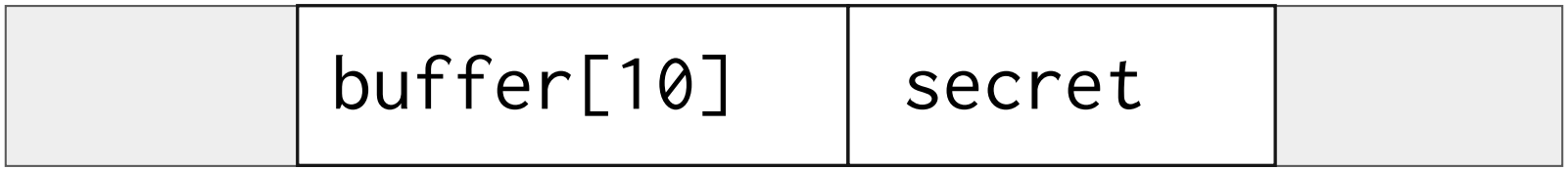
Delta Pointers: Buffer Overflow Checks Without the Checks

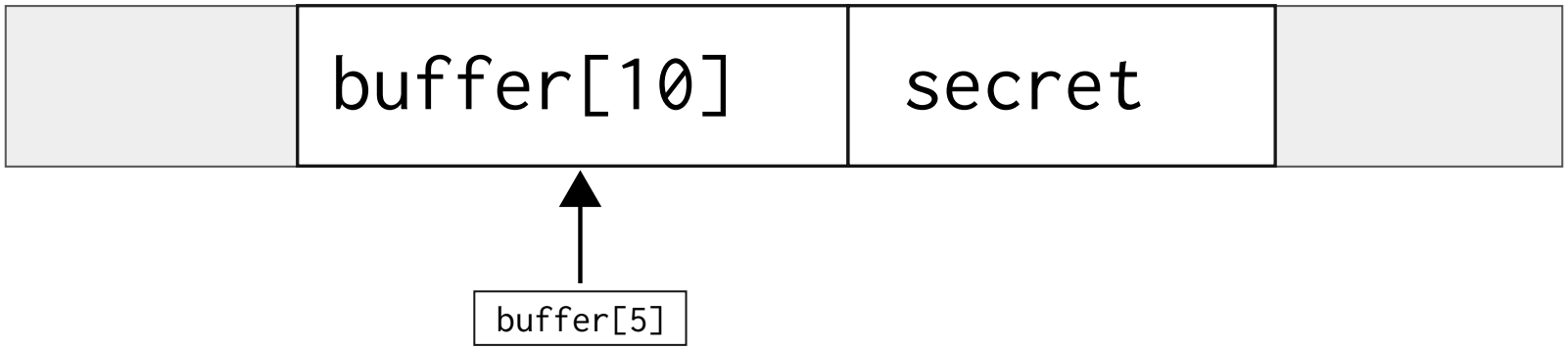
Taddeüs Kroes & Koen Koning
Erik van der Kouwe Herbert Bos
Cristiano Giuffrida

