Safecracker: Leaking Secrets through Compressed Caches

Po-An Tsai
Massachusetts Institute of Technology
poantsai@csail.mit.edu

Andres Sanchez
Massachusetts Institute of Technology
andressm@csail.mit.edu

Christopher W. Fletcher
University of Illinois at Urbana-Champaign
cwfletch@illinois.edu

Daniel Sanchez
Massachusetts Institute of Technology
sanchez@csail.mit.edu

Abstract

The hardware security crisis brought on by recent speculative execution attacks has shown that it is crucial to adopt a security-conscious approach to architecture research, analyzing the security of promising architectural techniques *before* they are deployed in hardware.

This paper offers the first security analysis of cache compression, one such promising technique that is likely to appear in future processors. We find that cache compression is insecure 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 it.

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 common compressed cache architecture, Safecracker lets an attacker compromise a secret key in under 10 ms, and worse, leak large fractions of program memory when used in conjunction with latent memory safety vulnerabilities. We also discuss potential ways to close this new compression-induced side channel. We hope this work prevents insecure cache compression techniques from reaching mainstream processors.

Po-An Tsai is now at NVIDIA Research (`poant@nvidia.com`).
Andres Sanchez is now at EPFL (`andres.sanchez@epfl.ch`).

ASPLOS'20, March 16–20, 2020, Lausanne, Switzerland
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7102-5/20/03…$15.00
10.1145/3373376.3378453

CCS Concepts • **Computer systems organization → Processors and memory architectures**; • **Information systems → Data compression**; • **Security and privacy → Side-channel analysis and countermeasures**.

Keywords cache; compression; side channel; security.

ACM Reference Format:
Po-An Tsai, Andres Sanchez, Christopher W. Fletcher, and Daniel Sanchez. 2020. Safecracker: Leaking Secrets through Compressed Caches. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 16 pages. 10.1145/3373376.3378453

1 Introduction

Over the past two 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 [44], Meltdown [46], and quickly expanding into a rich new sub-area of microarchitectural side channel research [22, 32, 36, 43, 86, 87].

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* [50], 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 performance feature of modern microprocessors. It is critical to ask: *are there other microarchitectural optimizations that enable a similarly large amount of data leakage?*

In this paper, 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 [3, 4, 8, 9, 37, 40, 55, 59, 60, 62, 69, 70, 81, 90, 92] and industry [17, 18, 31, 41, 45, 61]. Several deployed systems already use memory-hierarchy compression. For example, IBM POWER systems use hardware main-memory compression (MXT [78]), and both NVIDIA and AMD GPUs use hardware compression for image data (Delta Color Compression [6]). As data movement becomes increasingly critical, the natural next step is to adopt general-purpose cache compression. Nonetheless, despite strong interests from both academia and industry, prior research in this area has focused on performance and *ignored security*.

In this paper we offer 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.

A simple example: Fig. 1 shows the setup for 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 plaintext consecutively, so they fall on the same cache line.

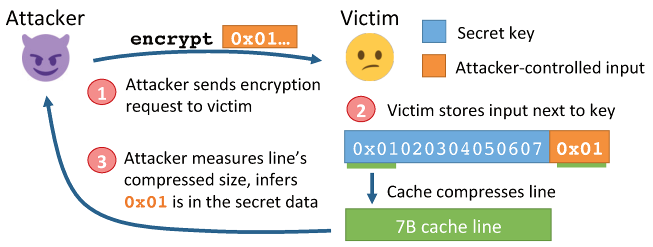

Attacker
encrypt 0x01...
Victim
Secret key
Attacker-controlled input
1 Attacker sends encryption request to victim
2 Victim stores input next to key
0x01020304050607 0x01
3 Attacker measures line's compressed size, infers 0x01 is in the secret data
Cache compresses line
7B cache line

Figure 1. A simple attack on a compressed cache, where the attacker exploits colocation with secret data to leak it.

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 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. These can enable a read gadget that leaks a significant amount of data (e.g., 8 bytes of every 64 bytes cache line in our evaluation).

We demonstrate the extent of this vulnerability by developing damaging attacks for a commonly used compressed cache architecture, VSC [3], using a common compression algorithm, BDI [62]. We first develop Pack+Probe (Sec. 3), a variant of Prime+Probe that enables an attacker to observe the compressed size of cache lines from the victim. We then design Safecracker (Sec. 4), an active attack on the BDI compression algorithm that leverages data colocation to recover the data from the compressed size. We evaluate these techniques in simulation (Sec. 5), and show that an attacker can obtain the victim's secret key in under 10 ms. Finally, we demonstrate how a latent buffer overflow in the target program allows the attacker to learn $O(M)$ Bytes of program memory, for an M-Byte memory (where BDI in particular leaks $1/8^*M$ Bytes). We then present several mitigation strategies (Sec. 6), including selective compression and cache partitioning, and evaluate a secure compressed cache, which leverages DAWG-style [43] partitioning on its tag and data arrays. Finally, we discuss how to generalize our attacks to other cache architectures and algorithms (Sec. 7).

In summary, this paper contributes:

- The first security analysis of compressed caches.
- The observation that cache line compressibility reveals information about the cache line's data, representing a new side channel.
- The observation that colocation of attacker-controlled data with secret data enables efficient extraction of secret data through this new side channel.
- Pack+Probe, an efficient protocol to learn cache line compressibility in VSC-style caches.
- Safecracker, an efficient attack that leverages data colocation to extract secret data from caches using BDI.
- Safecracker, augmented with a buffer overflow, to leak $O(M)$ Bytes for an M-Byte memory.
- A description of mitigation strategies and an evaluation of a secure, partitioned compressed cache.
- Discussions of how our attacks generalize to other compressed cache architectures and algorithms.

2 The cache compression side channel

In this section we introduce the security pitfalls of cache compression and summarize our contributions in a generic fashion, without focusing on specific compressed caches or compression algorithms. We first describe prior cache-based side channels and attacks using a general taxonomy. Then, we use this taxonomy to show how compression-based attacks differ from prior work and why prior defenses are insufficient. Next, we present generic passive and active attacks on compressed caches, which we term Pack+Probe and Safecracker. Finally, we sketch general mitigation strategies.

2.1 Cache-based side channels, attacks, and defenses

Fig. 2 shows an abstract view of a side-channel attack, which prior work introduced and used [43] to understand and classify attack types. An *attacker* process seeks to learn some *secret* data from a *victim* process. Attacker and victim reside in different protection domains, so the attacker resorts to a *side channel* (i.e., a means to convey information through an

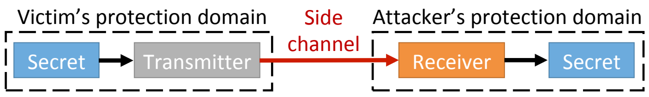


Figure 2. An abstract view of a side-channel attack [43].

implementation-specific detail, such as microarchitectural state) 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. This distinction between transmitter, channel, and receiver helps differentiate attacks and defenses. Defenses need to thwart at least one of these ingredients to prevent the attack.[1]

Prior **cache-based side channels** encode information through the presence or absence of a cache line. Information is conveyed only by the line's presence or absence, and its *location* in the cache, which reveals some bits of its *address*.

Receivers for this side channel rely on measuring timing differences between hits and misses to infer the line's presence. Prime+Probe [57] is a commonly used receiver. In Prime+Probe, the attacker first fills, i.e., *primes*, a cache set with its own lines. Then, it detects whether the victim has accessed a line mapped to the same set by re-accessing, i.e., *probing*, the same lines and checking if there are cache misses.

In the first reported cache-based side channel attacks, which leaked keys in AES [11] and RSA [89], the victim itself was the transmitter, leaking data through its access pattern. For example, these algorithms used lookup tables indexed by bits of the secret key, and the attacker inferred those bits by observing which cache sets the victim used.

To defend against these attacks, modern crypto implementations use *constant-time algorithms*, also known as *data-oblivious algorithms* [2, 7, 12, 13, 19, 25, 28, 52, 54, 56, 66, 71, 74, 77, 93]. Constant-time algorithms are carefully written so that *each instruction's execution* does not reveal the data it operates on over any microarchitectural side channel. For example, these algorithms do not leak bits of the key through their access pattern, making the original cache-based attacks ineffective. Unfortunately, the guidelines for writing constant-time code do not consider side channels based on *data at rest optimizations*, like cache compression, and are thus rendered insecure by those techniques, as we will later see.

Spectre [44] and other recent **speculation-based attacks** [16, 46, 72] all leverage speculative execution to *synthesize a transmitter*. Specifically, the attacker modifies microarchitectural state (e.g., the branch predictor) to get the processor to speculatively execute code on the victim's domain that the victim would not normally execute. The transmitter code encodes the secret into a side channel, e.g., the cache-based side channel using address bits to encode the secret.

