this remaining byte in the same manner as the previous example. Once the last byte is known, the attacker learns the second-to-last byte by allocating a smaller buffer that does not overwrite it and brute-forcing only this byte. This requires the victim to restore the data over multiple invocations, which is the case with local, stack-allocated variables. By repeating these steps, the attacker can learn the 8 bytes of secret data even faster.

SAFECRACKER LEAKS SECRETS EFFICIENTLY

We evaluate the effectiveness of Safecracker using proof-of-concept workloads (PoC) and architectural simulation (simulating a compressed cache with VSC+BDI). The first PoC has two separate processes, victim and attacker. The victim is a login server with a vulnerability that lets attacker-controlled input be stored next to a secret key. The attacker can provide input to the cache line where the key is located, without modifying the key. The attacker can also invoke victim accesses to the secret by issuing encryption requests that use the secret key. This lets the attacker perform Pack+Probe.

The attacker first finds the set that holds the secret cache line using standard Prime+Probe.[10] Once the conflicting set is found, the attacker uses Safecracker to steal the victim's secret key.

Table 1 reports the worst-case execution time needed to steal different numbers of bytes. Safecracker requires less than a second to crack a 6-byte secret value. Though Safecracker can steal up to 8 bytes when applied to BDI, trying to steal more than 6 bytes requires much longer runtimes (hours for 8 bytes), because the complexity of Safecracker on BDI grows exponentially. Nonetheless, as we explained earlier, Safecracker can be combined with latent memory violations to enhance its efficiency.

Buffer-Overflow-Based Attack

The second PoC builds on top of the first PoC. However, this time, the victim has a buffer-overflow vulnerability. The vulnerable function is as follows:

```
void encrypt(char *plaintext) {
 char result[LINESIZE];
 char data[DATASIZE]; // can be any size
 char key[KEYSIZE];
 memcpy(key, KEYADDR, KEYSIZE);
 strcpy(result, plaintext);
{...}
}.
```

The buffer overflow stems from the unsafe call to `strcpy`, which causes out-of-bounds writes when the `plaintext` input string exceeds `LINESIZE` bytes. The attacker exploits this buffer overflow to scribble over the stack and overwrite some of the bytes of the key. After `encrypt` returns, the scribbled-over line remains in the stack, and the attacker is then able to measure its compressibility with Pack+Probe and to run Safecracker as described before.

Buffer overflows give Safecracker much higher bandwidth by allowing it to guess a single byte on each step. Using a buffer overflow, Safecracker steals 8 bytes of secret data in under 10 ms. Table 1 describes that attack time with a buffer overflow grows linearly versus exponentially as it had with the first PoC.

While Safecracker applied to BDI can steal only 8 bytes per line, the above mentioned process can be repeated for different-sized attacker-controlled buffers to steal data in other lines, e.g., 8 bytes from multiple (potentially all) lines.

GENERALIZATION AND DEFENSES

Our ASPLOS'20 paper[8] also discusses how to generalize these attacks to other compressed cache architectures and algorithms, and presents other opportunities to colocate attacker-controlled data with secret data. We find that 1) Pack+Probe is applicable to most compressed cache architectures with a decoupled tag store with extra tags, and a data array divided in fixed-size sets where variable-sized blocks are laid over; 2) variants of Safecracker can be constructed case by case for other compression algorithms, and the better the algorithm compresses, the more information it can leak; and 3) there are many more ways to colocate attacker-controlled data near sensitive data, both *spatially* (e.g., heap spraying) and *temporally* (e.g., uninitialized data), and widely used software, such as the Linux kernel, suffers from these attack vectors.

Finally, we present multiple ways (e.g., obfuscation) to defend against Pack+Probe, Safecracker, and other attacks on compressed caches. We evaluate one of them, partitioning the compressed cache, to understand the tradeoff between security and performance. Our analysis shows that even though it is possible to make compressed caches secure, the straightforward solution that partitions both the tag and data array comes at a cost. How to limit this performance impact while minimizing leakage would be interesting future work.

LESSONS LEARNED

We have presented the first security analysis of cache compression and found that cache compression is insecure because *the compressibility of a cache line* reveals information about its contents.

With our proposed attacks, Pack+Probe and Safecracker, we have also shown how cache compression can potentially leak as much program privacy as speculative execution. This has significant implications for architects. It suggests we as a community need to revisit our vast literature and reexamine other microarchitectural optimizations through a security lens.

Beyond the above mentioned immediate and long-term message, our paper makes two other microarchitectural side channel "firsts."

The first work to show that memory-safety violations can enhance microarchitectural attacks: Memory safety vulnerabilities, e.g., buffer overflows, and microarchictural attacks are both prominent vulnerability classes in their own right. Fortunately, however, we have traditionally been able to treat them as orthogonal concerns with orthogonal sets of attacks and defenses.

In this work, we show that memory-safety violations can enhance microarchitectural attacks. In other words, memory safety vulnerabilities can be used to mount (and exacerbate) microarchitectural attacks. Worse, defenses against a given memory-safety vulnerability are not sufficient to block its "mirror image" in the microarchitectural-attack world. Our original paper illustrates this idea by showing how a "code-reuse buffer overflow" defense (StackGuard) is insufficient for preventing a "microarchitectural-attack buffer overflow" which, when combined with compressed caches, can leak (asymptotically) all of the program memory.