June 19, 2018

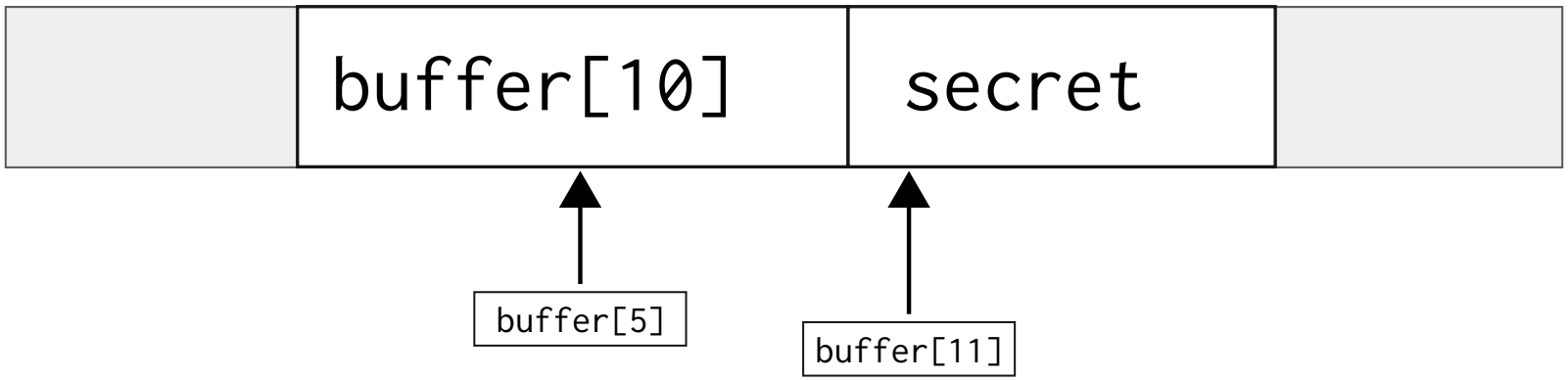
Preview



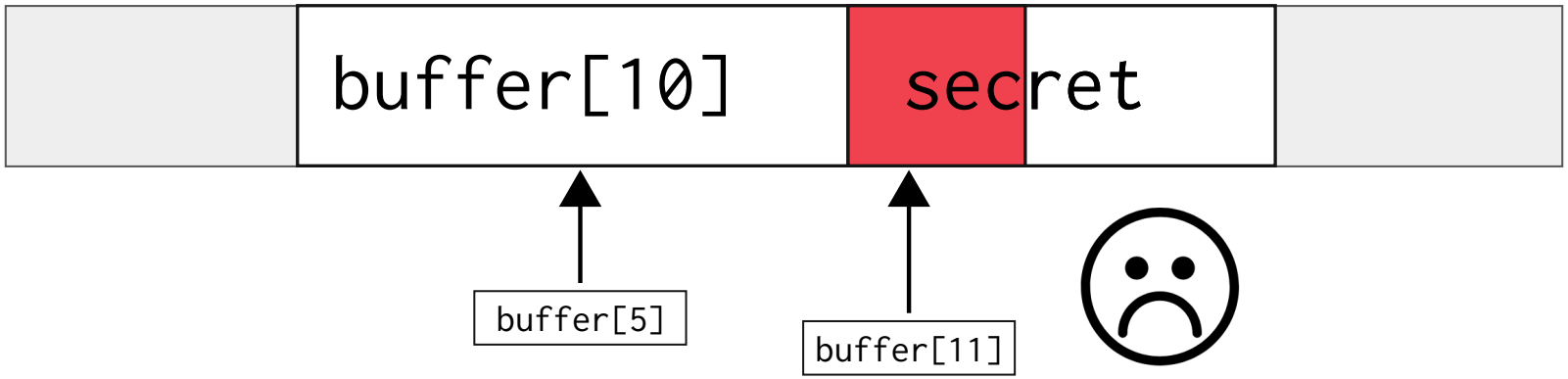
Preview



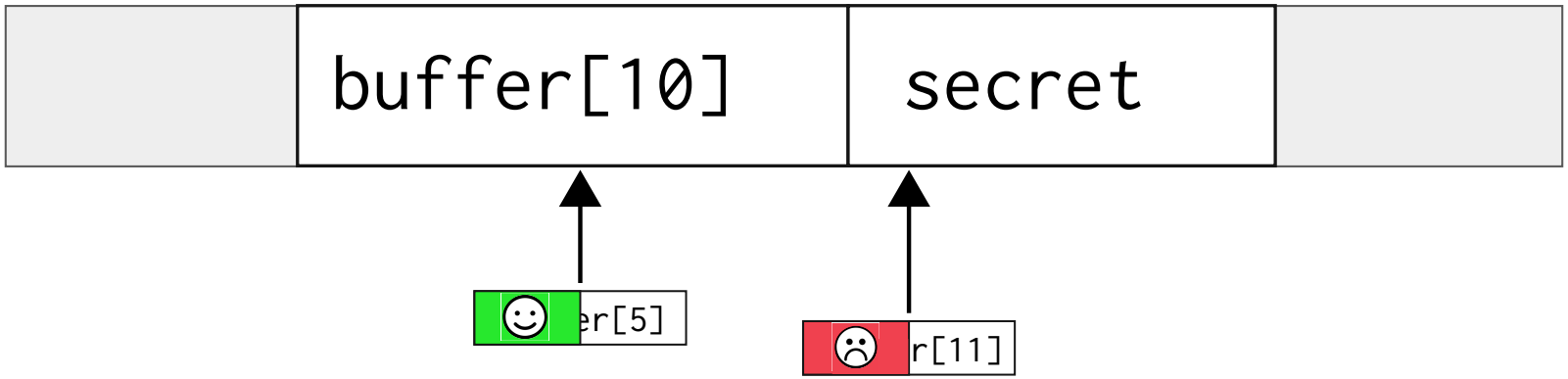
Preview



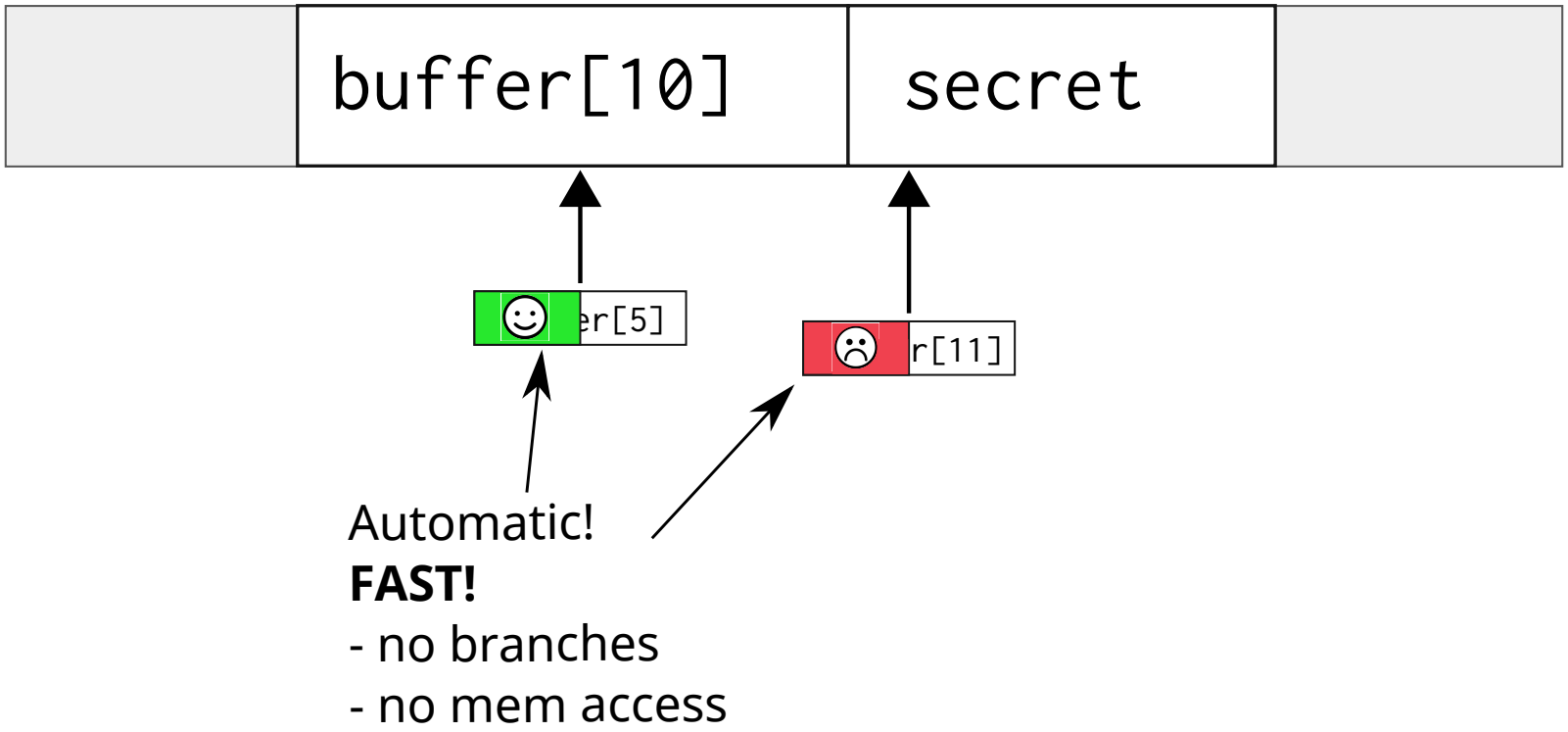
Preview



Preview



Preview


buffer[10]
secret
er[5]
r[11]
Automatic!
FAST!
- no branches
- no mem access

Buffer overflows still very common today

CVE List CNAs Board About News & Blog

NVD

Search CVE List | Download CVE | Data Feeds | Request CVE IDs | Update a CVE Entry