[1] Though we focus on side-channel attacks, covert channels have the same ingredients. The difference is that in a covert channel setup, both processes want to communicate through the side channel.

Now that we have understood where existing attacks fit in this taxonomy, it is easy to see why the attacks we report are different and more insidious.

2.2 Contribution 1: Cache compression introduces a new channel

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 [73]. It trivially follows that the *compressibility* of a data chunk, i.e., the compression ratio achieved, reveals information about the data.

Different compression algorithms reveal different amounts of information. In general, more sophisticated algorithms compress further and thus tend to reveal more information [20]. For instance, the example in Sec. 1 only revealed how many bytes were the same. But a different technique, *delta encoding* [62], encodes each byte or word as the difference with a base value and uses fewer bits for smaller deltas, thus revealing *how close different words are*.

Hence, compressed caches introduce a new, fundamentally different type of side channel. As a point of comparison, consider conventional cache-based side channels (Sec. 2.1). Conventional cache channels are based on the *presence or absence* of a line in the cache, whereas compressed cache channels are based on *data compressibility* 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: 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 attacks than conventional side channel attacks. Table 1 compares Spectre and the new attacks in this paper, using the abstract model in Fig. 2. Spectre attacks allow the attacker to create different transmitters by arranging different sequences of mispredicted speculations. Analogously, attacks based on cache compression allow the attacker to create different transmitters by writing different data into the cache.

	Spectre, using cache side channels	**Compressed cache attacks (this work)**
Side channel	Line's **presence** due to a secret-dependent memory access pattern	The **compressibility** of the secret itself (and data in the same line)
Transmitter	Speculatively executed instructions	Stores to secret data or data in the same line
Receiver	Timing difference to infer a line's presence	Timing difference to infer a line's compressibility

Table 1. A comparison between Spectre and compressed cache attacks using the abstract model in Fig. 2.

2.3 Contribution 2: Compressed caches allow compressibility to be observed

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.

We propose Pack+Probe, a general technique that leverages the timing difference due to a line's presence to also infer the compressibility (Sec. 3). Like Prime+Probe, Pack+Probe fills a set with cache lines, but with known and crafted compressed sizes, and observes how many lines are evicted after a victim accesses the block to infer the size (compressibility) of the victim's cache line. Since there is a wide variety of compressed cache organizations, we describe and evaluate Pack+Probe for VSC [3], a common set-associative design, and then describe the impact on other organizations (e.g., sectored compressed caches) in Sec. 7.

Given a channel and a receiver, an attacker can now carry out passive attacks, watching for information leakage through compressibilities.

2.4 Contribution 3: Compressibility can be manipulated to leak data quickly

Passively watching for compressibilities already reveals information about the secret data. But compressed caches also enable a far more damaging *active attack*, which we call Safecracker, where the attacker manipulates data spatially close to the secret in a strategic way to affect compressibility and extract the secret efficiently (Sec. 4).

For example, consider the situation in Fig. 1, where the attacker issues encryption requests to a victim server. The server allocates the attacker's message and the secret key contiguously, so the secret shares its cache line with part of the message. Over multiple requests, the attacker tries different messages and observes how the compressibility of the line containing the secret changes. With knowledge of the compression algorithm used, we show that *the attacker can perform a guided search to recover the key.*

To make matters worse, we find there are multiple ways the attacker can colocate its data with select victim data, *enabling Safecracker to leak attacker-selected victim data.* We find that attacker-controlled colocation can be due to either valid and invalid program behavior, such as failing to zero-out freed memory or latent memory safety vulnerabilities. This, to the best of our knowledge, makes Safecracker the first read gadget caused by a microarchitectural optimization that is not related to speculative execution.

2.5 Contribution 4: The compressed cache side channel can be closed, but at a cost

Finally, we present several potential defenses against compressed cache side-channel attacks (Sec. 6). One option is to let software control compression (e.g., [91]), but this implies ISA changes, storage overheads to track which data should not be compressed, and requires programmers to correctly identify secrets. A different option is cache partitioning, which is non-trivial because compressed caches have decoupled tag and data arrays with different geometries (and both must be partitioned), and partitioning reduces compression ratio. In short, while compressed caches can be made secure, straightforward solutions come at a cost, and it will be up for future work to develop refined defenses that retain security with a lower performance impact.

3 Pack+Probe: Observing compressibility

As discussed in Sec. 2, compressed caches already provide two major components, transmitter and side channel, to construct attacks. To complete an attack, the remaining and critical component is the *receiver*, i.e., a way to observe the compressibility of cache lines in compressed caches.

Building a receiver requires understanding the architecture of compressed caches (i.e., how the cache stores and manages compressed, variable-sized blocks) and is largely orthogonal to the compression algorithm used (i.e., how the cache compresses blocks). We first review the architecture of compressed caches, then present our Pack+Probe receiver.

3.1 Background on compressed cache architectures

Whereas conventional caches manage fixed-size cache lines, compressed caches manage variable-sized blocks. Thus, compressed caches divide the data array among variable-sized blocks and track their tags in a way that *(i)* enables fast lookups and insertions, *(ii)* allows high compression ratios, and *(iii)* avoids high tag storage overheads. These requirements have led to a wide variety of compressed cache organizations. Compressed caches typically perform serial tag and data array accesses, and require extra tag entries to track more compressed cache lines than uncompressed caches.

While prior work has proposed various compressed cache architectures, Pack+Probe is general and applies broadly. For concreteness, we explain ideas using a commonly used organization, Variable-Sized Cache (VSC) [3]. We discuss how to adapt Pack+Probe to other organizations in Sec. 7.1.

VSC extends a set-associative design to store compressed, variable-size cache lines. Fig. 3 illustrates VSC and compares it with a set-associative design. VSC divides each set of the data array into small *segments* (8B in Fig. 3). 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 Fig. 3). More tags per set allow tracking a larger number of smaller lines. Increasing the number of tags adds overheads (e.g., 6% area for 2× tags) but allows higher effective compression ratios (as with highly compressible lines, the number of tags per set and not the data array limits set capacity).

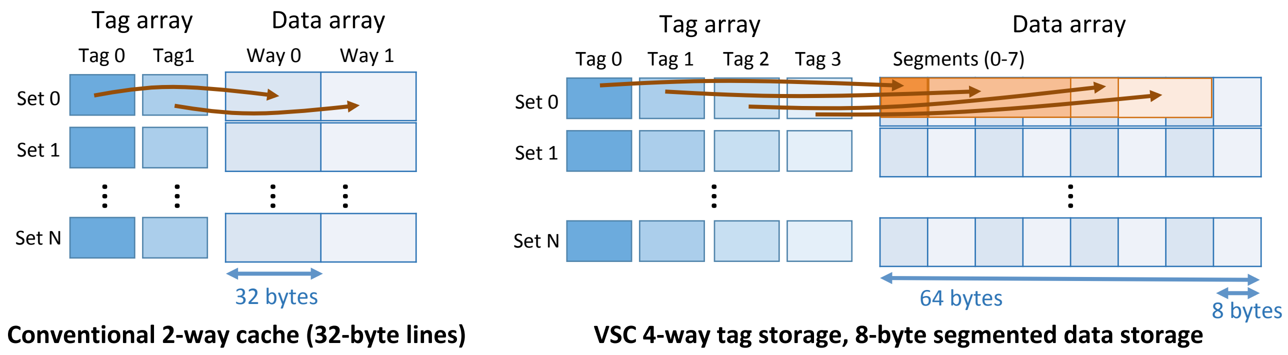


Figure 3. Comparison of VSC (right) vs. an uncompressed set-associative cache (left). VSC divides each set of the data array into small *segments* (8 bytes in this example), stores each variable-size line as a contiguous run of segments in the data array, modifies tags to point to the block's data segments, and increases the number of tags per set relative to the uncompressed cache (by 2× in this example) to allow tracking more, smaller lines per set.

VSC suffers from a few limitations, which prior work has sought to address. First, VSC increases tag overheads. To reduce these, DCC [70] and SCC [69] leverage decoupled sector caches to track multiple compressed lines per sector without extra tag entries. Second, VSC suffers from fragmentation across sets, which leaves space unused on each set. The Indirect-Indexed Cache [37] reduces fragmentation by not dividing the data array into sets and letting tags point to anywhere in the data array. Finally, VSC can require performing multiple evictions per fill, which adds complexity, and can interact poorly with cache replacement. The Base-Victim Cache [31] is a simpler organization that manages each set *and way* of the data array individually, and associates multiple tags to it. This simplifies operation but incurs additional fragmentation, reducing compression ratio.