WE HOPE OUR WORK CATALYZES A LINE OF WORK IN PROACTIVE SECURITY ANALYSIS OF MICROARCHITECTURAL OPTIMIZATIONS.

The first data-centric, differential microarchitectural attack: There is a rich heritage in the security community for performing chosen-plaintext attacks. In this model, the victim program P takes as input sensitive data S and attacker-controlled data C, and produces an observation O, i.e., $O = \text{View}(P|S, C)$. The attacker more-precisely learns S by varying C and monitoring changes in O. This style is also called a *differential* attack; it is used to perform cryptanalysis and amplify traditional side-channel, e.g., DPA, attacks.

This article presents the first data-centric differential attack in the microarchitectural attack setting. Specifically, C is attacker-controlled data that colocates with S. We show how by modulating C, the attacker can perform a guided search to recover S by observing changes in compressibility O. Making matters worse, depending on the cache compression algorithm, this search can be performed in asymptotically fewer steps than brute-force guessing *S*.

We hope this work prevents insecure cache compression techniques from reaching mainstream processors. More importantly, we hope our work catalyzes a line of work in proactive security analysis of microarchitectural optimizations.

ACKNOWLEDGMENTS

We would like to thank Maleen Abeydeera, Joel Emer, Mark Jeffrey, Anurag Mukkara, Quan Nguyen, Victor Ying, Guowei Zhang, and the ASPLOS'20 reviewers for their feedback. We would also like to thank Joel Emer for his insights on the taxonomy, and Paul Kocher for sharing his concerns about the security of memory compression. This work was supported in part by the National Science Foundation under Grant CAREER-1452994 and Grant SaTC-1816226, in part by the Google faculty research award, and in part by an Intel ISRA grant. The work of Andres Sanchez was supported through a MISTI grant by the Technical University of Madrid.

REFERENCES

1. P. Kocher *et al.*, "Spectre attacks: Exploiting speculative execution," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 1–19, doi: 10.1109/SP.2019.00002.
2. M. Lipp *et al.*, "Meltdown: Reading kernel memory from user space," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 973–990, doi: 10.1145/3357033.
3. R. McIlroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *CoRR*, vol. abs/1902.05178, 2019. [Online]. Available: http://arxiv.org/abs/1902.05178
4. B. Abali *et al.*, "Data compression accelerator on IBM power9 and z15 processors: Industrial product," in *Proc. 47th Annu. Int. Symp. Comput. Archit.*, 2020, pp. 1–14, doi: 10.1109/ISCA45697.2020.00012.
5. A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "Rambleed: Reading bits in memory without accessing them," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 695–711, doi: 10.1109/SP40000.2020.00020.
6. V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A defense against cache timing attacks in speculative execution processors," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchit.*, 2018, pp. 974–987, doi: 10.1109/MICRO.2018.00083.
7. A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *Proc. 31st Annu. Int. Symp. Comput. Archit.*, 2004, pp. 212–223, doi: 10.1109/ISCA.2004.1310776.
8. P.-A. Tsai, A. Sanchez, C. W. Fletcher, and D. Sanchez, "Safecracker: Leaking secrets through compressed caches," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2020, pp. 1125–1140, doi: 10.1145/3373376.3378453.
9. G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *Proc. 21st Int. Conf. Parallel Archit. Compilation Techn.*, 2012, pp. 377–388, doi: 10.1145/2370816.2370870.
10. D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Proc. 7th Cryptographers' Track RSA Conf. Topics Cryptology*, 2006, pp. 1–20, doi: 10.1007/11605805_1.

PO-AN TSAI is currently a Research Scientist with NVIDIA, Santa Clara, CA, USA. His current research focuses on tensor accelerators design/modeling and memory hierarchy optimizations for domain-specific accelerators. Tsai received a B.S. degree in electrical engineering from National Taiwan University, Taipei, Taiwan, in 2012, and S.M. and Ph.D. degrees in computer science from MIT, Cambridge, MA, USA, in 2015 and 2019, respectively. He is a Member of IEEE. Contact him at poant@nvidia.com.

ANDRES SANCHEZ is currently working toward a graduate degree with the École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland. His research topic is side-channels detection and mitigation, security-aware systems layering, compilation techniques, and characterization of sensitive information. Sanchez received a B.S. degree in mathematics and computer engineering from the Technical University of Madrid, Madrid, Spain, in 2019. Contact him at andres.sanchez@epfl.ch.

CHRISTOPHER W. FLETCHER is currently an Assistant Professor in computer science with the University of Illinois at Urbana-Champaign, Champaign, IL, USA. He has interests ranging from computer architecture to security to high-performance computing (ranging from theory to practice, algorithm to software to hardware). Fletcher received a Ph.D. degree from the Massachusetts Institute of Technology, Cambridge, MA, USA, in 2016. Contact him at cwfletch@illinois.edu.

DANIEL SANCHEZ is currently an Associate Professor with the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA. His research interests include scalable memory hierarchies, architectural support for parallelization, and accelerators for sparse computations. Sanchez received a Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, USA. Contact him at sanchez@csail.mit.edu.