TOTAL CVE Entries: 98433

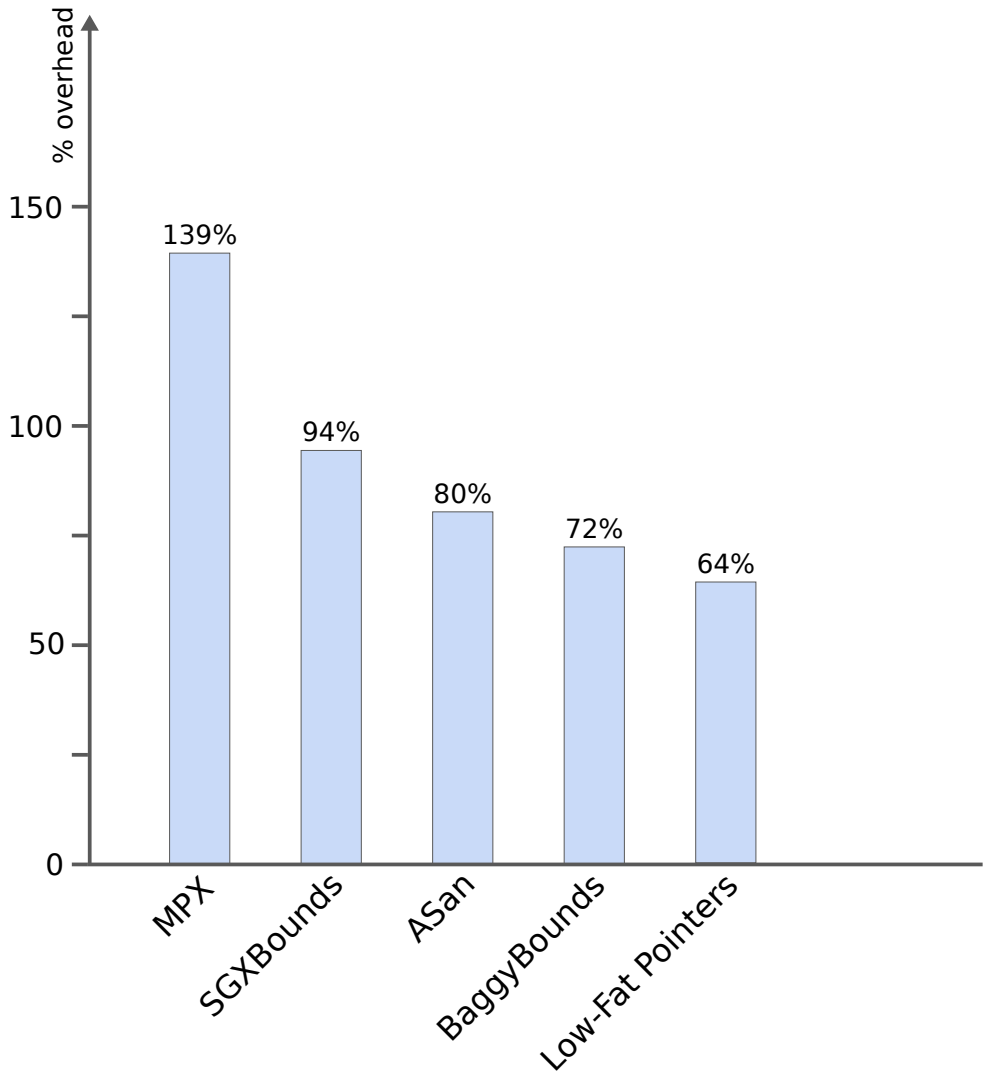
HOME > CVE > SEARCH RESULTS

Search Results

There are **9000** CVE entries that match your search.

Name	Description
CVE-2018-8905	In LibTIFF 4.0.9, a heap-based buffer overflow occurs in the function LZWDecodeCompat in tif_lzw.c via a crafted TIFF file, as demonstrated by tiff2ps.
CVE-2018-8828	A Buffer Overflow issue was discovered in Kamailio before 4.4.7, 5.0.x before 5.0.6, and 5.1.x before 5.1.2. A specially crafted REGISTER message with a malformed branch or From tag triggers an off-by-one heap-based buffer overflow in the tmx_check_pretran function in modules/tmx/tmx_pretran.c.
CVE-2018-8100	The JPXStream::readTilePart function in JPXStream.cc in xpdf 4.00 allows attackers to launch denial of service (heap-based buffer overflow and application crash) or possibly have unspecified other impact via a specific pdf file, as demonstrated by
CVE-2018-8000	In PoDoFo 0.9.5, there exists a heap-based buffer overflow vulnerability in PoDoFo::PdfTokenizer::GetNextToken() in PdfTokenizer.cpp, a related issue to CVE-2017-5886. Remote attackers could leverage this vulnerability to cause a de execute arbitrary code via a crafted pdf file.
CVE-2018-7886	An issue was discovered in CloudMe 1.11.0. An unauthenticated local attacker that can connect to the "CloudMe Sync" client application listening on 127.0.0.1 port 8888 can send a malicious payload causing a buffer overflow will result in code execution, as demonstrated by a TCP reverse shell, or a crash. NOTE: this vulnerability exists because of an incomplete fix for CVE-2018-6892.
CVE-2018-7877	There is a heap-based buffer overflow in the getString function of util/decompile.c in libming 0.4.8 for DOUBLE data. A Crafted input will lead to a denial of service attack.
CVE-2018-7873	There is a heap-based buffer overflow in the getString function of util/decompile.c in libming 0.4.8 for INTEGER data. A Crafted input will lead to a denial of service attack.
CVE-2018-7867	There is a heap-based buffer overflow in the getString function of util/decompile.c in libming 0.4.8 during a RegisterNumber sprintf. A Crafted input will lead to a denial of service attack.
CVE-2018-7752	GPAC through 0.7.1 has a Buffer Overflow in the gf_media_avc_read_sps function in media_tools/av_parsers.c, a different vulnerability than CVE-2018-1000100.
CVE-2018-7648	An issue was discovered in mj2/opj_mj2_extract.c in OpenJPEG 2.3.0. The output prefix was not checked for length, which could overflow a buffer, when providing a prefix with 50 or more characters on the command line.
CVE-2018-7561	Stack-based Buffer Overflow in httpd on Tenda AC9 devices V15.03.05.14_EN allows remote attackers to cause a denial of service or possibly have unspecified other impact.
CVE-2018-7553	There is a heap-based buffer overflow in the pcxLoadRaster function of in_pcx.cpp in sam2p 0.49.4. A crafted input will lead to a denial of service or possibly unspecified other impact.
CVE-2018-7519	In Omron CX-Supervisor Versions 3.30 and prior, parsing malformed project files may cause a heap-based buffer overflow.
CVE-2018-7513	In Omron CX-Supervisor Versions 3.30 and prior, parsing malformed project files may cause a stack-based buffer overflow.
CVE-2018-7511	In Eaton ELCSoft versions 2.04.02 and prior, there are multiple cases where specially crafted files could cause a buffer overflow which, in turn, may allow remote execution of arbitrary code.
CVE-2018-7487	There is a heap-based buffer overflow in the LoadPCX function of in_pcx.cpp in sam2p 0.49.4. A Crafted input will lead to a denial of service or possibly unspecified other impact.
CVE-2018-7445	A buffer overflow was found in the MikroTik RouterOS SMB service when processing NetBIOS session request messages. Remote attackers with access to the service can exploit this vulnerability and gain code execution on the syste authentication takes place, so it is possible for an unauthenticated remote attacker to exploit it. All architectures and all devices running RouterOS before versions 6.41.3/6.42rc27 are vulnerable.
CVE-2018-7409	In unixODBC before 2.3.5, there is a buffer overflow in the unicode_to_ansi_copy() function in DriverManager/__info.c.
CVE-2018-7284	A Buffer Overflow issue was discovered in Asterisk through 13.19.1, 14.x through 14.7.5, and 15.x through 15.2.1, and Certified Asterisk through 13.18-cert2. When processing a SUBSCRIBE request, the res_pjsip_pubsub module stores in the Accept headers of the request. This code did not limit the number of headers it processed, despite having a fixed limit of 32. If more than 32 Accept headers were present, the code would write outside of its memory and cause a cra
CVE-2018-7254	The ParseCaffHeaderConfig function of the cli/caff.c file of WavPack 5.1.0 allows a remote attacker to cause a denial-of-service (global buffer over-read), or possibly trigger a buffer overflow or incorrect memory allocation, via a maliciously crafted CA
CVE-2018-7247	An issue was discovered in pixHtmlViewer in prog/htmlviewerc in Leptonica before 1.75.3. Unsanitized input (rootname) can overflow a buffer, leading potentially to arbitrary code execution or possibly unspecified other impact.
CVE-2018-7238	A buffer overflow vulnerability exist in the web-based GUI of Schneider Electric's Pelco Sarix Professional in all firmware versions prior to 3.29.67 which could allow an unauthenticated, remote attacker to execute arbitrary code.
CVE-2018-7186	Leptonica before 1.75.3 does not limit the number of characters in a %s format argument to fscanf or sscanf, which allows remote attackers to cause a denial of service (stack-based buffer overflow) or possibly have unspecified other impact via a long demonstrated by the gplotRead and ptaReadStream functions.
CVE-2018-7183	Buffer overflow in the decodearr function in ntpq in ntp 4.2.8p6 through 4.2.8p10 allows remote attackers to execute arbitrary code by leveraging an ntpq query and sending a response with a crafted array.
CVE-2018-7039	CCN-lite 2.0.0 Beta allows remote attackers to cause a denial of service (buffer overflow) or possibly have unspecified other impact because the ccnl_ndntlv_prependBlob function in ccnl-pkt-ndntlv.c can be called with wrong arguments. Specifically, there is an incorrect integer data type causing a negative third argument in some cases of crafted TLV data with inconsistent length information.
CVE-2018-6953	In CCN-lite 2, the Parser of NDNTLV does not verify whether a certain component's length field matches the actual component length, which has a resultant buffer overflow and out-of-bounds memory accesses.
CVE-2018-6948	In CCN-lite 2, the function ccnl_prefix_to_str_detailed can cause a buffer overflow, when writing a prefix to the buffer buf. The maximal size of the prefix is CCNL_MAX_PREFIX_SIZE; the buffer has the size CCNL_MAX_PREFIX_SIZE. However, when NFN is enabled, additional characters are written to the buffer (e.g., the "NFN" and "R2C" tags). Therefore, sending an NFN-R2C packet with a prefix of size CCNL_MAX_PREFIX_SIZE can cause an overflow of buf inside ccnl_prefix_to_str_detailed.
CVE-2018-6892	An issue was discovered in CloudMe before 1.11.0. An unauthenticated remote attacker that can connect to the "CloudMe Sync" client application listening on port 8888 can send a malicious payload causing a buffer overflow condition. This will result in an attacker controlling the program's execution flow and allowing arbitrary code execution.
CVE-2018-6789	An issue was discovered in the base64d function in the SMTP listener in Exim before 4.90.1. By sending a handcrafted message, a buffer overflow may happen. This can be used to execute code remotely.
CVE-2018-6758	The uwsgi_expand_path function in core/utils.c in Unbit uWSGI through 2.0.15 has a stack-based buffer overflow via a large directory length.
CVE-2018-6638	A stack-based buffer overflow (Remote Code Execution) issue was discovered in Design Science MathType 6.9c. This occurs in a function call in which the first argument is a corrupted offset value and the second argument is a stack buffer. This is fixed in 6.9d.
CVE-2018-6537	A buffer overflow vulnerability in the control protocol of Flexense SyncBreeze Enterprise v10.4.18 allows remote attackers to execute arbitrary code by sending a crafted packet to TCP port 9121.
CVE-2018-6481	A buffer overflow vulnerability in the control protocol of Disk Savvy Enterprise v10.4.18 allows remote attackers to execute arbitrary code by sending a crafted packet to TCP port 9124.
CVE-2018-6358	The printDefineFont2 function (util/listfdb.c) in libming through 0.4.8 is vulnerable to a heap-based buffer overflow, which may allow attackers to cause a denial of service or unspecified other impact via a crafted FDB file.
CVE-2018-6297	Buffer overflow in Hanwha Techwin Smartcams