3.2 Pack+Probe idea

Threat model: Our threat model is that victim and attacker are two processes sharing the processor and a compressed cache. Attacker and victim can be on different cores as long as those cores share the compressed cache (which is typically the last-level cache). The attacker wants to learn information about data in the victim's protection domain by observing the compressed size of the victim's cache lines.

We assume the attacker knows the compression algorithm used, so that it can construct cache lines of known compressed sizes. We also assume that the attacker knows when the victim has accessed the secret data (e.g., by invoking victim code), so that it can run the receiver afterwards.

Key idea: 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.

To exploit this, Pack+Probe first *packs* the cache set with attacker-controlled lines to leave exactly *X bytes unused*, as well as enough tag space to avoid evictions due to insufficient tags. Once the victim accesses the target cache line, if its compressed size is ≤X bytes, no evictions will happen,

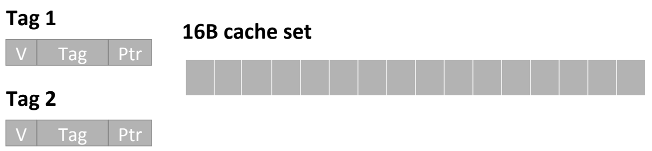
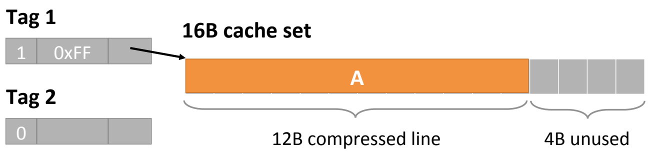
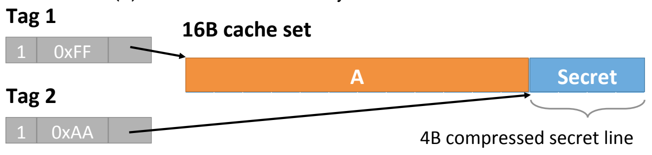


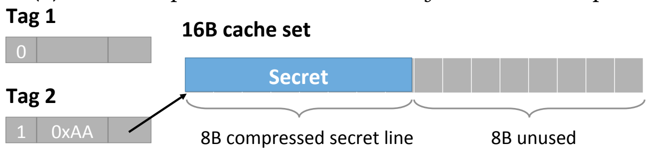
Figure 4. A simplified, 16B cache with 2 decoupled tags.



(a) Attacker leaves 4 bytes in the cache set.



(b) If the compressed secret line is $\leq 4 bytes$, then A is kept.



(c) If the compressed secret line is $> 4 bytes$, then A is evicted.

Figure 5. A simplified example of Pack+Probe.

whereas if it is larger than X bytes, at least one of the attacker-controlled lines will be evicted. Finally, the attacker *probes* 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.

A simple example: Consider the simplified compressed cache in Fig. 4, which has only one set, with 16 bytes in the data array and two decoupled cache tags.

Fig. 5 shows a single step of Pack+Probe, where the attacker tests whether the victim's line compresses to 4 bytes or smaller. The attacker first sets up the cache set by accessing an incompressible, 16-byte line (which evicts everything else in the data array), and then a 12-byte cache line, A. This leaves the cache with an empty tag and 4 bytes of space in the data array. Then the victim accesses the target cache line. If the target cache line is larger than 4 bytes, then it will evict A; otherwise, A will be kept. Finally, the attacker probes A to determine whether it's still in the cache, and thus whether the victim's line compresses to 4 bytes or smaller.

3.3 Pack+Probe implementation on VSC

We now build on the key idea of Pack+Probe to construct a concrete receiver for VSC (Sec. 3.1). As previously discussed, we choose VSC because it is the most commonly used in cache compression work [8, 9, 17, 41, 42, 60, 70, 81].

There are two differences between VSC and our simple example. First, VSC has multiple sets, so Pack+Probe requires finding the target set. Second, each VSC set has more than two tags and compresses lines to segments rather than bytes.

Before running Pack+Probe, the attacker uses standard techniques like Prime+Probe [57] to find the target set. With the target set known, Pack+Probe proceeds in multiple steps as outlined above. Pack+Probe finds the compressed size in segments (e.g., 8B chunks in Fig. 3), which is the allocation granularity in VSC, rather than bytes. Nonetheless, as we will see, this coarser granularity suffices to leak secret data.

Assume that the uncompressed cache lines are U segments long. Pack+Probe performs a binary search as follows. In the first step, Pack+Probe packs the target cache set with lines that leave $U/2$ segments and at least one tag unused. Then, the victim accesses the target line, and the attacker infers whether the line is $\leq U/2$ segments. In the next step of the binary search, Pack+Probe packs the target set to leave either $U/4$ unused segments (if victim line $\leq U/2$ segments) or $3U/4$ unused segments (if victim line $> U/2$ segments). As each step cuts the search space in half, Pack+Probe finds the target line size in $log_2(U)$ steps. Each step contains a few main memory accesses (plus a victim function call), taking less than 10K cycles in our experiments (Sec. 5).

This approach relies on leaving one tag unused to avoid tag conflicts. This is easily achievable, as tags are overprovisioned and the attacker can use incompressible lines to fill most of the set without using up the tags. For example, in a VSC cache with 64-byte uncompressed lines, 16·64 bytes (1024 bytes) per set in the data array, and 32 tags per set (i.e., a 2× tags design), the attacker can fill 15/16ths of the data array with 15 incompressible lines, leaving 17 tags to fill with compressible lines and perform the search.

Our Pack+Probe implementation is simple but general. If the attacker knows the replacement policy, more complex Pack+Probe variants exist that reduce the number of steps by watching for multiple evictions. For example, LRU enables a one-step Pack+Probe that first installs U 1-segment compressed lines in the U LRU positions, then counts the number of evictions to infer the size of the victim's line. Nonetheless, we show that our simple and general Pack+Probe implementation is fast enough for practical attacks.

4 Safecracker: Recovering secrets efficiently

While Pack+Probe lets the attacker passively learn the compressibility of secret data, this information alone may not suffice to leak secret data. For example, the secret may be an incompressible random string as in a cryptographic key. Although learning that this string is incompressible does leak something, it does not reveal the exact bits in the string.

To efficiently and precisely recover the secret, we propose Safecracker, an active attack that exploits attacker-controlled data colocated with secret data. 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.

Depending on the compression algorithm used, Safecracker needs different search strategies. Before explaining Safecracker, we review prior cache compression algorithms.

4.1 Background on cache compression algorithms

To design a compression algorithm, architects have to balance compression ratio and decompression latency. Since decompression latency adds to the critical path memory latency, most compressed caches forgo some compression opportunities in favor of simpler decompression logic.

Most compression algorithms compress each cache line individually, i.e., they exploit redundancy within a cache line but not across cache lines. For example, ZCA [24] removes zeros in a cache line. Frequent pattern compression (FPC) [4] recognizes repeated patterns or small-value integers and uses a static encoding to compress every 32-bit data chunk in the line. Base-Delta-Immediate (BDI) [62] observes that values in a cache line usually have a small dynamic range, and compresses the line into a base plus per-word deltas.

To achieve higher compression ratios, recent work also considers compressing consecutive cache lines. DISH [60] builds on decoupled sector caches to compress a super block by sharing the dictionary across multiple cache lines. Mbzip [40] extends BDI to store one base value for consecutive cache lines in the same set.

Finally, to achieve even higher compression ratios, some techniques look for redundancy across the whole cache. These designs achieve the highest compression ratio, but also incur significant design changes. For example, cache deduplication [76] and Doppelganger [51] add hardware to support content-based indexing. SC^2 [9] monitors the data of all cache misses to build an appropriate Huffman code, which it then uses to compress all cache lines.

Name	Compressed Size	Pattern	Group
B1D0	1	All zeros	(1,8]
B8D0	8	Eight same 8B value	(1,8]
B8D1	16	8B base + 8×1B deltas	(8,16]
B4D1	20	4B base + 16×1B deltas	(16,24]
B8D2	24	8B base + 8×2B deltas	(16,24]
B2D1	34	2B base + 32×1B deltas	(32, 40]
B4D2	36	4B base + 16×2B deltas	(32, 40]
B8D4	40	8B base + 8×4B deltas	(32, 40]
NoComp	64	Not compressed	(56, 64]

Table 2. BDI compression algorithm for 64-byte lines [62, Table 2].

We focus on building a Safecracker attack for compression algorithms that compress each line independently, both because these compression schemes are the most common and because they are *harder to exploit*. Algorithms that rely on shared state across lines open up additional opportunities (Sec. 7) for active attacks. For example, in SC^2, the attacker could leak across protection domains because cache lines for different domains are compressed together.

As discussed in Sec. 2.2, in general, the better the compression algorithm, the more information it reveals. For example, ZCA only eliminates lines full with zeros, which reveals very limited information. But we find relatively simple algorithms suffice for Safecracker to leak secret data efficiently. Specifically, we target the BDI algorithm [62], a simple and widely used algorithm that relies on delta encoding.

