ZipChannel: Cache Side-Channel Vulnerabilities in Compression Algorithms

Marina Minkin
University of Michigan

Baris Kasikci
University of Washington

***Abstract*—While cache side-channel attacks have been known for over a decade, attacks and defenses have been mostly limited to cryptographic algorithms. In this work, we analyze the security of compression algorithms and their susceptibility to cache side-channels. We design TaintChannel, a tool that automatically detects cache side-channel vulnerabilities and apply the tool to compression software to conduct a study of vulnerabilities in popular compression algorithms—LZ77, LZ78, BWT—and their mainstream implementations. We discover that the implementation of all of these algorithms leak some or all of their input data via cache side-channels. This is concerning, as compression algorithms are widely used in software that operates on sensitive data (e.g., HTTPS).**

We demonstrate the practicality of these vulnerabilities via two end-to-end attacks on Bzip2. These attacks work in two different threat models and use different attack techniques. Our first attack targets compression within an SGX enclave using the Prime+Probe cache attack technique and extracts the entire input while it is being compressed with an accuracy greater than 99%. Because existing cache attack techniques fall short in targeting applications with larger memory footprint such as compression software, we develop new attack techniques for larger buffers. Our second attack works in the threat model when one application attacks a different application. It allows the attacker to identify which file is being compressed out of multiple options.

I. Introduction

Side-channel attacks extract secrets by analyzing the side effects of application execution rather than on their nominal output. This type of attack has been proven to be devastating for cryptographic algorithms that were not designed with side-channels in mind, with prior work having successfully leaked secret encryption keys from secure implementations [1, 2].

Encryption is not the only important target for side-channel attacks, as it is common practice in many applications, such as HTTPS, password-protected zip, and PDF files, to compress sensitive data before encrypting it. For example, at the time of writing 88% of all websites compress their content [3] and 81.4% encrypt it [4]. In such applications, protecting encryption from side-channel leakage is not sufficient if the compression that occurs shortly before encryption leaks information about the plaintext.

Previous side-channel attacks on compression only target implementations based on the LZ77 [5] algorithm and assume that the attacker can trigger compression multiple times with partially controlled input. CRIME and BREACH [6] measure the compressed data size to leak secret HTTP cookies, and Schwarzl et al. [7] measure the execution time of various compression utilities to leak parts of their input. Since prior work measures the overall execution of compression software rather than taking advantage of more granular cache side channels, they have to resort to controlling the input to leak data. We show that using cache side channels provides additional information that lifts this restriction.

Cache side-channel attacks use a variety of techniques to extract information about the application's usage of the cache, with the most common ones being Prime+Probe [1] and Flush+Reload [2]. Although these techniques allow attackers to manipulate the cache's state and measure the timing of cache accesses to leak secrets, they were not designed to attack applications with a large memory footprint, such as compression software, where multiple memory accesses map to the same cache set. In this paper, we are the first to develop techniques to perform fine-grained cache attacks (i.e., those that monitor individual cache lines) on applications whose memory footprint exceeds the cache's size.

Although numerous tools detect cache side-channel vulnerabilities in software, they have some limitations. Some existing techniques [8, 9, 10] rely on exhaustive path exploration (e.g., via symbolic execution). Alas, these techniques scale poorly and are limited to leaking secrets from applications with simple control flow, such as cryptographic implementations. Other techniques [11, 12, 13, 14, 15, 16] analyze the execution traces of applications to correlate cache accesses with program inputs to discover potential cache side-channel vulnerabilities. However, such techniques cannot compute the precise relation between inputs and secret-dependent memory accesses, which is crucial to mount reliable side-channel attacks.

To that end, in this paper, we follow a two-pronged strategy to (1) automatically detect cache side-channel vulnerabilities in compression software and (2) demonstrate the viability of attacking such software via end-to-end attacks.

TaintChannel. We first present TaintChannel, a lightweight taint-tracking-based tool to automatically detect cache side-channel leakage. Unlike symbolic execution-based tools [8, 9, 10] that face scalability issues due to path explosion, TaintChannel analyzes a concrete execution path to detect cache side-channels, making it applicable to complex systems like compression software. Unlike trace-based tools [11, 12, 13, 14, 15, 16], TaintChannel computes the exact function that relates inputs to the memory addresses accessed during compression, which is required in cache side-channel attacks that analyze cache traces at a fine granularity.

2158-3927/24/$31.00 ©2024 IEEE
DOI 10.1109/DSN58291.2024.00033

Using TaintChannel, we perform an in-depth study of all the major compression algorithms and their signature implementations. Our analysis reveals side-channel vulnerabilities that can leak information about the data that these algorithms process during compression. More specifically, we studied a common implementation of each of the major algorithms and discovered leakage gadgets in all of them: LZ77 [5] used in Gzip [17], LZ78 [18] used in NCompress [19], and BWT [20] used in Bzip2 [21].

End-to-end attacks on Bzip2. Based on the leakage gadgets we found, we present ZipChannel, a new class of compression cache side-channel attacks that we demonstrate with two end-to-end attacks on Bzip2: Our first ZipChannel attack uses a version of Prime+Probe cache attack technique [1] that we enhance with a novel application of Intel CAT and a novel page selection technique in order to handle the large footprint of the histogram that Bzip2 constructs. The attack exploits a data flow gadget to leak a buffer with random data while an Intel SGX enclave compresses it. In our evaluation, ZipChannel achieves over 99% accuracy when leaking randomly-generated data. Our second attack uses the Flush+Reload attack technique with a control flow gadget to fingerprint different files while Bzip2 compresses them.

Improving cache side-channel techniques. We find that current Prime+Probe techniques do not perform well for our first attack. The main obstacle we face with Prime+Probe is that compression algorithms tend to have a significantly larger memory footprint, causing existing approaches to fall short because the victim application accesses multiple addresses that map to the same cache sets. We note that Shusterman et al. [22] even suggest introducing additional memory accesses to mitigate their cache attack.

Furthermore, unlike attacks that target cryptographic keys, (e.g., RSA keys), which leverage the redundancy in those algorithms to only extract a part of the key and reconstruct the rest based on the mathematical properties of the key, we can not rely on such techniques to recover arbitrary compressed data and require a more reliable channel.

To overcome these challenges and achieve a more reliable channel, ZipChannel leverages the Intel *Cache Allocation Technology* (CAT) to improve our cache side-channel attack's accuracy (for the first time to our knowledge). Additionally, ZipChannel introduces a novel frame selection technique that allows the attacker to select optimal frames for the attack.

The vulnerabilities we present in this work are particularly concerning due to the widespread nature of compression algorithms: security vulnerabilities in compression can have far-reaching consequences. Past attacks on compression algorithms have shown that an attacker can steal authentication cookies from an HTTPS session, leak sensitive content, or execute arbitrary code [6, 23]. Despite being known for over a decade, disabling compression to mitigate these attacks is the only known complete defense [6].

Overall, we make the following contributions.

1) We develop TaintChannel, a tool that helps developers detect cache-side channel vulnerabilities in binaries.
2) We perform a study to demonstrate that all major compression algorithms have gadgets performing memory accesses that depend on the entire compressed file.
3) We demonstrate two end-to-end attacks against Bzip2.
4) We substantially improve cache attack techniques on Intel SGX and achieve higher accuracy than prior work:
   a) ZipChannel is the first attack to extract a fine-grained cache access pattern from a buffer larger than a page using cache-side channels.
   b) ZipChannel is the first attack to utilize Intel CAT as an offensive technique.
   c) We develop a frame selection technique that improves attack accuracy.
   d) We implement a single-stepping mechanism from user space, making the attack easier to mount.

The rest of the paper is organized as follows. Section II provides background. Section III introduces TaintChannel, our automated tool to find cache side-channel leakages. Section IV surveys popular compression algorithms and the vulnerabilities found by TaintChannel, Section V is an end-to-end attack on Bzip2 compression within SGX, Section VI is our end-to-end attack on fingerprinting files compressed with Bzip2, Section VII is related work. In Section VIII we discuss potential mitigations against our attack. We conclude in Section IX.

Open Source. Artifacts for this paper can be found here https://github.com/efeslab/ZipChannel.

Responsible disclosure. Following the responsible disclosure guidelines, we disclosed this work to Intel.

II. BACKGROUND

In this section we provide background on compression software and cache side-channel attacks.