Bounds checking is slow ☹

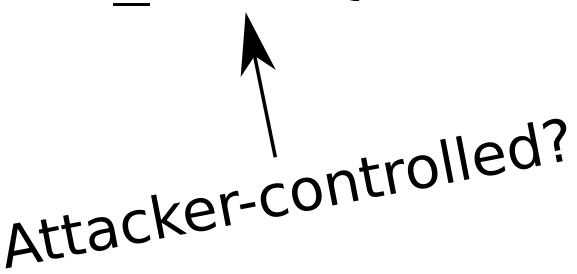


What is bounds checking?

```
void foo(char *buffer, size_t n) {
    buffer[n] = 10;
}
```

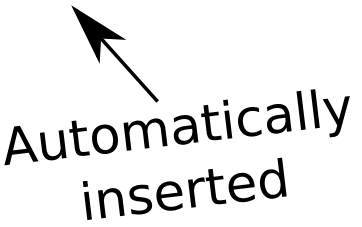
What is bounds checking?

```
void foo(char *buffer, size_t n) {
    buffer[n] = 10;
}
```



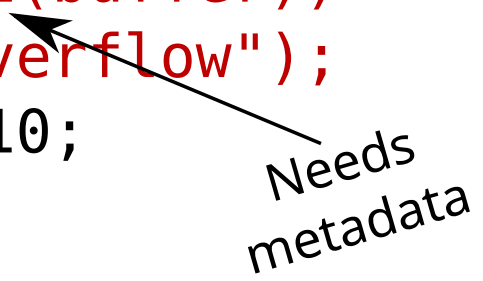
What is bounds checking?

```
void foo(char *buffer, size_t n) {
    if (n >= SIZE(buffer))
        ERROR("overflow");
    buffer[n] = 10;
}
```