BDI algorithm: The Base-Delta-Immediate (BDI) compression algorithm [62] performs intra-cache-line compression by storing a common base value and small deltas. For example, for eight 8-byte integer values ranging from 1280 to 1287, BDI will store an 8-byte base of 1280, and eight 1-byte values from 0 to 7. With 64-byte cache lines, depending on the base value and the ranges of the deltas, BDI compresses a cache line into 8 different sizes, shown in Table 2. If none of these patterns exist, BDI stores the cache line uncompressed.

4.2 Safecracker implementation on BDI

Since Safecracker's search process depends on the compression algorithm, we demonstrate the idea directly with our implementation on BDI, and discuss other algorithms in Sec. 7.

Threat model: Safecracker assumes the attacker can colocate *attacker-controlled data* in the same cache line as victim secret data. There are various real-world scenarios where this can happen for semantically correct programs. An obvious example is if the programmer explicitly allocates secrets next to (attacker-controlled) program inputs, as we saw in Fig. 1. More subtle, the compiler may colocate secrets with attacker data, e.g., due to pushing function arguments or spilling register values to the stack. Further, semantically incorrect (but also real-world) programs, i.e., those with latent memory safety vulnerabilities, create even worse problems. For example, heap sprays and buffer overflows enable the attacker to co-locate its data with potentially any secret data.

Beyond colocation, we assume that the attacker can measure the compressibility of lines with colocated data, e.g., with Pack+Probe. And again, we assume the attacker knows the compression algorithm used.

Key idea: Safecracker exploits that, with attacker-controlled data colocated with the secret and knowledge of the compression algorithm, the attacker can perform a guided search on the compressibility of the cache line to leak the secret.

To exploit this, the attacker first makes a guess about the secret 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 Fig. 1). By measuring the compressibility of the line, the attacker knows whether the guess was correct or not.

Moreover, the compression algorithm often allows the attacker to make *partial guesses* about the secret, e.g., guessing on particular bytes or half-words of the secret. Thus, to leak an X-bit secret, the attacker need not guess $O(2^X)$ times. Instead, the attack can be divided into sub-steps to leak parts of secret more cheaply.

Safecracker attack on BDI: Assume that the attacker wishes to steal an N-byte secret, located in a cache line where all other data is attacker-controlled. Since BDI is a delta encoding technique, Safecracker works by guessing a base value that is close enough to the secret content so that when the attacker fills the line with this base value, it triggers compression for the whole line. The closer the guess is, the smaller the cache line compresses, enabling a multi-step approach.

For an N-byte secret, the Safecracker algorithm on BDI works as follows:

- Select a size of the base value M that is larger than or equal to the size of the secret, i.e., $M \geq N$.
- Target a compression pattern using that base value size with the largest delta X (BMDX in Table 2).
- Brute-force the base value in the attacker-controlled data and measure the compressed size (using Pack+Probe) until the cache line is compressed to the pattern (BMDX).
- Record the base value that causes compression and target a pattern with a smaller delta X'.
- Repeat the previous two steps until the pattern is B8D0 (i.e., smallest size, all 8-byte values are the same). This means *the guessed base value is the secret.*

For example, if the secret is a 4-byte value and attacker controls the content of the remaining 60 bytes of the cache line, the attacker uses a 4-byte base and starts by targeting the B4D2 pattern. It then brute-forces the first two bytes of every 4-byte word in the attacker-controlled data (i.e., trying patterns `0x00000000,0x00010000,...0xFFFF0000`, at most 2^{16} guesses), and uses Pack+Probe to see if the cache line

Bytes	Sequence	Attempts
2B	NoComp -> B2D1 -> B8D0	$O(2^8)$
4B	NoComp -> B4D2 -> B4D1 -> B8D0	$O(2^{16})$
8B	NoComp -> B8D4 -> B8D2 -> B8D1 -> B8D0	$O(2^{32})$

Table 3. Safecracker brute-force sequences for BDI.

compresses to the B4D2 pattern. Once it sees the B4D2 pattern, the attacker records the two bytes it tried, targets the B4D1 pattern, and brute-forces the third byte (taking at most 2^8 guesses). When the attacker sees the size corresponding to the B4D1 pattern, it then targets B8D0 and brute-forces the last byte; and when the attacker sees the B8D0 pattern's size, *the 4-byte guess matches the 4-byte secret.*

Since the largest base in BDI is 8 bytes, Safecracker can steal up to 8 bytes in a 64-byte line. Though it cannot leak all memory, leaking 8 bytes of secret data 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 [14].

Table 3 shows the sequence of target patterns and observed sizes that Safecracker uses to steal contents ranging from 2 bytes to 8 bytes. As shown in Sec. 3, in VSC, Pack+Probe only learns the compressed size at 8-byte granularity. Thus, each step in Table 3 falls into different groups (multiples of 8 bytes), so that Safecracker can observe changes.

Enhancing Safecracker with buffer overflows: So far we have assumed that the attacker-controlled data is a fixed-size buffer located right next to the secret. This limits the attacker to only leak contents contiguous to attacker-controlled data.

However, if the attacker can find a buffer overflow vulnerability in the victim program, then this vulnerability can significantly increase the amount of data leaked and enhance the efficiency of Safecracker:

1. Reach: First, classic buffer overflow attacks allow the attacker to control *where* the attacker-controlled data is located. For example, if a victim suffers a buffer overflow on a stack-allocated array, and there is a secret elsewhere in the stack, the attacker can exploit the buffer overflow to place attacker-controlled data right up to the (non-contiguous) secret.

In the worst case, if the compression algorithm allows leaking up to X bytes per line, then by using a buffer overflow, Safecracker could be applied line by line, so the attacker can leak X bytes in *every cache line.* That is, if the victim has a memory footprint of M, then Safecracker with 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). In fact, there are compression algorithms that allow leaking the whole line with Safecracker, like simple run-length encoding (RLE), where single-bit changes cause compressibilitiy to change. Thus, a very sensitive compression algorithm, combined with a vulnerability that lets the attacker place data in the victim's memory, can give the attacker a *universal read gadget* as powerful as Spectre.

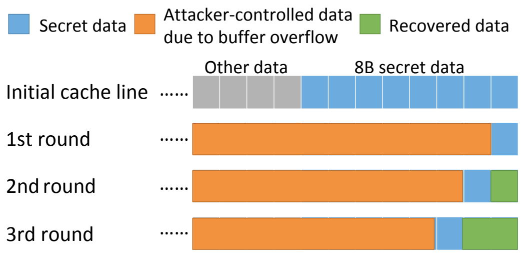


Figure 6. Buffer overflows further speed up Safecracker.

2. Efficiency: Second, buffer overflows also allow the attacker to control *how many bytes* the attacker-controlled data has. Therefore, by repeatedly performing buffer overflows with different buffer sizes, an attacker can recover the secret much faster than in the previous example.

Fig. 6 shows an example of Safecracker using a buffer overflow to leak secret data more efficiently. To start, the attacker allocates a buffer (the orange region) 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 with about 2^8 tries, because it divides the task into 8 sub-steps. With BDI, this improves efficiency susbtantially, as stealing 8 bytes of secret data takes take 8×2^8 tries in the worst case, instead of 2^{32} (Table 3).

Each interaction in the above fashion may eventually cause a program crash (e.g., the buffer overwrites a control variable). Yet, many services restart after a crash [15], thus allowing the attacker to extract secrets over multiple crashes.

Note that while a traditional buffer overflow exploit requires subsequent steps (e.g., to find ROP gadgets), this new exploit leaks data *at the buffer overflow step.* This defeats certain software defenses, such as stack canaries, since leakage occurs before the buffer allocation completes. We demonstrate this by adding compiler protection (i.e., `-fstack-protect`) in our evaluation (Sec. 5).

5 Evaluation

In this section we evaluate the effectiveness of Safecracker using two Proof-of-Concept workloads (PoC). The first PoC exploits static colocation of secret data and attacker-controlled input, and the second PoC exploits a buffer-overflow-based vulnerability to colocate the data dynamically. In Sec. 7.2, we discuss other ways to achieve effective colocation.

5.1 Baseline system and methodology

We evaluate our PoCs using architectural simulation, as we do not have access to hardware with a compressed cache.

Core	x86-64 ISA, 2.3 GHz, Skylake-like OOO [21, 68]: 32B-wide ifetch; 2-level bpred with 2k×18-bit BHSRs + 4k×2-bit PHT, 4-wide issue, 36-entry IQ, 224-entry ROB, 72/56-entry LQ/SQ
L1	32 KB, 8-way set-associative, split D/I caches, 64 B lines, LRU
L2	256 KB private per-core, 8-way set-associative, LRU
LLC	8 MB shared, 16-way set-associative, LRU
Mem	3 DDR3-1333 channels
Algorithm	BDI [62] compression algorithm
Architecture	VSC [3] compressed cache (2× tag array)

Table 4. Configuration of the simulated system.