A. Compression Software

Compression algorithms are prevalent, as they reduce the data size, allowing for more data to be stored or transferred for the same capacity and bandwidth, respectively. All the general-purpose compression software that we could find employs one of three fundamental algorithms: *LZ77* [5], *LZ78* [18], and *Burrows-Wheeler Transform* (BWT) [20]. **LZ77**, also known as LZ1 and *sliding window compression*, replaces repeating values with a reference to a previous occurrence of the same value. The DEFLATE compression format, used by Gzip [17], Zlib [24], Zip [25], and PNG [26] (e.g. used in eponymous utilities) and the Brotli format [27], the successor of Gzip for network traffic compression, both use the LZ77 algorithm. At the time of this writing, 87% of the total Internet traffic is compressed using Gzip, Brotli, or DEFLATE [3]. **LZ78**, known as LZ2 and used by the Linux utility `compress` [28], stores input strings in a dictionary that maps them into *codes* and outputs a stream of codes. While it compresses faster than most compression utilities, due to originally being protected by a patent [29], it is not as commonly used as the other utilities. **BWT**, used in Bzip2 [21], rearranges the input into sequences with more repetitions to make compression efficient.

B. Cache Side-Channel attacks

CPU caches act as a buffer between the main memory and the processor to improve the memory access latency by storing frequently accessed data for quick access. To improve their utilization, caches are usually shared among different applications. As a result, the time it takes for an application to access its own memory may be affected by the memory access pattern of other applications.

Cache side-channel attacks use the variability in memory access latency to leak secrets across the application boundary. Specifically, an attacker application can measure how long it takes to access its own memory to leak secrets. Some of the most famous cache side-channel attacks are Prime+Probe [1] and Flush+Reload [2].

Prime+Probe. This cache attack consists of three steps: 1) The attacker application reads their own data to populate the cache. 2) the victim application executes, and as a side-effect, might evict the attacker's data from the cache. 3) the attacker measures how long it takes to access their own data. At the end of the attack, the attacker knows if the victim accessed addresses evicting the attacker's data, and can use this information to infer secrets from the victim.

Flush+Reload. To mount the attack, the attacker uses the CPU flush instruction to evict a cache line that is shared between the attacker and the victim. Next, the attacker waits for the victim application to execute. If the victim accesses the memory location backing the evicted cache line, the CPU fetches it into the cache. For the final step of this attack, the attacker accesses the cache line and measures the access latency; should it be short, then this indicates that the victim has accessed the monitor address. With this information, attackers can leak sensitive data from their victims' applications.

III. A Tool to Detect Side-Channel Leakage

A. Design of TaintChannel

In this section, we present TaintChannel, a novel tool for detecting cache side-channel vulnerabilities. We named our tool this way because it relies on taint tracking of the input and displays the results in a human-readable way to detect cache side-channels. In a nutshell, taint tracking [30, 31] marks some input data as "tainted" and traces its flow through a program. In our case, this "taint" originates from the input file to the program. TaintChannel then monitors the movement of the tainted data, remembering the movement history for each input byte. Finally, TaintChannel then reports on the potential vulnerabilities it discovers.

Because TaintChannel analyzes a concrete execution path, it scales to applications, such as compression, that are more complex than symbolic execution can handle [8, 9, 10]. Additionally, TaintChannel outputs the exact computation that converts the input into a dereferenced pointer, which is impossible with fuzzing-based tools [11, 12, 13, 14, 15, 16]. Although some existing tools implement taint tracking functionality, such as CacheD [8] to narrow the scope of symbolic execution analysis and CaSym [9] to represent array data, TaintChannel is the first tool for discovering side channel vulnerabilities where taint tracking is the core technique and not just an optimization.

A common pitfall with taint tracking is that it is prone to over-estimating the taint. To mitigate this, while still discovering data-dependent memory accesses, TaintChannel propagates taint for *direct* data manipulation, e.g., propagates taint affected by instructions and arithmetic operations (such as `x = y + z`) but not *indirect* data manipulations, i.e., taint propagated via control flow divergence. For example, for the code snippet `if(x<5) cnt++`, taint from `x` would not propagate into `cnt`. We show that this form of taint tacking is sufficient for TaintChannel to discover the gadget in the seminal attack on AES [1] and to be the first to discover specific cache side-channel leakage gadgets in compression software (rather than looking at the entire compression time [6] or compression ratio [7] while controlling parts of the input [6, 7]).

TaintChannel fulfills three main objectives to find vulnerabilities in compression software: (1) scalability for applications with large memory footprints and unbound loops (2) emitting the computation that converts secret inputs to memory access patterns. (3) the user should not have to instrument, modify, or recompile the source code. Thus, TaintChannel records the propagation of all input bits through the execution, to detect memory dereferences where the addresses depend on the input.

Scalability of the analysis. Symbolic execution-based tools [8, 9, 10] do not scale well, because instead of executing an application along a single path, these tools explore all possible execution paths, making the exploration time exponential in the size of the application. Although symbolic execution could theoretically provide the most comprehensive capability for detecting cache side-channels, exploring all possible execution paths does not scale to complex applications in practice. This is why symbolic execution-based tools usually sacrifice the coverage of symbolic execution to achieve reasonable performance. E.g., Daniel et al. [10] verifies applications that were designed to be constant time and CacheD [8] only performs symbolic execution along an execution trace. As we analyze compression software with TaintChannel, we see that most of vulnerabilities are in data-dependent array accesses. In these cases, symbolic execution is not helpful, because symbolic execution duplicates the state and address space for each possible array index value, exhausting system resources.

Automatically determining the relation between the secret input and the accessed addresses. In applications that are vulnerable to cache side-channels, the victim performs a memory access whose address depends on a secret input unknown to the attacker. The attacker, in turn, observes the accessed address to infer the secret. While knowing that such gadget exists indicates that an application is vulnerable, it is not sufficient for mounting an attack. It is also important to understand how exactly to compute the relevant dereferenced pointer given a secret input, because without this information, an attacker would not know how to interpret the pointer they leaked. Unfortunately, trace-based tools that run the application with different inputs and look for statistical correlation

between inputs and cache trace [11, 12, 13, 14, 15, 16] inherently cannot determine the exact relation between the input and the pointer. In contrast, TaintChannel outputs all instructions accessing the secret. Therefore, users can directly see how the accessed address was computed based on the input (i.e. the secret the attacker is trying to infer).

User interface. The user has to provide a command line to invoke the application to test, and does not need the source code. If debug information is available, TaintChannel displays it. TaintChannel creates a unique visualization for each leakage gadget with information it leaks as illustrated in Fig. 2, making it easy to identify how much information a gadget might leak.

B. Implementation

We implement TaintChannel as a DynamoRIO [32] tool in 1630 lines of code. DynamoRIO is an instrumentation framework that is portable and works on different architectures, i.e., IA-32, AMD64, ARM, and on different operating systems, i.e., Windows, Linux, and has experimental MacOS support. Although we only test TaintChannel on Intel CPUs in a Linux environment, we choose DynamoRIO over Intel Pin [33] due to its portability. In TaintChannel's code we use DynamoRIO abstractions avoid hard-coding Intel instructions, which should reduce the effort in porting TaintChannel to other platforms.

TaintChannel assigns a sequential index for each input byte, i.e., the first byte read with the system call `read` would be #1, the second would be #2 etc. At the end of the analysis, TaintChannel outputs the propagation of the taint per input byte that results in input-dependent dereferenced memory address. The output includes a report of the taint propagation from each input byte, starting when the value is first read, across all operations that directly use the value, e.g., copying and arithmetic operations. For each instruction that propagates taint, TaintChannel outputs its opcode, address, mnemonic, relevant operand values, and whether they are tainted. For the instruction that performs the final taint-dependent pointer dereference, TaintChannel additionally outputs ASCII art that illustrates which operand bits are tainted with what tag. Fig. 2 contains the entry of the taint-dependent memory dereference TaintChannel discovered in Zlib. It shows that the instruction writes 2 bytes from the register `ax` into the address pointed by `rdx`. Below that, we can also see in Fig. 2 the taint breakdown in `rdx`: bits 1-8 are tainted with data from input byte 5752, bits 6-12 with byte 5751 and bits 11-15 with byte 5750.

Taint propagation. For each instruction that the application executes, TaintChannel maintains the state of the taint per register and memory location, as well as a per-taint-tag history. To implement the taint propagation, TaintChannel groups instructions based on the type of their operands, e.g., operations with a memory source operand and a register destination operand, instructions with an immediate as a source operand and register as destination, etc. For instructions that include multiple sources, such as `xor` and `or`, TaintChannel simply merges the taint of the sources into the destination operand, as each bit can hold an arbitrary number of taint tags. For example if the register `rax` holds taint from input byte 5 in bits 0 and 1 and the register `rbx` holds taint from the input byte 6 in the bits 1 and 2, `xor rax, rbx` would have taint from input byte 5 in bit 0, taint from both input bytes 5 and 6 in bit 1 and taint from input byte 6 in bit 2. Some instructions e.g., `and` with a mask, or arithmetic shift, require special handling. If the application performs an `and` between a tainted value and an untainted value, the result of the operation would include the original tainted tags only at the locations where the untainted values were 1. An arithmetic shift shifts the taint the same number of bits as the instruction itself.