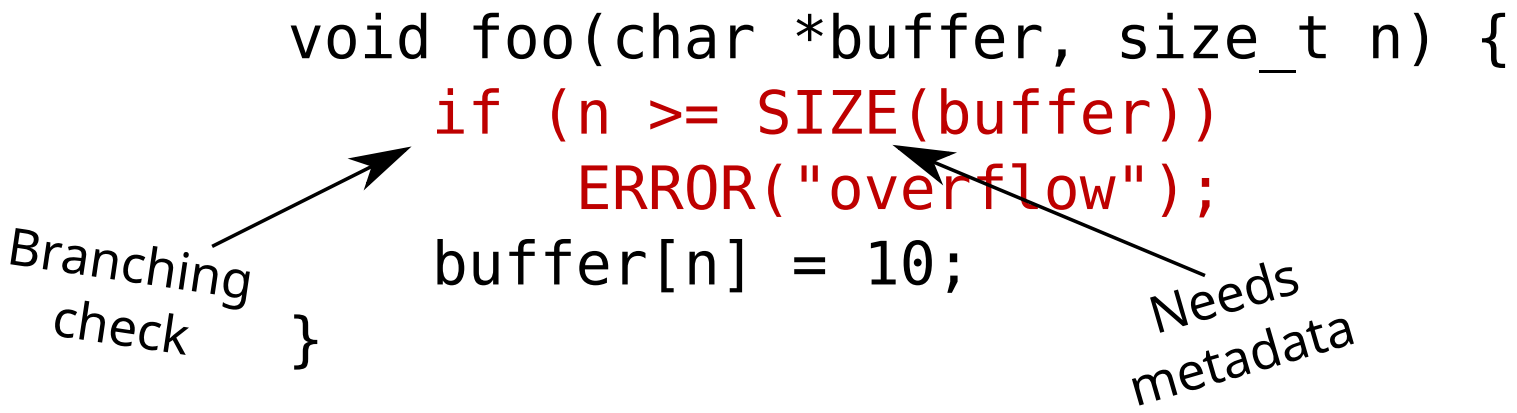


What is bounds checking?

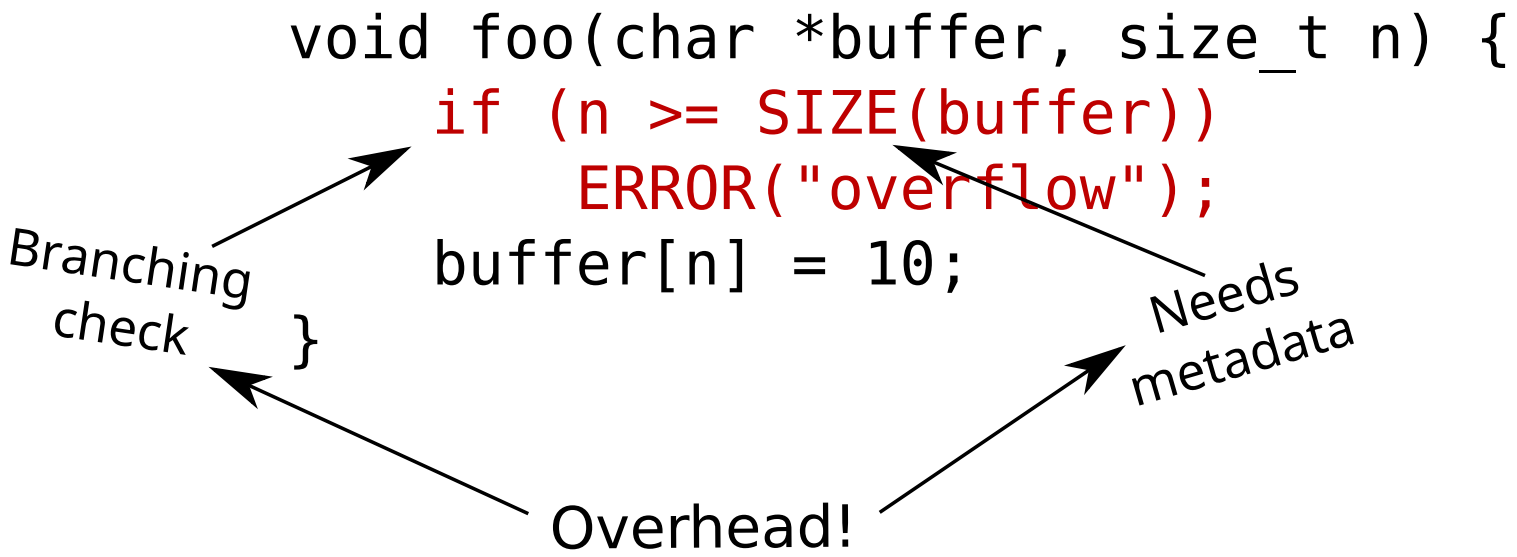
```
void foo(char *buffer, size_t n) {
    if (n >= SIZE(buffer))
        ERROR("overflow");
    buffer[n] = 10;
}
```



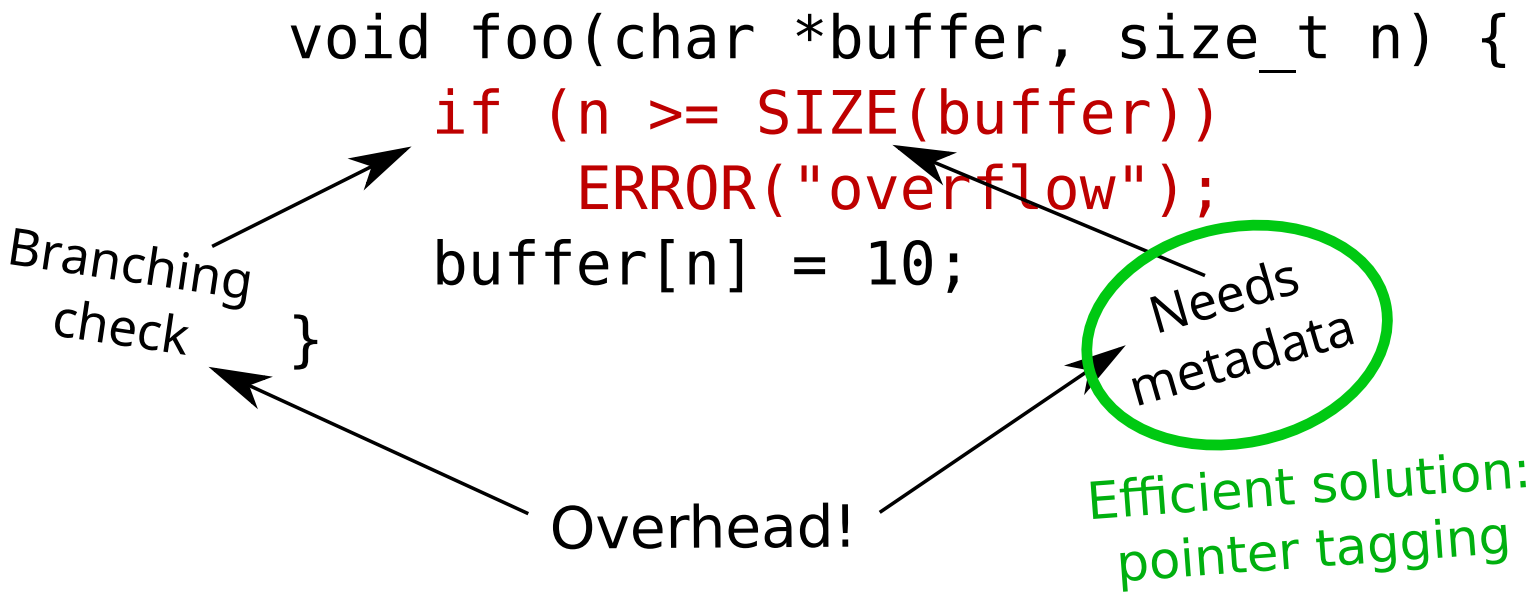
What is bounds checking?