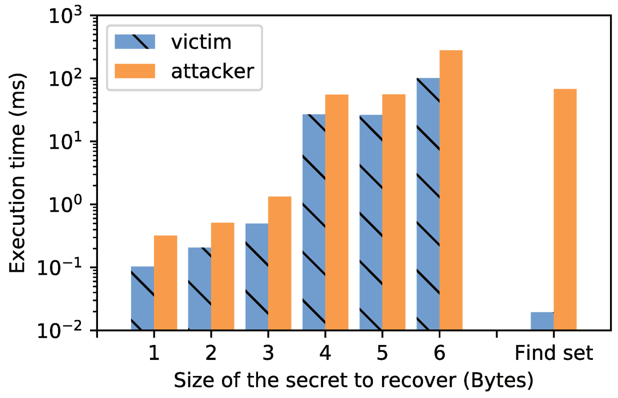


Figure 7. Safecracker results in simulation: worst-case time to leak secret with attacker-controlled data in a fixed-size buffer.

Simulated system: We use ZSim [68], an execution-driven, Pin-based [49] simulator. We model a four-core system with a three-level cache hierarchy similar to Skylake [21], with a compressed last-level cache. Table 4 shows the parameters of the simulated system. We use `-fstack-protect` to compile all programs, which blocks conventional buffer overflows.

5.2 PoC 1: Static colocation

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 allocated, 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, as explained in Sec. 3.2.

The attacker first finds the set that holds the secret cache line using standard Prime+Probe [57]. Once the conflicting set is found, the attacker follows the Safecracker process to steal the secret key of the victim.

Safecracker steals secrets efficiently: Fig. 7 shows 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. For a secret with 1–3 bytes, most of the overhead comes from finding the conflicting set (which takes 50 ms), and when stealing 4–6 bytes, finding the set is still a significant part of the runtime.

Though Safecracker can steal up to 8 bytes when applied to BDI, trying to steal more than 6 bytes requires much longer run-times. As shown in Table 3, the complexity of Safecracker on BDI grows exponentially. So when stealing 4 bytes, the guess is over 2 bytes in a single step. Compared with guessing only 1 byte in a single step (stealing 2 bytes), this increase in complexity already incurs in a substantial overhead, as shown in Table 3. With a similar exponential growth, to steal 7 bytes, the attacker needs to guess 3 bytes in a single step, which by extrapolation would take about 16 seconds (which is too long to simulate). Finally, to steal 8 bytes, the attacker must guess 4 bytes ($O(2^{32})$ tries) in a single step, which would take take 90 hours wall-clock time.

Finally, Fig. 7 shows that the victim takes at most 20% of the execution time of the attacker. This is due to the communication between victim and attacker and the time taken by attacker-triggered requests.

5.3 PoC 2: Buffer-overflow-based attack

As in Sec. 5.2, the second PoC consists of separate victim and attacker processes. However, this time, the victim has a buffer-overflow vulnerability. Its encryption process uses two local arrays, to store the key and an external input, respectively. 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 in Section 4.2.

Efficiency: As explained in Sec. 4.2, buffer overflows give Safecracker much higher bandwidth by allowing it to guess a single byte on each step. Using buffer overflow, Safecracker steals 8 bytes of secret data in under 10 ms. Fig. 8 shows that attack time with a buffer overflow grows linearly, vs. exponentially with static colocation (Fig. 7).

While Safecracker applied to BDI can steal only 8 bytes per line, this buffer-overflow attack could extend to steal data in other lines, e.g., using longer overflows to steal secrets from lines further down the stack.

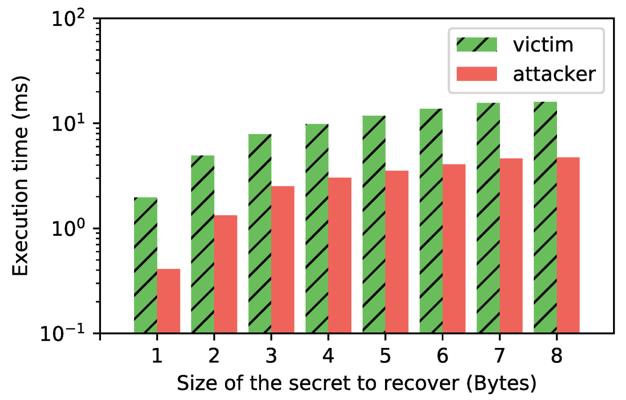


Figure 8. Safecracker results in simulation: worst-case time to leak secret with a buffer-overflow-based attack.

Finally, note that other compression algorithms can allow stealing more data. For example, with run-length encoding (Sec. 4.2), Safecracker would leak beyond the 8-byte-per-line limit, at an expected rate of hundreds of bytes per second.

6 Defenses against cache compression attacks

To defend against Pack+Probe, Safecracker, and other attacks on compressed caches, architects must disable at least one of the three components in Fig. 2: channel, transmitter, or receiver. We discuss a few alternative approaches to do so.

Preventing compression of sensitive data: Compressed cache attacks are ineffective if the cache does not compress sensitive data. This can be implemented with hardware support, e.g., by letting software specify address ranges or pages that should never be compressed. However, this requires additional state in hardware to track incompressible data. Software-only mitigations are also possible, e.g., by padding every word of sensitive data with enough incompressible data to render the line incompressible given the machine's compression algorithm. However, this approach is inefficient and brittle, as changes to the compression scheme may make the chunk compressible. Moreover, both hardware and software mitigations require program modifications and rely on correct identification of sensitive data, which is hard.

Partitioning the compressed cache: Cache partitioning [10, 67, 79] provides isolation between processes sharing the compressed cache. Unlike the previous approach, partitioning prevents compressed cache attacks without software changes. This approach resembles prior partitioning-based mitigations [43, 47] for conventional caches. However, compressed caches have a different organization from conventional ones, so partitioning them is not as easy.

Specifically, to ensure full isolation, both tag and data arrays must be partitioned. Partitioning only the tag array or the data array does not provide full isolation, since a different process can detect compressibility indirectly, from pressure on either the data array (as we saw with Pack+Probe) or on the tag array. If only the data array is partitioned, the attacker can still *prime* the whole tag array by filling the attacker's partition with all compressed lines (e.g., all-zero lines) and then observe the pressure from the victim in the tag array to learn the compressibility of victim data.

Obfuscation-based defenses: Since Pack+Probe relies on timing measurements to learn compressibility, mitigations that only expose coarse-grained timings [38] can reduce the receiver's effectiveness. Also, randomized cache replacement policies [48] and architectures [30, 64] add significant noise to the channel and prevent precise per-line compressibility measurements in Pack+Probe. However, the attacker may still learn valuable information, e.g., by monitoring the overall compressibility of the victim's working set.

6.1 Partitioning compressed caches

As mentioned earlier, partitioning is an effective way to to prevent compressed cache attacks by providing full cache isolation. However, partitioning a compressed cache limits the performance benefits of compression. To see how partitioning affects performance, we study four cache architectures:

- Uncompressed cache without partitioning
- Compressed cache using VSC+BDI (see Sec. 5)
- Conventional cache with static way partitioning
- Compressed cache with both static tag and data array partitioning

Static partitioning evenly divides cache resources among four cores in the system. For compressed caches, we partition both the tag array and data array to provide isolation.

We simulate mixes of SPEC CPU2006 apps on systems with these four configurations. We use the 12 compression-sensitive SPEC CPU2006 apps, and we use weighted speedup as our performance metric and use uncompressed and unpartitioned cache as our baseline. Our benchmark selection and performance metric mirror the methodology in the BDI paper [62]. We fast-forward all apps in each mix for 20B instructions. We use a fixed-work methodology similar to FIESTA [39]: the mix runs until all apps execute at least 2B instructions, and we consider only the first 2B instructions of each app to report performance.