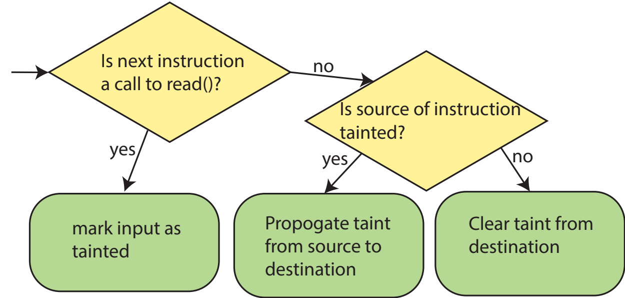

Fig. 1: A simplified decision tree that TaintChannel traverses for each executed instruction. TaintChannel first initiates the input data as tainted, and then propagates taint between source and destination operands of different instructions, using the DynamoRIO abstraction to identify the source and destination of each instruction.

Discovering data-flow vulnerabilities. TaintChannel discovered cache side-channel vulnerabilities in implementations of all major compression algorithms, as we analyze in Section IV. we also verified TaintChannel that TaintChannel finds the vulnerability Osvik et al. [1] in the software implementation of AES in OpenSSL.

Discovering control-flow vulnerabilities. TaintChannel effectively reduces a complex application to a small trace of input-dependent instructions. These traces simplify the comparison of the application execution across different inputs. This is how we discover control flow vulnerabilities. This is significant because the most reliable cache side-channel technique for the application-application threat model, Flush+Reload [2], is particularly effective in spying on code executed by shared libraries. We describe the vulnerability that we discovered in Bzip2 in Section VI.

While analyzing compression software, we discovered control-flow vulnerability in `memcpy()`. The vulnerability is that there are multiple control flow paths within `memcpy()` based on the size of the data being copied - if the size of the data is an exact multiple of the size of an AVX register, it uses these registers to implement the copy. Otherwise, `memcpy()` copies as much as it can using the AVX registers, and the rest byte by byte. This can reveal information about the exact size of data that is being copied. Especially if combined with the estimate of the total runtime of the function.

IV. Compression Algorithms and Their Vulnerability to Cache Side-Channels

In this section, we study the three most popular compression algorithms as outlined in Section II. We first provide background on how an attacker uses the cache as a side channel to leak secrets (Section IV-A). Using TaintChannel's output, we then show that all three major algorithms that underlie the most widely used compression utilities have cache side-channel leakage gadgets that can leak their entire input (Sections IV-B, IV-C, and IV-D). For two of these implementations, we show how an attacker can extract their entire input.

A. The Cache Channel and the Threat Model

Cache side-channels attacks can observe the cache line a victim accesses but not the offset within the cache line. Because the cache consists of *cache lines* of 64 bytes they cannot observe the 6 least significant bits of each address (6 least significant address bits are used to represent the offset within the cache line because $\log_2 64 = 6$). Therefore, when we describe the attack gadgets in the rest of this section, for each gadget we present we assume that when the attacker generates a trace of the memory accesses that are performed by the gadget in the victim application, the 6 least significant bits are not visible to the attacker(even though they appear in TaintChannel's output). In the rest of the section, we assume a threat model where an attacker knows the base addresses of all the arrays the victim accesses. When we describe our end-to-end attack in Section V, we show that these cache channel properties and the threat model are realistic.

B. LZ77

The LZ77 [5] algorithm, eliminates string repetitions in the plaintext by replacing them with backward references that include the distance to the previous occurrence of the same string, and the length of the repeated string. LZ77 is widespread among different compression formats, including Brotli [27] and DEFLATE [34]. While DEFLATE is a standalone format, it gains its popularity from being a part of other formats, e.g., Gzip [17], Zlib [24], Zip [25] and PNG [26]. DEFLATE and Gzip, among other use cases, are supported by all major browsers for encoding HTTP traffic [35].

We study DEFLATE's specification [34] as well as its implementation in the Zlib [24] library that is likely the most a used implementation of DEFLATE, Zlib and Gzip.

Implementation of the Zlib compressor. The role of an LZ77-based compressor is to find repetitions in the input stream and replace them with backward references to a previous occurrence. The references are represented by the distance between the previous occurrence of a string and the current one, and additionally by the length of the repeated string. The Zlib compressor, as it implements the DEFLATE protocol, has the freedom to choose which repetitions to eliminate, and which previous instance to point to. For this reason, implementers of the Zlib compressor had to find a balance between compression ratio, performance and memory footprint

```
1 #define UPDATE_HASH(ins_h , c)              \
2     ins_h = ((ins_h<< 5) ^ c) & 0x7fff)
3
4 #define INSERT_STRING(i , window , head) \
5     UPDATE_HASH(ins_h , window[i + 2]) , \
6     ...                                  \
7     // head is of type unsigned short* \
8     head[ins_h] = (Pos)(i))
```

Listing 1: Simplified version of the LZ77 compression gadget. Code adapted from from gadget on line 182 in `deflate.c` in Zlib 1.2.13

(e.g., finding a longer previous match might give a better compression, but take more resources to find).

Besides performance trade-offs, compressor implementations, including Zlib, avoid violating patents. Conveniently, the specification of DEFLATE [34] provides a recommendation for a compressor implementation that is not restricted by patents: "The compressor uses a chained hash table to find duplicated strings, using a hash function that operates on 3-byte sequences." We observe that a typical implementation that follows this recommendation would use an underlying array to implement the hash table and use hashes of triplets of bytes as array indices. Accessing the array using these hashes can leak information about the hashes via the cache side-channel, and consequently leak information about the input that was used to compute the hashes. To quantify this input leakage, we examine the Zlib compressor that follows the guideline [34].

Listing 1 is a simplified version of the code that populates the hash table `head`, where the array `window` contains the input to be compressed. The main code calls the macro `INSERT_STRING` with with sequential values of `i`, starting from 0 and the macro inserts strings of length 3 starting from the index `i` in `window` to the hash table `head`. Before a call to `INSERT_STRING`, `ins_h` is initialized to `(window[i] << 5) ^ window[i+1]`. When `INSERT_STRING` is invoked, it starts its operation by calling `UPDATE_HASH` (Line 1) on Line 5. Resulting in the variable `ins_h` always contains information about the three latest input bytes in a particular structure. Hence, when `head[ins_h]` is accessed it causes an input-dependent address dereference.

We use TaintChannel's output (Fig. 2) to illustrate the breakdown of the input bits within the pointer to `head[ins_h]`, the address being dereferenced in Line 8 from Listing 1, and stored in the register `rdx`. Due to pointer arithmetics, the value inside `rdx` is `head + ins_h<<1`. Fig. 2 illustrates that the taint in `rdx` in bits 1-8, 6-13 and 11-15 contains taint originating from 3 consecutive input bytes. By inspecting TaintChannel's output that is not included in Fig. 2 we can further see the computation that includes the xor of the 3 latest input bytes left-shifted by 1, 6 and 11 bits, respectively (we elaborate on this process in Section IV-C). In terms of the source code, every time `UPDATE_HASH` is called, it starts its operation by left-shifting the previous value of `ins_h` by 5 bits. Then `UPDATE_HASH` xor's `ins_h` with the new byte `c` and masks `ins_h` with `0x7fff` to leave the 15 bits that are

```
Taint-dependent memory access
0x00007f43da2ff954 /path/to/libz.so.1.2.11!deflate_slow+468
  0x00007f43da2ff954  66 89 02            data16 mov    %ax -> (%rdx)[2byte]
rdx = 10 b7 43 d6 43 7f 00 00 [  C C   ] (tainted)
        5750: | x| x| x| x| x|  |  |  |  |  |  |  |  |  |  |  |
        5751: |  |  | x| x| x| x| x| x| x| x|  |  |  |  |  |  |
        5752: |  |  |  |  |  |  |  | x| x| x| x| x| x| x| x|  |
              |15|14|13|12|11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
```

Fig. 2: A snippet of the output of TaintChannel running on Zlib showing the `mov` instruction that corresponds to the gadget in Listing 1. The instruction copies data from the register `ax` to the address in the tainted register `rdx`. The snippet shows *how* `rdx` is tainted. For example, bits 6-13 are tainted with information from input byte 5751. (TaintChannel also outputs instruction address, file, function, offset within the function.)

depicted in Fig. 2. Finally, on Line 8, the code accesses the array `head` at the index `ins_h`. Finally, because the type of `head` is `short unsigned int` (2 bytes), therefore, the compiler replaces the accesses to `head[ins_h]` on Line 8 with a dereference to the address `head + ins_h<<1`.