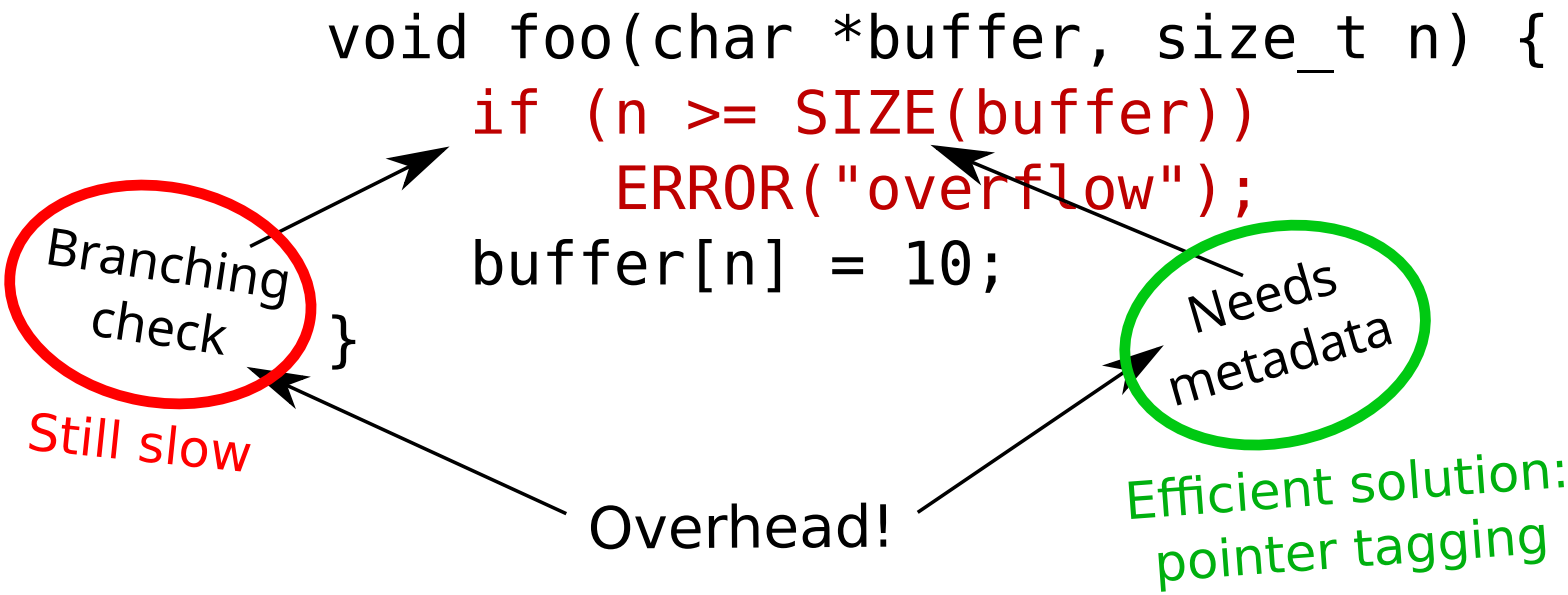
What is bounds checking?



What is bounds checking?



What is bounds checking?



Our approach: Delta Pointers

- Use pointer tagging
  - No memory access for metadata lookup
- No need for branches
  - Delegate checks to (off-the-shelf) hardware instead
- Focus on common case: upper bound on x86_64
  - Mitigates all CVEs reported by related work

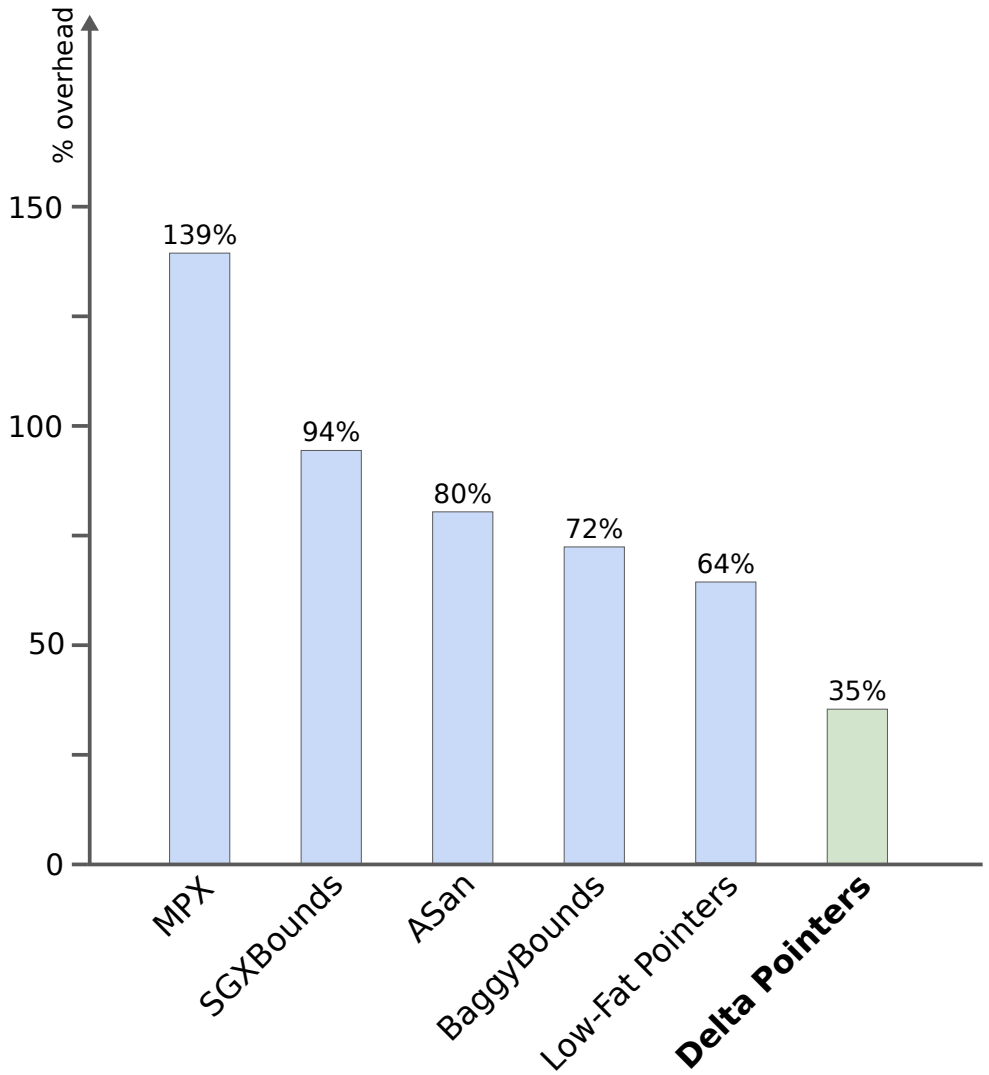
Our approach: Delta Pointers

- Use pointer tagging
  - No memory access for metadata lookup
- No need for branches
  - Delegate checks to (off-the-shelf) hardware instead
- Focus on common case: upper bound on x86_64
  - Mitigates all CVEs reported by related work