Fig. 9 shows the weighted speedup over the baseline (uncompressed and unpartitioned cache) for different cache organizations on 25 4-app mixes. The compressed, unpartitioned cache improves performance by up to 14% and by 3.2% on average over the baseline (these benefits are similar to those reported in prior work). However, static partitioning hurts performance for both the uncompressed and compressed caches. Static partitioning hurts performance of the baseline by up to 16% and by 4% on average. It also hurts performance of the compressed cache by up to 13% and by 5% on average. The performance loss in the partitioned compressed cache

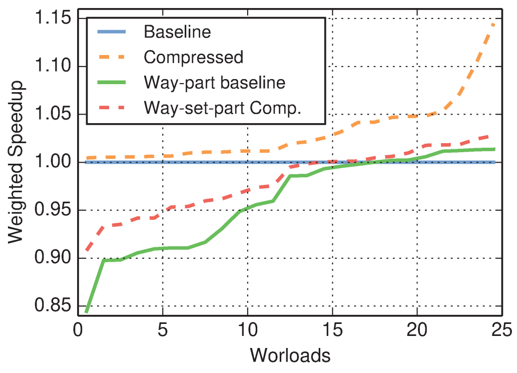


Figure 9. Weighted speedup over uncompressed shared cache on 4-application mixes.

stems from not only the reduced number of ways and capacity, but also a decrease in compression ratio, which is due to limited tag and data array resources. For example, the data array has more internal fragmentation, as each partition is managed independently.

These results show 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 with dynamic partitioning would be interesting future work.

7 Discussion

In this section, we discuss how to generalize our attacks to other compressed caches, and how Safecracker can exploit other ways to colocate data.

7.1 Generalizing attacks to other compressed caches

While we implement and evaluate our attacks on VSC+BDI, Pack+Probe and Safecracker apply to other cache architectures and algorithms.

Attacking other compressed cache architectures: As discussed in Sec. 3.1, most compressed cache architectures share the two ingredients Pack+Probe uses to attack VSC: a decoupled tag store with extra tags, and a data array divided in fixed-size sets where variable-sized blocks are laid over. With these two ingredients, it's easy for Pack+Probe to generate data-array capacity pressure to find the size of a victim line.

There are two important exceptions to the above. First, architectures with decoupled tag and data stores, such as IIC [37], V-Way cache [65], and Zippads [80, 81], do not divide the data array in sets, and tags can point anywhere in the data array. This makes Pack+Probe much harder, as getting a conflict requires a full pass over the data array, and noise from other accesses may make measuring the compressibility of an individual line infeasible.

Second, some architectures have limited or no tag overprovisioning. Specifically, DCC [70] and SCC [69] use decoupled sector caches to track multiple compressed lines. Performing Pack+Probe on them requires carefully constructing access and data patterns that leverage multi-block tags and leave the right amount of space with some tags unused. For example, since DCC has the same number of cache tags as baseline, it is important to make sure some tags track more than two lines so that there are no tag conflicts during Pack+Probe.

Attacking other compression algorithms: While we demonstrate Safecracker on BDI, we believe it is straightforward to come up with a search process for other compression algorithms, though the amount of information learned will vary. As discussed in Sec. 4.1, the better the algorithm compresses, the more information it can leak.

More importantly, if the compression algorithm performs compression across protection domains (e.g., schemes that compress across lines such as cache deduplication [76]), then there is opportunity to leak even more data. For example, if the size of an attacker-controlled line is influenced by kernel data, Safecracker could be used to leak this kernel data.

7.2 Colocating attacker-controlled data

In Sec. 4 and Sec. 5 we consider two ways for the attacker to colocate data. But there are many more ways to colocate attacker-controlled data near sensitive data, since programmers do not avoid this scenario intentionally. For example, for sensitive data in the memory heap, attackers can leverage both *spatial* and *temporal* colocation opportunities.

Spatial heap-based attacks: *Heap spraying* is an effective way to increase the chance to colocate data. In heap spraying, the attacker fills the victim's heap with its own data at various places (e.g., by forcing the victim to allocate data). This increases the possibility that attacker-controlled data is next to some sensitive data. Once the attacker succeeds, it can apply a brute-force attack similar to the chosen-plaintext example (Sec. 5) by changing the values of its data in the heap, or forcing the victim to free and reallocate the data.

Temporal heap-based attacks: The attacker may also learn information about data not at the edge of an object by taking advantage of uninitialized memory. The idea is that before being allocated, the addresses to store new secret data may be storing attacker-controlled data or vice versa. If the private object is not initialized, e.g., zeroed-out, on allocation, old attacker data written previously can locate right next to the secret data.

We manually inspected the source of several real-world programs and found exploitable patterns for temporal attacks. For example, a common pattern in `OpenSSL-1.1.1` is:

```
buf = malloc(len);
memcpy(buf,user_input,len);
```

That is, the program allocates and fills a buffer without zero-ing the buffer contents in between, similar to our PoC 2 in Sec. 5.3. Combined with Safecracker, this code is akin to Heartbleed [23], giving the attacker a chance to learn the heap data stored at the buffer address prior to the allocation.

As another example, Linux's kernel allocator lets objects used in different protection domains share cache lines by default. This colocation due to a centralized memory allocator could allow leaking kernel data using Safecracker by changing same-line, attacker-controlled data (e.g., a file buffer).

While there are many opportunities to colocate attacker-controlled data with sensitive data, some cases are harder or more expensive to exploit. For example, if colocating sensitive data crashes the victim program, the wait between two Safecracker attempts might be too long for the full attack to be useful. A quantitative analysis of the side-channel bandwidth due to different compression algorithms under various colocation opportunities is interesting future work.

8 Related work

8.1 Microarchitectural side/covert channel attacks

This paper builds on a rich literature on microarchitectural side/covert channel attacks. Many processor structures have been shown to leak privacy over these channels, including a variety of cache architectures [57, 87–89], branch predictors [1, 27], pipeline components [5, 7, 34], and other structures [26, 33, 53, 63, 83, 85]. All these channels reveal information about data "in transit," i.e., being operated on by specific instructions in the sender (victim) program. By contrast, leaking privacy through cache compression is, to our knowledge, the first microarchitectural side/covert channel that leaks information about data at rest. That is, compression applies in the same way, regardless of what instructions were used to produce the data.

Prior work on Data Oblivious ISA extensions [91] briefly mentions the use of cache compression to defeat constant time/data oblivious programming, but does not go into details or allude to active cache compression attacks (Sec. 4).

8.2 Defenses on cache side-channel attacks

Prior work has also studied how to thwart the various cache side-channel attacks found in commercial machines.

On the one hand, software techniques try to eliminate cache side channels from existing systems. For example, some techniques use cache partitioning hardware [43, 47] or segregate threads from different users into different cores [29] to avoid cache side-channel attacks. On the other hand, hardware techniques aim to develop mechanisms that balance performance and security. These techniques include adding randomness to the system [48, 64] or changing the cache organization [84, 86].

In Sec. 6, we discussed how some of these techniques may be adapted to prevent attacks on compressed caches.

8.3 Attacks on software compression

Prior work has already identified that compression and variable-length messages can leak information in other contexts, such as HTTP requests [82]. Similarly, attacks on memory deduplication [35, 58, 75] have shown that page deduplication in virtualized environments can leak information about the host or be used as a covert channel between guests.

Our contribution over this prior work is to realize that compressed caches are especially affected by these problems, and to develop practical attacks that demonstrate the extent of the problem. Beyond the severity of the problem, a key difference with compressed caches is that they are *software-transparent*, so software has no control over whether data is compressed or not. Moreover, since hardware patches are much more difficult to apply, it is important to prevent insecure cache compression implementations from reaching mainstream processors.

9 Conclusion

We have presented the first security analysis of cache compression, a promising technique that is likely to appear in future processors. While to the best of our knowledge no commercial general-purpose processors implement compressed caches, the recent security crisis brought on by speculative execution attacks has shown that it is crucial to analyze the security of architectural techniques *before* they are deployed in hardware.

We find that cache compression is insecure 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 it.

We present two techniques that make attacks on compressed caches practical: Pack+Probe lets an attacker learn the compressibility of victim cache lines, and Safecracker exploits colocation with attacker-controlled data to leak secret data efficiently. Our evaluation shows that, on a common compressed cache architecture, Safecracker lets an attacker compromise a secret key in under 10 ms. We also present potential defenses against compressed cache attacks, and show that partitioning, the most complete one, comes at a cost.

We hope this work sparks follow-on research on high-performance defenses and, more importantly, prevents insecure cache compression techniques from reaching mainstream processors, averting a potential security crisis.

Acknowledgments

We sincerely thank Maleen Abeydeera, Joel Emer, Mark Jeffrey, Anurag Mukkara, Quan Nguyen, Victor Ying, Guowei Zhang, and the anonymous reviewers for their feedback. We thank Joel Emer for his insights on the taxonomy in Sec. 2. We thank Paul Kocher for sharing his concerns about the security of memory compression. This work was supported in part by NSF grants CAREER-1452994 and SaTC-1816226, a Google faculty research award, and an Intel ISRA grant. Andres Sanchez was supported by a MISTI grant by the Technical University of Madrid.

A Artifact Appendix

A.1 Abstract

Our artifact includes the source code and simulation framework for the two PoC attacks in Sec. 5, as well as scripts to run the experiments and reproduce the results. The source code includes a self-contained library of the Safecracker attack to the BDI compression algorithm described in Sec. 4, which can be reused in other PoCs.