Recall that in our threat model, the attacker can observe all memory accesses modulo cache offset. For simplicity, assume that the array `head` is cache aligned. To extract information from the cache channel, at iteration `i`, an attacker subtracts the known address of `head` from the address they observe (`head + ins_h *2`). The attacker then divides the subtracted value by 2 to undo the pointer arithmetic that multiplies `ins_h` by 2 during array dereference, and gets a value that contains all the bits with indices greater than 5 from Fig. 2 because, as we discuss in Section IV-A, the cache channel does not leak the 6 least significant bits of addresses in a memory trace.

The attacker directly obtains the 2 middle bits that are not xor'ed with other values from `window[i+1]`, therefore they can recover 25% of the input plaintext data. For each input byte, the attacker also obtains the value of the 3 least significant bits xor'ed most significant bits of previous byte and xor of 3 most significant bits from the next byte. Additional knowledge about the plaintext can help the attacker recover further information. For example, in a file that composed of lower-case ASCII characters, all bytes would be in the range `0x61`-`0x7a` meaning that the 3 most significant bits would always be `011`, allowing leakage of the entire content of the file with minor losses due to cache misalignment, which we overcome with a detailed example for Bzip2 in Section IV-D.

C. *LZ78*

LZ78 [18] forms the foundation for the popular compression utility Ncompress [19], which all major browsers support for HTTP encoding [35] and which comes with major operating systems, such as MacOS and variants of UNIX (invoked using the command `compress`). LZ78 is a *dictionary-based* compression algorithm, where the compressor outputs a stream of *codes*. During compression, the compressor maintains a dictionary that maps codes (represented as integers) to *strings* of one or more characters from the uncompressed input. The compressor constructs the dictionary based on the currently and previously processed uncompressed input characters. The dictionary in LZ78 is designed such that that the decompressor can reconstruct it based on the previously decompressed data. (Which is helpful for data recovery in our attack.)

Ncompress is based on LZW [29], and not on a pure LZ78 implementation, which is not used in popularly available tools. The main modification that LZW introduces to LZ78 is the addition of pre-initialized dictionary entries. Rather than adding all characters and strings to the dictionary as the decompressor reads the input, the dictionary is pre-initialized with all possible characters. Specifically, in the Ncompress implementation, the dictionary is initialized to contain a mapping that maps all codes 0–256 to themselves. The values 0–255 represent the same value, and 256 represents EOF.

Fig. 3 contains output from TaintChannel operating on Ncompress and has information about the taint propagation of an input byte with a value of 0x20. The value is used to compute the address of a dereferenced pointer. In (1) in Fig. 3 this value is read into a memory address ending with `b49`. It is then copied to several registers: from memory to `eax`, from `al` (an alias for `eax`) to `bl`, and from `rax` to `rsi`. In step (3), the value is shifted left by 9 bits using the `shl` instruction. In step (3) It is xor'ed with the value in the register `rdx` in step (3), which we will show to contain a dictionary entry when we analyze the source code. Next, the value that is now shifted and xor'ed with a dictionary is copied into the register `r9`. Finally, the computed value is used as an index to an array whose base is in the register `rbp`, after being multiplied by 8 as indicated by the addressing mode in (4).

Listing 2, includes the code snippet from Ncompress that exhibits an input-dependent access pattern and corresponds with step (4) in Fig. 3 Analyzing the source code allows us to validate the output from TaintChannel and to quantify the leakage. The code snippet in Listing 2 is invoked repeatedly for every input in the order they appear. The newly read input byte is stored in the variable `c`. In each invocation of the snippet, the variable `ent` contains a dictionary entry that is computed based on the previous plaintext bytes in a deterministic method. Listing 2 does not show this computation. Instead, Listing 2 only shows the usage of the value of `ent`.

The code in Listing 2 performs the first step of a lookup into the hash table, `htab`, which contains information about whether a given entry appears in the dictionary. In particular, this lookup checks whether a certain value of `fc` is already in the hash table `htab` at the index `hp`. The value of `hp` is computed on Line 9 of Listing 2, and is a simple hash function computation that uses the input byte `c` and `ent` as inputs. `hp` is used for indexing the hash table `htab` (Line 11). Thus, causing a dereference to a pointer that depends on `hp`.

To correlate with Fig. 3, the value of `hp` is stored in `rax` from step 4, and the base of the array `htab` is stored in `rbp`. Looking at the addressing mode of the instruction, the accessed address is computed as `rbp + rax*8`. The reason for multiplying the array index by 8 is evident in the source code in Listing 2, where the type of `htab` is an array of `count_int` that is defined as `unsigned long` - an 8-byte

integer. Finally, bits 9-16 in Fig. 3 are marked as tainted with data from input byte 1001, which matches to the variable `hp` containing data from input byte `c` shifted left by 9 bits.

Because in the implementation we study, the address `htab` is always cache line aligned, the attacker can subtract the address of `htab` from the leaked address to retrieve the value of `hp`. Initially, the attacker can shift the result to the right by 3 to obtain the value of `hp` because `ent` is variable length and grows from 9 to 16 bits as the dictionary grows. At this point, the attacker has all bits except bits 0-2 from Fig. 3.

As `ent` grows beyond 9 bits, the attacker has to get more sophisticated and use the key observation that because the compression algorithm is designed to be reversible, knowledge of all previous input bytes allows the attacker to compute all dictionary entries in the same manner as the compressor does. In particular, the attacker can xor the variable `ent` they compute with the observed value of `hp` to gain each input byte `c`. However, there is a special case for the first input byte; in the first invocation of the snippet of Listing 2, `c` is already initialized with the second input byte while `ent` contains the first input byte (not shown in the snippet). This allows the attacker to leak 5 bits out of the first input byte (since the last 3 bits are lost as mentioned above). With the assumption the attacker knows these 3 bits from the first byte, they can fully recover the input. If we relax this assumption, the attacker can check all $2^3 = 8$ possible triplets of bits and choose the most feasible input out of these options. We verified that this attack is possible with a Python script that simulates the attack, given a trace of memory accesses performed by Line 11.

D. BWT

Burrows–Wheeler transform (BWT) [20] batches together similar input bytes. BWT is useful for compression because it is easier to compress a string with multiple repeating characters [20]. BWT is reversible, which allows decompression.

BWT forms the basis of the compression utility Bzip2 [21] that we target because of its popularity as it comes pre-installed with most Linux-based distributions (e.g., Android and Ubuntu) and MacOS. In this section, we analyze its implementation and how it can lead to a side-channel vulnerability. Bzip2 is built as a stack of multiple algorithms, each of which is reversible, which allows decompression of a compressed file. The first step in the Bzip2 compression (and the last in decompression) is run-length-encoding (RLE), which compresses sequences of the same byte in the plaintext input. Because RLE does not affect most inputs, and because we attack BWT (the step that follows RLE), in the rest of this paper, we refer to the data compressed with RLE as the input.

```
1  long hp;
2  char_type     c;
3  unsigned short   ent;
4  long fc;
5  typedef long int        count_int;
6  count_int     htab[HSIZE];
7  ...
8
9  hp = ((((long)c) << 9) ^ (long)ent);
10 // fc = (ent << 8) | c
11 if ((htab[hp]) == fc) goto hfound;
```

Listing 2: A simplified code gadget from (N)compress 5.1 to leak data during compression. Adapted from the file `compress.c:1176`

```
taint source
data (1 bytes from 0x00007fb86b3fcb49): 20      [ ]      (1)
==============================
0x00007fb86b2b4c7a /path/to/ncompress/compress!compress+826
  0x00007fb86b2b4c7a  43 0f b6 44 25 00    movzx  0x00(%r13,%r12)[1byte] -> %eax
data (1 bytes from 0x00007fb86b3fcb49): 20      [ ]
==============================
0x00007fb86b2b4c83 /path/to/ncompress/compress!compress+835
  0x00007fb86b2b4c83  88 c3                mov    %al -> %bl
rax = 20 00 00 00 00 00 00 00 [        ] (tainted)
==============================
0x00007fb86b2b4c85 /path/to/ncompress/compress!compress+837
  0x00007fb86b2b4c85  48 89 c6             mov    %rax -> %rsi
rax = 20 00 00 00 00 00 00 00 [        ] (tainted)
==============================                                (2)
0x00007fb86b2b4c88 /path/to/ncompress/compress!compress+840
  0x00007fb86b2b4c88  48 c1 e0 09          shl    $0x0000000000000009 %rax -> %rax
rax = 20 00 00 00 00 00 00 00 [        ] (tainted)
==============================
0x00007fb86b2b4c8c /path/to/ncompress/compress!compress+844    (3)
  0x00007fb86b2b4c8c  48 31 d0             xor    %rdx %rax -> %rax
rax = 00 40 00 00 00 00 00 00 [ @      ] (tainted)
rdx = 0a 01 00 00 00 00 00 00 [        ]
==============================
0x00007fb86b2b4c8f /path/to/ncompress/compress!compress+847
  0x00007fb86b2b4c8f  49 89 d9             mov    %rbx -> %r9
rbx = 20 00 0a 01 00 00 00 00 [        ] (tainted)
==============================
Taint-dependent memory access                                 (4)
0x00007fb86b2b4c92 /path/to/ncompress/compress!compress+850
  0x00007fb86b2b4c92  48 8b 4c c5 00       mov    0x00(%rbp,%rax,8)[8byte] -> %rcx
rax = 0a 41 00 00 00 00 00 00 [ A      ] (tainted)
          1001: | x| x| x| x| x| x| x| x|  |  |  |  |  |  |  |  |
                |16|15|14|13|12|11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
rbp = c0 a1 2f 6b b8 7f 00 00 [  /k    ]
```

Fig. 3: The propagation of the taint for an input byte is shown in Listing 2. The different stages in the propagation of the taint are divided using rows filled with the symbol =. The snippet follows the value that is read from the input (1), copied across different registers until it is shifted by 9 bits (2), xor'ed with a dictionary value (3), and eventually used for a memory dereference (4). In the last instruction, we can see that the bits 9-16 in the array index are tainted and that the array index is multiplied by 8 before the dereference.