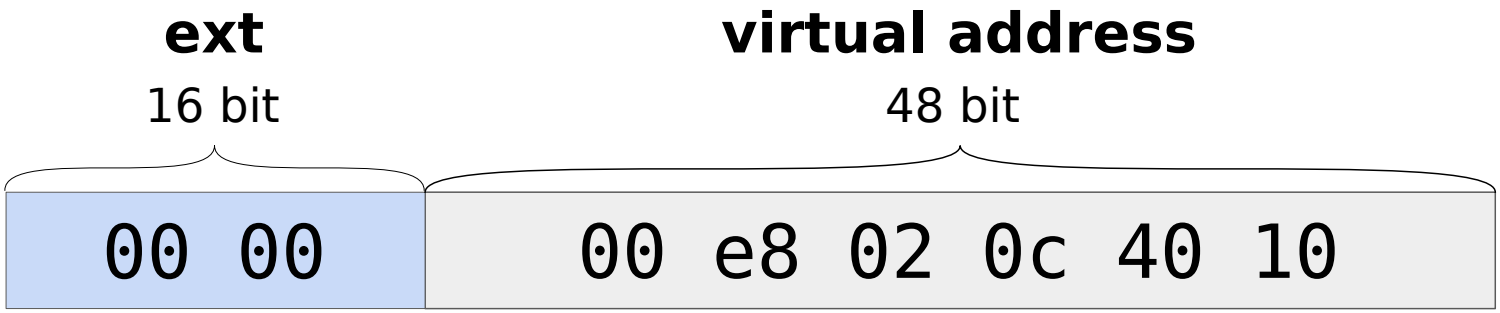
Our approach: Delta Pointers

- Use pointer tagging
  - No memory access for metadata lookup
- No need for branches
  - Delegate checks to (off-the-shelf) hardware instead
- Focus on common case: upper bound on x86_64
  - Mitigates all CVEs reported by related work