To ease reproducibility, our artifact uses Vagrant to automatically set up and provision a virtual machine with all necessary dependences.

A.2 Artifact check-list (meta-information)

- **Compilation:** `scons, make, gcc`
- **Run-time environment:** Any OS where Vagrant is supported, or Linux for native execution.
- **Hardware:** Any x86-64 platform.
- **Execution:** About 5 minutes for the main experiments.
- **Metrics:** Simulated cycles.
- **Output:** Proof-of-concept attacks and simulation results.
- **How much disk space required (approximately)?:** <100MB
- **How much time is needed to prepare workflow (approximately)?:** About 30 minutes.
- **How much time is needed to complete experiments (approximately)?** About 30 minutes.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** GNU GPL v2.
- **Workflow framework used?:** No.
- **Archived (provide DOI)?:** 10.5281/zenodo.3560520 (https://zenodo.org/record/3560520)

A.3 Description

A.3.1 How delivered

Our artifact is available at `DOI:10.5281/zenodo.3560520` (https://zenodo.org/record/3560520) and is packaged as a tarball (`cc_artifact_evaluation.tar.gz`).

A.3.2 Hardware dependencies

Any x86-64 platform.

A.3.3 Software dependencies

The recommended Vagrant-based setup should work on any system where Vagrant is supported (e.g., Linux, Windows, MacOS), and only requires Vagrant to be installed (available at https://www.vagrantup.com/). We also recommend using VirtualBox (https://www.virtualbox.org/) with Vagrant.

Alternatively, a Linux system (preferebly Ubuntu 14.04) can be used for native execution.

A.4 Installation

Please follow the `README.md` in the tarball to install zsim and compile two proof-of-concept workloads. We highly recommend to use Vagrant (http://vagrantup.com) to set up the environment. The artifact includes a `Vagrantfile` (Vagrant configuration file) that automatically provisions a VM with all the dependences. Assuming you have Vagrant installed (`sudo apt-get install vagrant` on Ubuntu or Debian), follow these steps:

```
tar -xvf cc_artifact_evaluation.tar.gz
cd cc_artifact_evaluation
vagrant up
vagrant ssh
```

Once logged into in the VM, simply build the simulator and PoCs by running:

```
cd /vagrant/zsim_simulator/
scons -j4 # Build zsim
cd ..
make # Build PoCs
```

A.5 Experiment workflow and expected results

Once both the simulator and workloads are successfully built, under `/vagrant/plots`, executing

```
bash generate.sh
```

will simulate our PoCs using different sizes for the secret and generate plots in Fig. 7 and Fig. 8. If you are not using Vagrant, edit `generate.sh` to set `$ZSIM` to the path to zsim.

During the simulation, the victim will first report the secret key (e.g., `[S] Key is: 41,224,151,78,`), and the attacker will perform our Safecracker attack to leak and report the key (e.g., `[A] Secret value is: 41,224,151,78,`).

Finally, `/vagrant/README.md` provides detailed documentation on how to read and use our APIs. We hope this documentation helps artifact users understand and reuse our infrastructure.

References

[1] Onur Aciicmez, Cetin Kaya Koc, and Jean-Pierre Seifert. 2006. Predicting Secret Keys via Branch Prediction. In *Proc. of the 7th Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA'07)*.

[2] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. 2018. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX. In *Proc. of the 25th Annual Network and Distributed System Security Symposium (NDSS-25)*.

[3] Alaa R Alameldeen and David A Wood. 2004. Adaptive cache compression for high-performance processors. In *Proc. of the 31st annual Intl. Symp. on Computer Architecture (ISCA-31)*.

[4] Alaa R Alameldeen and David A Wood. 2004. *Frequent pattern compression: A significance-based compression scheme for L2 caches.* Technical Report 1500. Dept. Comp. Sci., Univ. Wisconsin-Madison.

[5] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2018. Port Contention for Fun and Profit. Cryptology ePrint Archive, Report 2018/1060. https://eprint.iacr.org/2018/1060.

[6] AMD. 2015. AMD Radeon R9 Fury X Graphics Card.

[7] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. 2015. On Subnormal Floating Point and Abnormal Timing. In *Proc. of the 2015 IEEE Symposium on Security and Privacy (SP)*.

[8] Angelos Arelakis, Fredrik Dahlgren, and Per Stenstrom. 2015. HyComp: A hybrid cache compression method for selection of data-type-specific compression methods. In *Proc. of the 48th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-48)*.

[9] Angelos Arelakis and Per Stenstrom. 2014. SC2: A statistical compression cache scheme. In *Proc. of the 41st annual Intl. Symp. on Computer Architecture (ISCA-41)*.

[10] Nathan Beckmann, Po-An Tsai, and Daniel Sanchez. 2015. Scaling Distributed Cache Hierarchies through Computation and Data Co-Scheduling. In *Proc. of the 21st IEEE intl. symp. on High Performance Computer Architecture (HPCA-21)*.

[11] Daniel J. Bernstein. 2005. *Cache-timing attacks on AES.* Technical Report. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[12] Daniel J. Bernstein. 2005. The Poly1305-AES Message-Authentication Code. In *Intl. Workshop on Fast Software Encryption*.

[13] Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *Intl. Workshop on Public Key Cryptography (PKC 2006)*.

[14] Karthikeyan Bhargavan and Gaëtan Leurent. 2016. On the practical (in-) security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In *Proc. of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*.

[15] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. 2014. Hacking Blind. In *Proc. of the 2014 IEEE Symposium on Security and Privacy (SP)*.

[16] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proc. of the 27th USENIX Security Symposium (USENIX Security 18)*.

[17] Esha Choukse, Mattan Erez, and Alaa Alameldeen. 2018. Compresso: Pragmatic main memory compression. In *Proc. of the 51st annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-51)*.

[18] Esha Choukse, Mattan Erez, and Alaa Alameldeen. 2018. Compress-Points: An evaluation methodology for compressed memory systems. *Computer Architecture Letters* 17, 2 (2018).

[19] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. 2009. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *Proc. of the 2009 IEEE Symposium on Security and Privacy (SP)*.

[20] Thomas M Cover and Joy A Thomas. 2012. *Elements of information theory.* John Wiley & Sons.

[21] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. 2017. Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake. *IEEE Micro* 37, 2 (2017).

[22] Goran Doychev and Boris Köpf. 2017. Rigorous Analysis of Software Countermeasures Against Cache Attacks. *ACM SIGPLAN Notices* 52, 6 (2017).

[23] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and John Halderman. 2014. The matter of Heartbleed. In *Proc. of the 2014 Internet Measurement Conference (IMC)*.

[24] Julien Dusser, Thomas Piquet, and André Seznec. 2009. Zero-content augmented caches. In *Proc. of the Intl. Conf. on Supercomputing (ICS'09)*.

[25] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *Proc. VLDB Endow.* 13, 2 (2019).

[26] Dmitry Evtyushkin and Dmitry Ponomarev. 2016. Covert Channels Through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations. In *Proc. of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*.

[27] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proc. of the 23rd intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIII)*.

[28] Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. 2017. IRON: Functional Encryption Using Intel SGX. In *Proc. of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*.

[29] Anders Fogh and Christopher Ertl. 2018. Wrangling with the Ghost: Aninside story of mitigating speculative execution side channel vulnerabilities. https://i.blackhat.com/us-18/Thu-August-9/us-18-Fogh-Ertl-Wrangling-with-the-Ghost-An-Inside-Story-of-Mitigating-Speculative-Execution-Side-Channel-Vulnerabilities.pdf

[30] Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Austin Harris, Zhixing Xu, Baris Kasikci, Valeria Bertacco, Sharad Malik, Mohit Tiwari, and Todd Austin. 2019. Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn. In *Proc. of the 24th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*.

[31] Jayesh Gaur, Alaa R Alameldeen, and Sreenivas Subramoney. 2016. Base-victim compression: An opportunistic cache compression architecture. In *Proc. of the 43rd annual Intl. Symp. on Computer Architecture (ISCA-43)*.

[32] Daniel Genkin, Luke Valenta, and Yuval Yarom. 2017. May the Fourth Be With You: A Microarchitectural Side Channel Attack on Several Real-World Applications of Curve25519. In *Proc. of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*.

[33] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *Proc. of the 27th USENIX Security Symposium (USENIX Security 18)*.

[34] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. 2009. Side-Channel Analysis of Cryptographic Software via Early-Terminating Multiplications. In *Proc. of the 2009 Intl. Conf. on Information Security and Cryptology (ICISC'09)*.

[35] Daniel Gruss, David Bidner, and Stefan Mangard. 2015. Practical memory deduplication attacks in sandboxed javascript. In *Proc. of the 20th European Symposium on Research in Computer Security (ESORICS)*.