BWT starts its operation by sorting all cyclical shifts of the input string in lexicographic order. For example, the first step of applying BWT on the string `BANANA`, sorts the strings `BANANA`, `ABANAN`, ... `ANANAB`. To implement that, Bzip2 starts its execution with the creation of a histogram that counts the numbers of all pairs of bytes that appear in the input. Listing 3 is a simplified version of the histogram construction code. The histogram—referred to as a frequency table in the code—is represented by the array `ftab`. This array has 65537 entries, where indices represent all the values that the byte-pairs can take (2^{8+8}). Each element of `ftab` represents the number of times this pair of bytes appears in the input.

Bzip2 divides the input into blocks for processing. The variable `nblock` (Line 7), contains the size of the `block` being compressed. Elements of the `quadrant` array are set to zero in (Line 8), as a cache-related performance optimization as per the comments in the original source code. As we demonstrate later, these writes to the `quadrant` array make

```
1  UInt32* ftab = malloc(65537 * sizeof(UInt32));
2
3  /*-- set up the 2-byte frequency table --*/
4  for (i = 65536; i >= 0; i--) ftab[i] = 0;
5
6  j = block[0] << 8;
7  for (i = nblock - 1; i >= 0; i--) {
8     quadrant[i] = 0;
9     j = (j >> 8) | ( ((UInt16)block[i]) << 8);
10    ftab[j]++;
11 }
```

Listing 3: construction for frequency tabels in bzip2 source code. A simplified version of the gadget in the function `mainSort` in the file `blocksort.c`, line 769. bzip2-1.0.6

our attack more reliable (Section V). Therefore, even though other parts of the Bzip2 code also perform memory accesses dependent on the entire input, the accesses to the `quadrant` array make this particular snippet a more effective gadget.

To construct the frequency table, the code in Listing 3 iterates over `block` bytes in reverse order using the iterator `i`. `j` is initialized in Line 6 to the value of `block[0]` shifted 8 bits to the left and is used as index into `ftab`. Within the `for` loop starting at Line 7, `j` holds the value of a concatenated pair of input bytes at a time. Looking at the first iteration, in Line 9, `j` is shifted 8 bits to the right to contain the value of `block[0]`, which is concatenated with the value of the last byte in the block, which resides in the higher 8 bits of `j`. On Line 10, `ftab[j]` is incremented to keep track of the count of j, which represents the current pair of bytes.

In each iteration, for a given `i`, the value of `j` on Line 10 equals `(block[i + 1] << 8) | block[i]` (where the first iteration wraps around the beginning of the array). To illustrate, the bytes of `j` as a 2-element byte array in the fist few assignments to `j` (Line 9) are the pairs of `block` entries at the indices: `nblock-1` and `0`, `nblock-2` and `nblock-1`, `nblock-3` and `nblock-2`, etc.

Fig. 4, contains 2 entries from TaintChannel's output that correspond to Line 10. `j` from Listing 3 is in the register `rcx`: in the first entry in Fig. 4 the bottom 8 bits are tainted with data from input byte 1689 and the upper 8 bits are tainted with data from input byte 1688. The second entry contains data from bytes 1690 and 1689 respectively.

Because `ftab`'s type is `Int32*` (Line 1), accessing `ftab[j]` is translated into accessing the memory at the address `ftab + j*4`, equivalent to `ftab + block[i]<<10 + block[i+1]<<2`. In most cases, an attacker who knows all but 6 least significant bits of the address can unambiguously compute `block[i]`. However, because the array `ftab` is not aligned to the size of a cache line, there can be an off-by-one ambiguity in the value of `block[i]`. One example where an attacker would see unresolved ambiguity that emerges from `ftab` not being cache aligned is when `ftab = 0x7ffff47177b0`. and the attacker observes an access to `0x7ffff472eb80` from the cache channel on iteration `i+1` (6 least significant bits are zeroed), the attacker would be able to infer the plaintext values in two ways: either the value of `block[i]` is between `0x00` and `0x03` and `block[i+1] = 0x5d`. or `block[i]` is between `0xf4` and `0xff` and `block[i+1] = 0x5c` the attacker can resolve this ambiguity by looking at the values of because We note that even if there is an off-by-one ambiguity on iteration `i`, the attacker can still know if `block[i]` is between `0x00` and `0x03` or between `0xf4` and `0xff`.

E. Survey Summary

In this section, we have shown how the three major algorithms used for compression—LZ77, LZ78, and BWT—are vulnerable to cache side channel attacks by identifying and studying gadgets to mount cache side channel attacks in all of them. For 2 out of 3 implementations, we find that an attacker who can observe all memory accesses at a cache line granularity can extract the entire plaintext. We show an end-to-end attack on BWT in Section V.

V. Attacking Bzip2 within SGX

In this section, we present an end-to-end attack on Bzip2 compression running inside an SGX enclave. We use the leakage gadget presented in Section IV-D. This gadget can theoretically leak the entire input, and we show that we can extract more than 99% of any randomly-generated input.

Intel Software Guard Extensions (SGX) is a set of instructions built into certain Intel CPUs that create "enclaves," protected regions of memory where code and data are encrypted while in use. This hardware-based isolation shields sensitive information from unauthorized access, even from the operating system or other applications running on the same machine.

In section IV-D, we introduced an attack gadget that exploits the frequency table `ftab` accessed on line 10 in Listing 3. However, existing attack methods are insufficient for this scenario due to the large size of `ftab` (64Kb) and the requirement for cache line granularity in accessing memory. To overcome these obstacles, we developed a new attack that combines novel and established techniques.

We implement a state machine to track the execution of Lines 8 to 10 from Listing 3 as required (these lines appear in Fig. 5 as well). The rest is organized as follows: Section V-A describes the technique we use to transition between different states to effectively single-step the victim. In Sections V-B

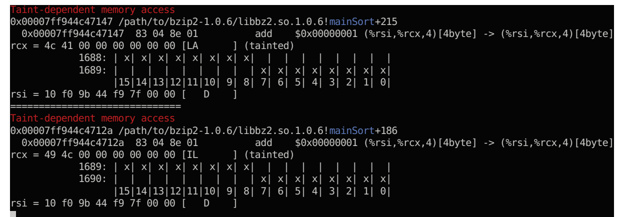

Fig. 4: Two entries from TaintChannel's trace for input byte 1689. In the top entry, the byte is in bits 0-7 of the array index. In the second entry it is in bits 8-15.

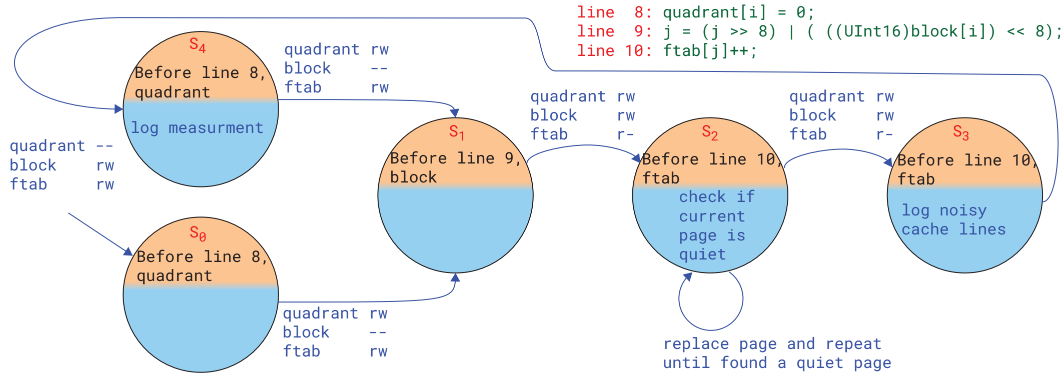

Fig. 5: State machine created by the controlled channel attack in ZipChannel. We include code from Listing 3 as reference.

and V-C we show how to leak the accessed page address and offset, respectively. Sections V-C1 and V-C2 introduce our novel improvements to the accuracy, and Section V-D describes how to aggregate the reading to retrieve the data.

A. Single-Stepping

To improve Prime+Probe precision and separate individual accesses to `ftab`, we single-step the enclave's execution. Previous methods rely on timer interrupts [36] for that, but we found these interrupts to be unreliable. Instead, we use a controlled-channel attack similar to Xu et al. [37]'s work. Instead of modifying the OS, we use the `mprotect` system call, which modifies the access permissions for a memory region. In our attack, we revoke access to the arrays `quadrant`, `block`, and `ftab`, one at a time. Since each of the lines 8, 9, and 10 from Listing 3 accesses exactly one of these, we effectively single-step through the attack gadget.

Fig. 5 illustrates our attack as a state machine, where the arrows are labeled with the required permissions to go to the next state. To illustrate a few of the transitions: To enter the initial state (S_0), the attacker revokes access to `quadrant` and lets the enclave execute until a page fault occurs. To transition into S_1, the enclave is about to execute Line 9. At this point, the attacker reverts access to `quadrant`, and revokes it from `block`. To transition from S_1 to S_2, the code reverts access to `block` and revokes write access from `ftab`.

B. Identifying the Accessed Virtual Page

The transition to S_2 provides the attacker with information about the virtual page the victim accessed in the signal handler. As Xu et al. [37] discovered, even though SGX masks the page offset, the OS has architectural access to the address of page that caused the page fault, albeit without the 12 lower address bits. Thus the attacker only has to leak the page offset bits using other, less accurate, techniques.

C. Leaking Page Offset

The victim performs exactly one architectural access to `ftab` in each iteration of the loop, in the transition between S_3 and S_4. To measure the state of the cache, at the end of S_3, the attacker fills a portion of the cache with their own data, allows the victim to access `ftab`, and accesses their own data again while measuring access time. Ideally, the attacker observes exactly one cache miss that corresponds to the address that the victim accessed. In practice, there could be false positive and false negative measurements due to, e.g., other applications or signal handlers using the same cache. In the remainder of this section, we show our novel techniques that reduce the number of wrong readings and improve attack accuracy.

1) Cache Partitioning: Intel CAT allows system administrators to statically partition the cache *ways* between different CPUs to avoid memory contention. To our knowledge, we are the first to use Intel *Cache Allocation Technology* (CAT) as an offensive tool. More specifically, to avoid cache contention from unrelated applications that can lead to false positives in the cache timing attack, we isolate the CPU that performs the attack from the rest of the system. Unlike traditional cache attacks where the cache replacement policy may pose a challenge, Intel CAT can effectively reduce the cache to a single way.

Furthermore, in addition to *ways*, the cache is also divided into *slices*. Prior work [38] reverse engineered the previously undocumented complex hash function used to map address to their corresponding slices. Since SGX is limited to at most 128M of physical memory on our platform, we precompute the slicing function for these addresses instead of reverse engineering the full function on our platform.

2) Frame Selection Technique: The transition between states might interfere with our attack as it pollutes the cache with memory accesses from SGX and the OS. To overcome this, we develop a framework where the attacker can choose physical frames that do not use conflicting cache sets.

The attacker repeats S_2 until such a frame in an idle cache set is found, i.e., performing all the state transition logic while not performing actual access to `ftab`. If the attacker detects cache activity on the monitored cache sets, the state transition caused this activity, rather than accesses to `ftab`. Therefore, the attacker remaps the frame until they find one that does not collide with noise from the system (or until a timeout).

Before the victim accesses `ftab`, the attacker triggers a transition from S_2 to S_3 that is more similar to the actual access, as the attacker no longer modifies page table mappings. The attacker simply logs any noisy cache lines due to evictions, and will treat them as false positives later on. Finally, the attacker lets the state machine transition from S_3 to S_4, performs the access to `ftab`, and records the measurements.

D. Algorithmic Computation

Section IV-D describes the exact computation for an attacker to convert addresses into input data. Upon collecting all the traces, the attacker performs the same computation. Additionally, as Section IV-D explains, bits from the same plaintext byte are repeated on consecutive iterations of the leakage gadget. In case there are multiple value candidates, the attacker uses this redundancy as a form of error correction.

E. Evaluation

We used an Intel i7-7567U CPU running Ubuntu 20.04 with kernel version 5.4.0 with hyper-threading disabled and all existing mitigations (e.g., against Foreshadow [39]) enabled. We leak 10KB of randomly generated data inside SGX. Due to the lack of redundancy, random data is the hardest data to leak through a side channel. As such, unlike prior work [40, 41], we cannot rely on any sort of error correction. The attack always takes less than 30 seconds to run end-to-end and correctly leaks over 99% of the data bits. Despite the aforementioned techniques, the noise results in a $\leq$1% inaccuracy that cannot be eliminated (e.g., when we exhaust all free physical pages, searching for one unaffected by context switches).

VI. Fingerprinting attack on Bzip2

In this section, we describe our second attack in which a malicious application can identify which file the Bzip2 utility is compressing. Specifically, the attacker application monitors the usage of two cache lines that include the entry point of two functions within the Bzip2 implementation. The reason that these function leaks information about the input file is an optimization in Bzip2 where it uses either of these functions based on the input's repetitiveness and exact length.

Discovering control flow divergence in Bzip2. When we experimented with TaintChannel, we noticed that TaintChannel discovers different leakage gadgets when invoked with different inputs; one in `mainSort()` (which we describe in Section IV-D and exploit in Section V), and another in `fallbackSort()`. We describe below the relevant parts of the Bzip2 compressor that lead to control flow divergence.

Bzip2 implementation. Although BWT (basis for Bzip2) usually achieves better compression than LZ77 and LZ78, it is typically slower as it sorts all the input string's cyclical shifts. The Bzip2 compressor implements multiple performance optimizations to achieve reasonable performance, resulting in different code paths when compressing different files.

Bzip2 splits the input data into blocks and compresses each block separately. Each block is 10,000 bytes, whereas the last block is shorter if the file size is not a multiple of

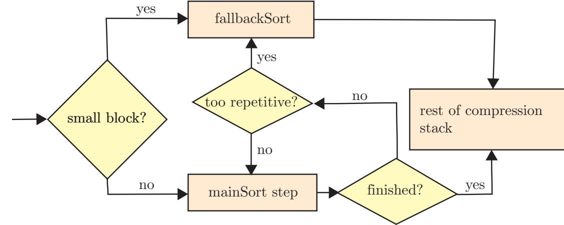

Fig. 6: Control flow of the sorting component of the Bzip2 compressor per input block.

10,000. For each block Fig. 6 illustrates the control flow of Bzip2's sorting. For all 10,000-byte, Bzip2 starts the sorting with `mainSort()`, whereas for shorter blocks it uses `fallbackSort()`. For strings that Bzip2 classifies as too repetitive, it abandons `mainSort()` mid-way and continues with `fallbackSort()`. In our attack, we use this divergence to distinguish different input files. Some differences include the compressor spending more time in `fallbackSort` in a more repetitive file or spending different amount of time in `mainSort` before retreating to `fallbackSort`.

Flush+Reload [2] is arguably the most reliable cache side-channel attack technique for retrieving the control flow within a shared library (e.g., `libbz2`) under the threat model in which the attacker and the victim are different applications on the same system. In our implementation of the Flush+Reload attack, the attacker calls a function in the shared library `libbz2` to map it to their address space. The attacker then starts the active attack: Flushing and reloading the two monitored addresses (one for each monitored function) until the attacker application detects a cache hit in one of them. The attacker then continues flushing and reloading them for an additional 10,000 iterations, which we empirically found to cover the execution duration of the victim. The attacker finally prints out 2 arrays of 10,000 boolean values, where each value represents either a measured cache hit or a cache miss for each of the functions `mainSort()` and `fallbackSort()`. The attacker passes these arrays to a classifier.

Trace classification To classify traces, we train a *deep neural network* (DNN) with Pytorch [42] and 9,334 traces that we collect with compressed files for each experiment chosen at random. As each trace contains 10,000 samples from 2 cache lines (the output from our Flush+Reload program), the input tensor's shape to the neural network is $2 \times 1,000$ filled with values 0 and 1, where 0 and 1 represents a cache miss and hit respectively. If the attacker does not detect any cache activity from the victim after a timeout of 5 seconds, we encode this with a tensor filled with the value 2. Finally, we split our data into training, evaluation, and testing sets of the ratios of 90%, 10%, and 10% respectively. We use these sets for network training, mid-training evaluation and the final evaluation respectively, that we present in this paper.