[36] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-Resolution Side Channels for Untrusted Operating Systems. In *Proc. of the 2017 USENIX Annual Technical Conference (USENIX ATC 17)*.

[37] Erik G Hallnor and Steven K Reinhardt. 2005. A unified compressed memory hierarchy. In *Proc. of the 11th IEEE intl. symp. on High Performance Computer Architecture (HPCA-11)*.

[38] Gernot Heiser, Gerwin Klein, and Toby Murray. 2019. Can We Prove Time Protection?. In *Proc. of the Workshop on Hot Topics in Operating Systems (HotOS).*

[39] Andrew Hilton, Neeraj Eswaran, and Amir Roth. 2009. FIESTA: A sample-balanced multi-program workload methodology. *Proc. MoBS* (2009).

[40] Raghavendra Kanakagiri, Biswabandan Panda, and Madhu Mutyam. 2017. Mbzip: Multiblock data compression. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 4 (2017).

[41] Jungrae Kim, Michael Sullivan, Esha Choukse, and Mattan Erez. 2016. Bit-plane compression: Transforming data for better compression in many-core architectures. In *Proc. of the 43rd annual Intl. Symp. on Computer Architecture (ISCA-43).*

[42] Seikwon Kim, Seonyoung Lee, Taehoon Kim, and Jaehyuk Huh. 2017. Transparent dual memory compression architecture. In *Proc. of the 26th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT-26).*

[43] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A defense against cache timing attacks in speculative execution processors. In *Proc. of the 51st annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-51).*

[44] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Proc. of the 2019 IEEE Symposium on Security and Privacy (SP).*

[45] Donghyuk Lee, Mike O'Connor, and Niladrish Chatterjee. 2018. Reducing Data Transfer Energy by Exploiting Similarity within a Data Transaction. In *Proc. of the 24th IEEE intl. symp. on High Performance Computer Architecture (HPCA-24).*

[46] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proc. of the 27th USENIX Security Symposium (USENIX Security 18).*

[47] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. 2016. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *Proc. of the 22nd IEEE intl. symp. on High Performance Computer Architecture (HPCA-22).*

[48] Fangfei Liu and Ruby B Lee. 2014. Random fill cache architecture. In *Proc. of the 47th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-47).*

[49] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI).*

[50] Ross McIlroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. In *arXiv preprint arXiv:1902.05178.*

[51] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. 2015. Doppelgänger: a cache for approximate computing. In *Proc. of the 48th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-48).*

[52] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. 2018. Oblix: An Efficient Oblivious Search Index. In *Proc. of the 2018 IEEE Symposium on Security and Privacy (SP).*

[53] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2019. MemJam: A False Dependency Attack against Constant-Time Crypto Implementations. *International Journal of Parallel Programming* 47, 4 (2019).

[54] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2005. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *Proc. of the 2005 Intl. Conf. on Information Security and Cryptology (ICISC).*

[55] Tri M Nguyen and David Wentzlaff. 2015. MORC: A manycore-oriented compressed cache. In *Proc. of the 48th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-48).*

[56] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *Proc. of the 25th USENIX Security Symposium (USENIX Security 16).*

[57] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Proc. of the 7th Cryptographers' Track at the RSA Conference on Topics in Cryptology.*

[58] Rodney Owens and Weichao Wang. 2011. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In *Proc. of the 30th IEEE International Performance Computing and Communications Conference (IPCCC).*

[59] David J Palframan, Nam Sung Kim, and Mikko H Lipasti. 2015. COP: To compress and protect main memory. In *Proc. of the 42nd annual Intl. Symp. on Computer Architecture (ISCA-42).*

[60] Biswabandan Panda and André Seznec. 2016. Dictionary sharing: An efficient cache compression scheme for compressed caches. In *Proc. of the 49th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-49).*

[61] Gennady Pekhimenko, Evgeny Bolotin, Nandita Vijaykumar, Onur Mutlu, Todd C Mowry, and Stephen W Keckler. 2016. A case for toggle-aware compression for GPU systems. In *Proc. of the 22nd IEEE intl. symp. on High Performance Computer Architecture (HPCA-22).*

[62] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. 2012. Base-delta-immediate compression: Practical data compression for on-chip caches. In *Proc. of the 21st Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT-21).*

[63] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *Proc. of the 25th USENIX Security Symposium (USENIX Security 16).*

[64] Moinuddin K. Qureshi. 2018. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *Proc. of the 51st annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-51).*

[65] Moinuddin K. Qureshi, David Thompson, and Yale N. Patt. 2005. The V-Way cache: Demand based associativity via global replacement. In *Proc. of the 32nd annual Intl. Symp. on Computer Architecture (ISCA-32).*

[66] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *Proc. of the 24th USENIX Security Symposium (USENIX Security 15).*

[67] Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In *Proc. of the 38th annual Intl. Symp. on Computer Architecture (ISCA-38).*

[68] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proc. of the 40th annual Intl. Symp. on Computer Architecture (ISCA-40).*

[69] Somayeh Sardashti, André Seznec, and David A Wood. 2014. Skewed compressed caches. In *Proc. of the 47th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-47).*

[70] Somayeh Sardashti and David A Wood. 2013. Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching. In *Proc. of the 46th annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-46).*

[71] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. 2018. ZeroTrace: Oblivious Memory Primitives from Intel SGX. In *Proc. of the 25th Annual Network and Distributed System Security Symposium (NDSS-25).*

[72] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proc. of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19).*

[73] Claude Elwood Shannon. 1948. A mathematical theory of communication. *Bell system technical journal* 27, 3 (1948).

[74] Fahad Shaon, Murat Kantarcioglu, Zhiqiang Lin, and Latifur Khan. 2017. SGX-BigMatrix: A Practical Encrypted Data Analytic Framework With Trusted Processors. In *Proc. of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*.

[75] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. 2011. Memory deduplication as a threat to the guest OS. In *Proc. of the Fourth European Workshop on System Security*.

[76] Yingying Tian, Samira M Khan, Daniel A Jiménez, and Gabriel H Loh. 2016. Last-level cache deduplication. In *Proc. of the Intl. Conf. on Supercomputing (ICS'16)*.

[77] Shruti Tople and Prateek Saxena. 2017. On the Trade-Offs in Oblivious Execution Techniques. In *Proc. of the Intl. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*.

[78] R Brett Tremaine, Peter A Franaszek, John T Robinson, Charles O Schulz, T Basil Smith, Michael E Wazlowski, and P Maurice Bland. 2001. IBM memory expansion technology (MXT). *IBM Journal of Research and Development* (2001).

[79] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. 2017. Jenga: Software-defined cache hierarchies. In *Proc. of the 44th annual Intl. Symp. on Computer Architecture (ISCA-44)*.

[80] Po-An Tsai, Yee Ling Gan, and Daniel Sanchez. 2018. Rethinking the memory hierarchy for modern languages. In *Proc. of the 51st annual IEEE/ACM intl. symp. on Microarchitecture (MICRO-51)*.

[81] Po-An Tsai and Daniel Sanchez. 2019. Compress objects, not cache lines: An object-based compressed memory hierarchy. In *Proc. of the 24th intl. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*.

[82] Common Vulnerabilities and Exposures. 2012. CVE-2012-4929. Available from MITRE, CVE-ID CVE-2012-4929. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2012-4929

[83] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *Proc. of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*.

[84] Zhenghong Wang and Ruby B. Lee. 2007. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proc. of the 34th annual Intl. Symp. on Computer Architecture (ISCA-34)*.

[85] Y. Xu, W. Cui, and M. Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proc. of the 2015 IEEE Symposium on Security and Privacy (SP)*.

[86] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. 2017. Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks. In *Proc. of the 44th annual Intl. Symp. on Computer Architecture (ISCA-44)*.

[87] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. 2019. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *Proc. of the 2019 IEEE Symposium on Security and Privacy (SP)*.

[88] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proc. of the 23rd USENIX Security Symposium (USENIX Security 14)*.

[89] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: A timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering* (2017).

[90] Vinson Young, Prashant J Nair, and Moinuddin K Qureshi. 2017. DICE: Compressing DRAM caches for bandwidth and capacity. In *Proc. of the 44th annual Intl. Symp. on Computer Architecture (ISCA-44)*.

[91] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. 2019. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. In *Proc. of the 26th Annual Network and Distributed System Security Symposium (NDSS-26)*.

[92] Qi Zeng, Rakesh Jha, Shigang Chen, and Jih-Kwon Peir. 2018. Data Locality Exploitation in Cache Compression. In *Proc. of the 24th Intl. Conf. on Parallel and Distributed Systems (ICPADS-24)*.

[93] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *Proc. of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*.