Evaluation. We use the test files that come with the Brotli [27] compression software, the most comprehensive collection of compression test files we could find and includes

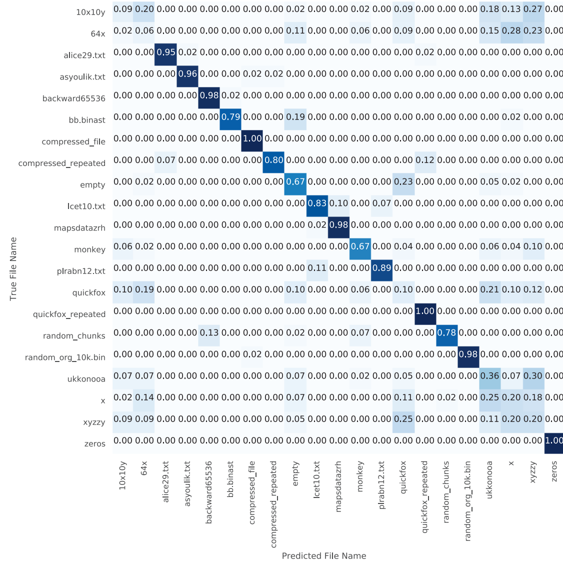

Fig. 7: The accuracy of our classifier on different test files.

21 files. Fig. 7 is the confusion matrix of our trained classifier. In a confusion matrix, the columns present the files that the classifier was challenged with, and the rows are the possible outputs from the classifier. for a perfect classifier, the main diagonal of the confusion matrix where the file names in the row and in the columns are identical would only contain 1, and the rest of the matrix would be 0. We see that our classifier achieves decent accuracy for most files, and struggles to distinguish files that immediately go into `fallbackSort()` without starting from `mainSort()`. For example, the file `x`, whose content is identical to its name, is classified correctly 20% of the time, where the probability of a random guess to classify correctly is 4.76%. Overall, we parsed the data, trained a neural network, evaluated it and generated Fig. 7 using Pytorch using Google Colab resources in under 5 minutes.

Leaking how repetitive a file is. We conduct an experiment to show that an attacker can distinguish files in the absence of other differences between them, such as the location of the repetition, the number of blocks in a file, or whether there exists a shorter block at the end of the file. We create a series of 5 similar files of the same size, 20,000 bytes each. For generating these files, we use the Python utility `lipsum` to output 5 random paragraphs that look similar to English text. To normalize the lengths of these paragraphs, we truncate each of them to the first 20 characters. To generate the ith file, where $1 \leq i \leq 5$, we output a random selection from i first paragraphs. Fig. 8 shows the confusion matrix of the same classifier, this time trained on the traces generated compressing the 5 file. With the baseline of a random guess that 20%, we classify 1st file is correctly classified 98% of the time, and the rest with accuracy between 32% and 52%. Overall, the more repetitive the file is the more accurate the classification is.

VII. Related Work

A. Tools for Verifying that Code is Constant-Time

Most of the existing tools for detecting cache-side channels in software are designed to verify that cryptographic code is constant-time [8, 9, 10, 11, 13, 14, 15, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55], while TaintChannel aims to detect vulnerabilities in general-purpose code. While some of the tools also attempt to find the source of the leakage [16, 56], they do not provide the computations that leads to the leakage, which is information that most attackers rely on.

1) Symbolic Execution Based Tools: These tools [8, 9, 10, 45, 46, 47, 48, 49, 53] verify that applications are constant-time by attempting to explore all of their possible execution paths. Verifying this way that cryptographic libraries are constant-time is easier compared to general applications for a few reasons: First, cryptographic libraries, especially ones that are designed to be constant-time have fewer branching instructions than general-purpose code, limiting the number of paths for a symbolic execution engine to explore. Second, cryptographic code usually only has *reads* from symbolic array indices. However, all the gadgets in compression discussed in Section IV include *writes* to arrays in symbolic indices, e.g., for constructing histograms of the input stream. Writes to symbolic array indices can be extremely inefficient in symbolic execution engines, for example KLEE [57] forks the memory state for each possible value in each possible index. In the case of Bzip2, that would mean $65,536$ forks of the memory for each pair of input bytes, which is infeasible.

2) Trace Based Tools: Trace based tools operate in two steps: First, they collect a trace of all cache accesses that an application performs either by instrumenting the application [13, 14, 15, 16, 43, 56] to output its memory accesses or by mounting an end-to-end cache attack [11, 16, 44, 56]. In the second step, these tools process the traces and look for a correlation between the secret inputs and the traces. Although these tools aim to detect whether there is a statistical correlation between a secret input and the trace generated running the application with that input, they inherently cannot yield the exact computation that converts the input into a

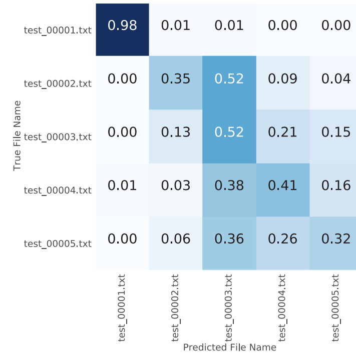

Fig. 8: The classifier's accuracy on similar files with different repetitiveness factor.

secret-dependent memory access, knowledge that was used in the overwhelming majority of existing side-channel attacks . Interestingly, a recent work, CacheQL [56] whose main contribution is combining features from a wide array of existing tools does manual taint tracking.

3) Static Analysis Based Tools: Static analysis tools analyze the information flows of an application without executing it. To verify that programs do not have cache side-channel leakage, static analysis tools rely on specific features of particular programming languages, which do not include C, the language of most compression software. Examples of such tools are: **FlowTracker [50]** verifies that programs written in the *While* programming language are constant-time. FlowTracker also has a prototype that works on LLVM IR, however, because LLVM IR has different properties than While this prototype does not hold the proven security guarantees; **Blazer [51]** is a tool that statically verifies that Java ByteCode is constant-time; **Themis [52]** verifies Java applications; **BPT17 [53]** verifies that code written in the *C#Minor* programming language is constant-time; **CT-Wasm [54]** defines a subset of Web assembly that the tool can verify to be constant-time. **virtualCert [55]** verifies programs in a subset of the Match programming language they call MatchIR.

B. Attacks related to compression

Arguably the most famous compression-related attack is the zip bomb[58, 59]. This type of attack involves the creation of a compressed file that, when decompressed, expands to an unmanageable and potentially system-crashing size. Zip bombs function by exploiting the repetitive patterns and redundancies within compression algorithms, causing an exponential increase in file size upon decompression. These files, often masquerading as harmless archives, can overwhelm storage resources, exhaust system memory, and disrupt normal operations. Unlike zip bomb, in this paper we explore side-channel leakage.

The CRIME (Compression Ratio Info-leak Made Easy) and BREACH (Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext) attacks [6] are security vulnerabilities that exploit compression algorithms in the context of web communication. Both attacks rely on the attacker controlling a portion of the compressed data to leak unknown data, such as HTTP cookies. CRIME, disclosed in 2012, primarily targets the TLS (Transport Layer Security) protocol. By observing the compression ratio of encrypted data, an attacker can infer information about the plaintext, potentially leading to the disclosure of sensitive information. On the other hand, BREACH, introduced in 2013, extends the threat to web applications relying on HTTP compression. BREACH leverages the compression oracle to infer details about the content of web responses, posing a significant risk to the confidentiality of data exchanged between users and web servers. However, unlike ZipChannel, CRIME and BREACH assume partial control over the compressed input and the ability to compress it multiple times.

Schwarzl et al. [7] also assume a partial control over compressed data by the attacker and measure the execution time of LZ77-based compression software.

Kelsey [60] explore theoretical side-channel attacks on compression, observing the difference in size between the original and compressed data. This differs from our work since we look at practical implementations and the side-channel, which might contain more information than the file size.

C. Attacks on SGX

Controlled-channel attacks on SGX [37, 41] were demonstrated on libjpeg [61] using a simulator. On the contrary, ZipChannel works on real hardware. While prior work does not report leakage accuracy, the papers include visibly distorted JPEG images leaked through the attack, potentially indicating a lower accuracy than our SGX attack where we correctly leak 99% of the data.

Cache attacks on SGX. Most cache attacks on SGX have targeted the L1 cache [41, 62], however, since the Foreshadow mitigation [39] blocks L1 cache attacks in some CPUs, attacks, including ours, resort to other levels.

Sieck et al. [40] target the L3 cache. However, they do not leak the entire key and rely on mathematical properties of RSA to retrieve the missing bits. While they do not report raw leakage rate, in related work, Yarom et al. [63] report that they can reconstruct RSA keys from 60% the key.

VIII. Mitigations

Constant time compression. The deployed mitigation for cryptographic side-channels is constant-time code. Thus, constant-time compression implementations may be helpful.

Application-specific constant-time compression. It may be feasible to secure application-specific compression from side-channel leakage using additional knowledge about the format. For example, in the case of HTML, most opening tags must have a matching closing tag.

IX. Conclusion

We presented TaintChannel, a tool to automatically detect cache side-channel vulnerabilities. We used TaintChannel to study cache side-channel vulnerabilities in the implementation of the widely used compression algorithms: LZ77, LZ78, and BWT. We discovered that all of these algorithms are vulnerable. Finally, we present two end-to-end attacks on Bzip2 using the vulnerabilities discovered by TaintChannel- in the first attack the victim application runs within an SGX enclave and in the second attack both the attacker and the victim applications share the same system.

X. Acknowledgments

We thank the anonymous reviewers and our shepherd, Herbert Bos, for their insightful suggestions and feedback. The work was supported by Marina Minkin's Meta fellowship and the Assured Micropatching a DARPA program.

References

[1] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' track at the RSA conference*. Springer, 2006, pp. 1–20.

[2] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 719–732.

[3] W3Techs, "Usage statistics of compression for websites," https://w3techs.com/technologies/details/ce-compression, 2022.

[4] ——, "Usage statistics of default protocol https for websites," https://w3techs.com/technologies/details/ce-httpsdefault, 2023.

[5] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[6] T. Duong and J. Rizzo, "Breach attack," http://breachattack.com/, (Accessed on 08/06/2021).

[7] M. Schwarzl, P. Borrello, G. Saileshwar, H. Müller, M. Schwarz, and D. Gruss, "Practical timing side channel attacks on memory compression," *arXiv preprint arXiv:2111.08404*, 2021.

[8] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "{CacheD}: Identifying {Cache-Based} timing channels in production software," in *26th USENIX security symposium (USENIX security 17)*, 2017, pp. 235–252.

[9] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. Kandemir, "Casym: Cache aware symbolic execution for side channel detection and mitigation," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 505–521.

[10] L.-A. Daniel, S. Bardin, and T. Rezk, "Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1021–1038.

[11] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive {Last-Level} caches," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 897–912.

[12] Y. Xiao, M. Li, S. Chen, and Y. Zhang, "Stacco: Differentially analyzing side-channel traces for detecting ssl/tls vulnerabilities in secure enclaves," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 859–874.

[13] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar, "Microwalk: A framework for finding side channels in binaries," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 161–173.

[14] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl, "{DATA}–differential address trace analysis: Finding address-based {Side-Channels} in binaries," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 603–620.

[15] S. Weiser, D. Schrammel, L. Bodner, and R. Spreitzer, "Big numbers-big troubles: Systematically analyzing nonce leakage in ({EC) DSA} implementations," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1767–1784.

[16] Y. Xiao, M. Li, S. Chen, and Y. Zhang, "Automated side channel analysis of media software with manifold learning," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/yuan

[17] P. Deutsch *et al.*, "Gzip file format specification version 4.3," May 1996.

[18] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[19] "ncompress: a public domain project," https://ncompress.sourceforge.io/, (Accessed on 10/12/2021).

[20] M. Burrows and D. Wheeler, "A block-sorting lossless data compression algorithm," in *Digital SRC Research Report*. Citeseer, 1994.

[21] J. Seward, "bzip2 and libbzip2," *avaliable at http://www.bzip. org*, 1996.

[22] A. Shusterman, Z. Avraham, E. Croitoru, Y. Haskal, L. Kang, D. Levi, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Website fingerprinting through the cache occupancy channel and its real world practicality," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2042–2060, 2020.

[23] "Gzip file compression utility buffer overflow used by many ftp servers allows remote users to execute arbitrary code on the ftp server - securitytracker," https://securitytracker.com/id/1002771, (Accessed on 08/06/2021).

[24] P. Deutsch and J.-L. Gailly, "Zlib compressed data format specification version 3.3," RFC 1950, Tech. Rep., May 1996.

[25] "https://pkware.cachefly.net/webdocs/casestudies/appnote.txt," https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT, 1989, (Accessed on 08/19/2021).

[26] T. Boutell, "Png (portable network graphics) specification version 1.0," RFC 2083, Tech. Rep., Mar. 1997.

[27] J. Alakuijala, A. Farruggia, P. Ferragina, E. Kliuchnikov, R. Obryk, Z. Szabadka, and L. Vandevenne, "Brotli: A general-purpose data compressor," *ACM Transactions on Information Systems (TOIS)*, vol. 37, no. 1, pp. 1–30, 2018.

[28] D. Berz, M. Engstler, M. Heindl, and F. Waibel, "Comparison of lossless data compression methods," 2015.

[29] T. A. Welch, "A technique for high-performance data compression," *Computer*, vol. 17, no. 06, pp. 8–19, 1984.

[30] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," in *USENIX Security Symposium*, 2004, pp. 321–336.

[31] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration

of exploits on commodity software." in *NDSS*, vol. 5. Citeseer, 2005, pp. 3–4.

[32] "Dynamorio," https://dynamorio.org/, (Accessed on 04/26/2022).

[33] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[34] P. Deutsch, "Deflate compressed data format specification version 1.3," RFC 1951, May 1996.

[35] "Content-encoding - http — mdn," https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Encoding, (Accessed on 08/19/2021).

[36] J. Van Bulck, F. Piessens, and R. Strackx, "Sgx-step: A practical attack framework for precise enclave execution control," in *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017, pp. 1–6.

[37] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 640–656.

[38] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 605–622.

[39] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 991–1008.

[40] F. Sieck, S. Berndt, J. Wichelmann, and T. Eisenbarth, "Util:: Lookup: Exploiting key decoding in cryptographic libraries," *arXiv preprint arXiv:2108.04600*, 2021.

[41] M. Hähnel, W. Cui, and M. Peinado, "High-resolution side channels for untrusted operating systems," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 299–312.

[42] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[43] S. He, M. Emmi, and G. Ciocarlie, "ct-fuzz: Fuzzing for timing leaks," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 466–471.

[44] O. Reparaz, J. Balasch, and I. Verbauwhede, "Dude, is my code constant time?" in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1697–1702.

[45] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, "Cacheaudit: A tool for the static analysis of cache side channels," *ACM Transactions on Information and System Security (TISSEC)*, vol. 18, no. 1, pp. 1–32, 2015.

[46] S. Wang, Y. Bao, X. Liu, P. Wang, D. Zhang, and D. Wu, "Identifying {Cache-Based} side channels through {Secret-Augmented} abstract interpretation," in *28th USENIX security symposium (USENIX security 19)*, 2019, pp. 657–674.

[47] Q. Bao, Z. Wang, X. Li, J. R. Larus, and D. Wu, "Abacus: Precise side-channel analysis," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 797–809.

[48] S. Chattopadhyay, M. Beck, A. Rezine, and A. Zeller, "Quantifying the information leakage in cache attacks via symbolic execution," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 1, pp. 1–27, 2019.

[49] C. Sung, B. Paulsen, and C. Wang, "Canal: a cache timing analysis framework via llvm transformation," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 904–907.

[50] B. Rodrigues, F. M. Quintão Pereira, and D. F. Aranha, "Sparse representation of implicit flows with applications to side-channel detection," in *Proceedings of the 25th International Conference on Compiler Construction*, 2016, pp. 110–120.

[51] T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei, "Decomposition instead of self-composition for proving the absence of timing channels," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 362–375, 2017.

[52] J. Chen, Y. Feng, and I. Dillig, "Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 875–890.

[53] S. Blazy, D. Pichardie, and A. Trieu, "Verifying constant-time implementations by abstract interpretation (extended version)," *Journal of Computer Security*, 2018.

[54] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "Ct-wasm: type-driven secure cryptography for the web ecosystem," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[55] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie, "System-level non-interference for constant-time cryptography," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1267–1279.

[56] Y. Yuan, Z. Liu, and S. Wang, "Cacheql: Quantifying and localizing cache side-channel vulnerabilities in production software," *arXiv preprint arXiv:2209.14952*, 2022.

[57] C. Cadar, D. Dunbar, and D. Engler, "Klee: unassisted

and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[58] "42.zip," https://unforgettable.dk/, (Accessed on 02/20/2024).

[59] D. Fifield, "A better zip bomb," in *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, 2019.

[60] J. Kelsey, "Compression and information leakage of plaintext," in *International Workshop on Fast Software Encryption*. Springer, 2002, pp. 263–276.

[61] W. B. Pennebaker and J. L. Mitchell, *JPEG: Still image data compression standard*. Springer Science & Business Media, 1992.

[62] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How sgx amplifies the power of cache attacks," in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017, pp. 69–90.

[63] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: a timing attack on openssl constant-time rsa," *Journal of Cryptographic Engineering*, vol. 7, pp. 99–112, 2017.