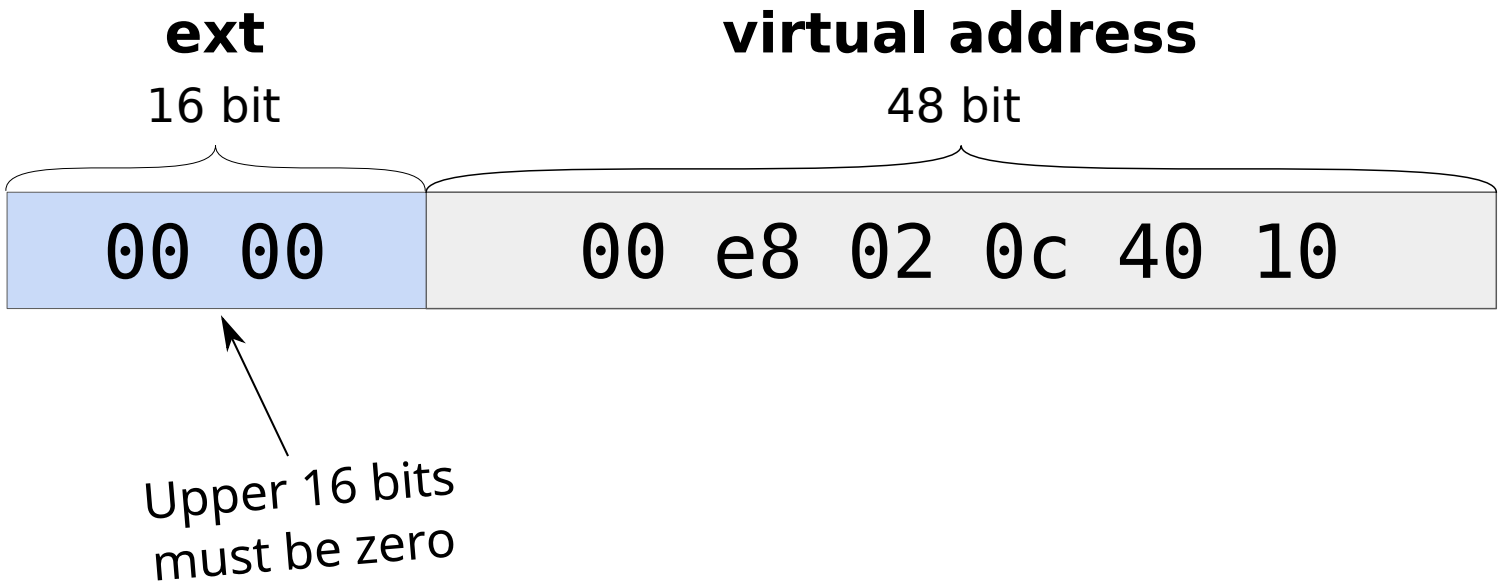
Our approach: Delta Pointers



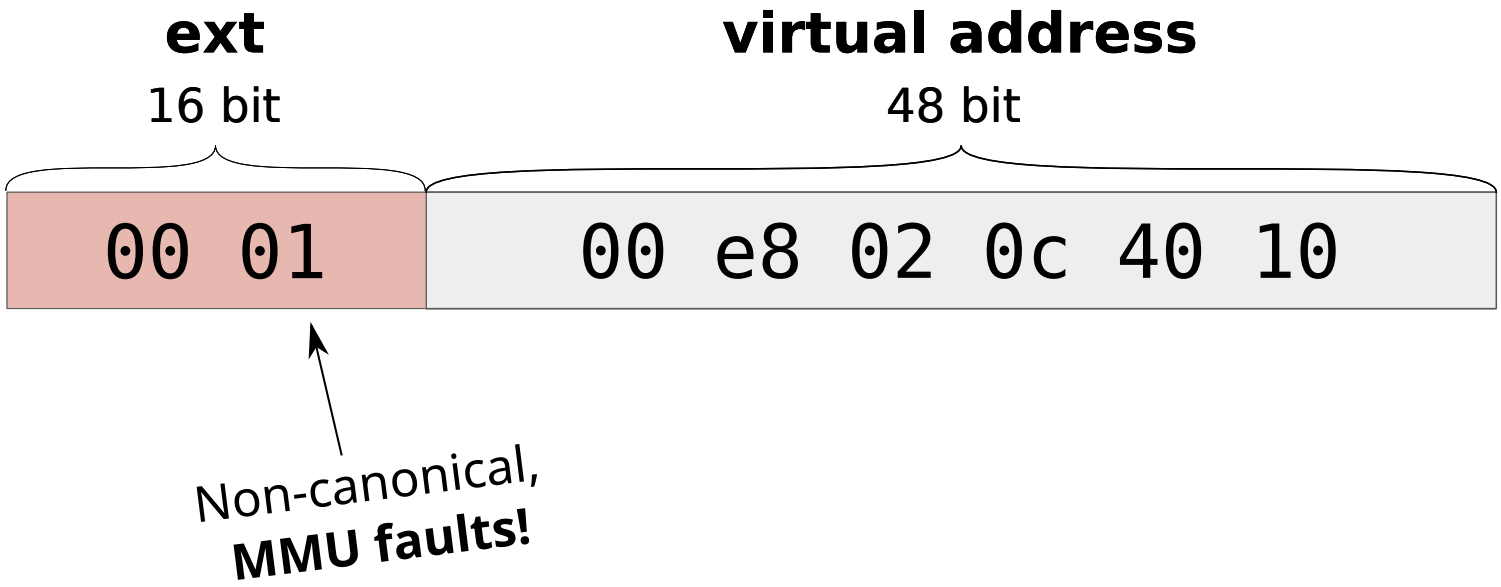
Regular pointers



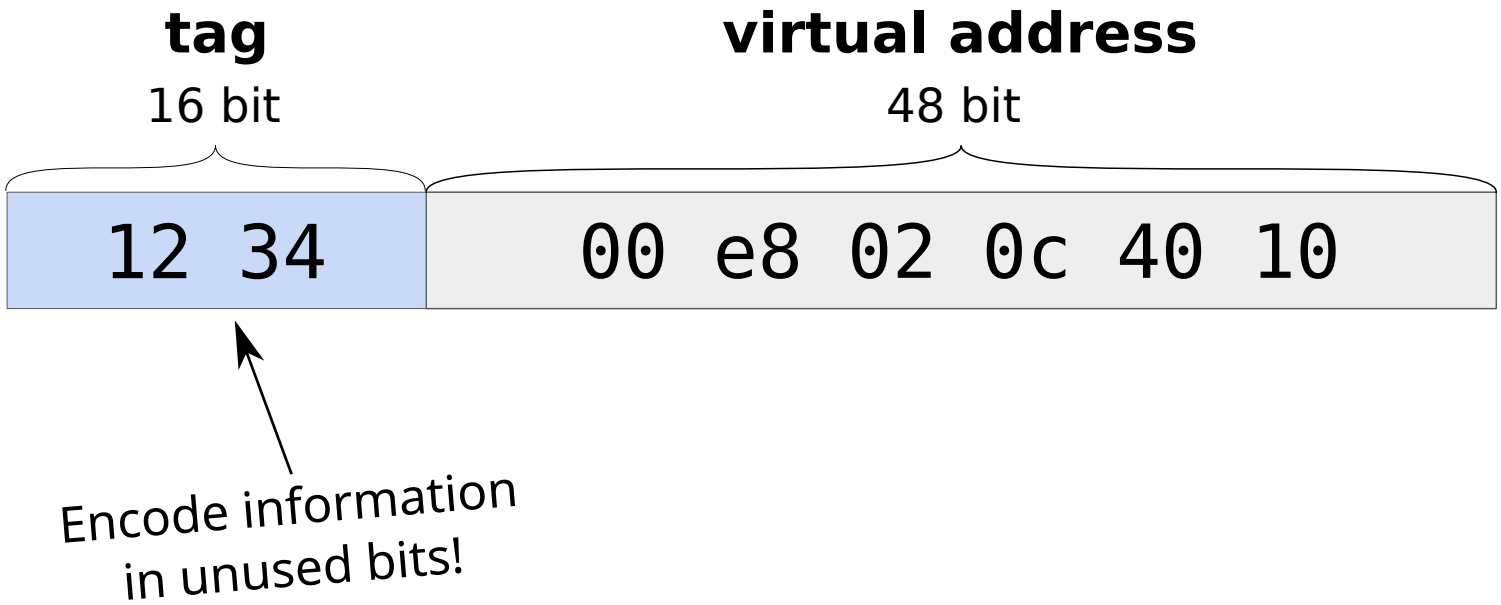
Regular pointers



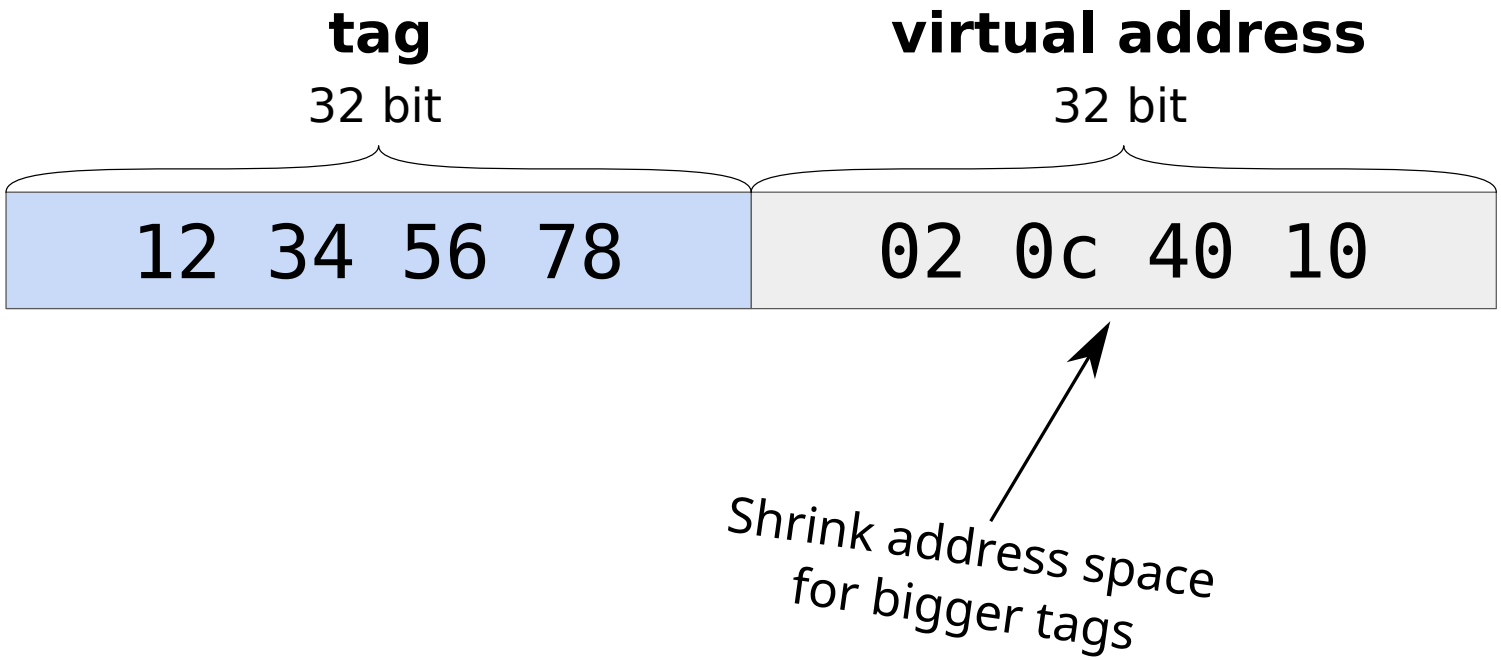
Regular pointers



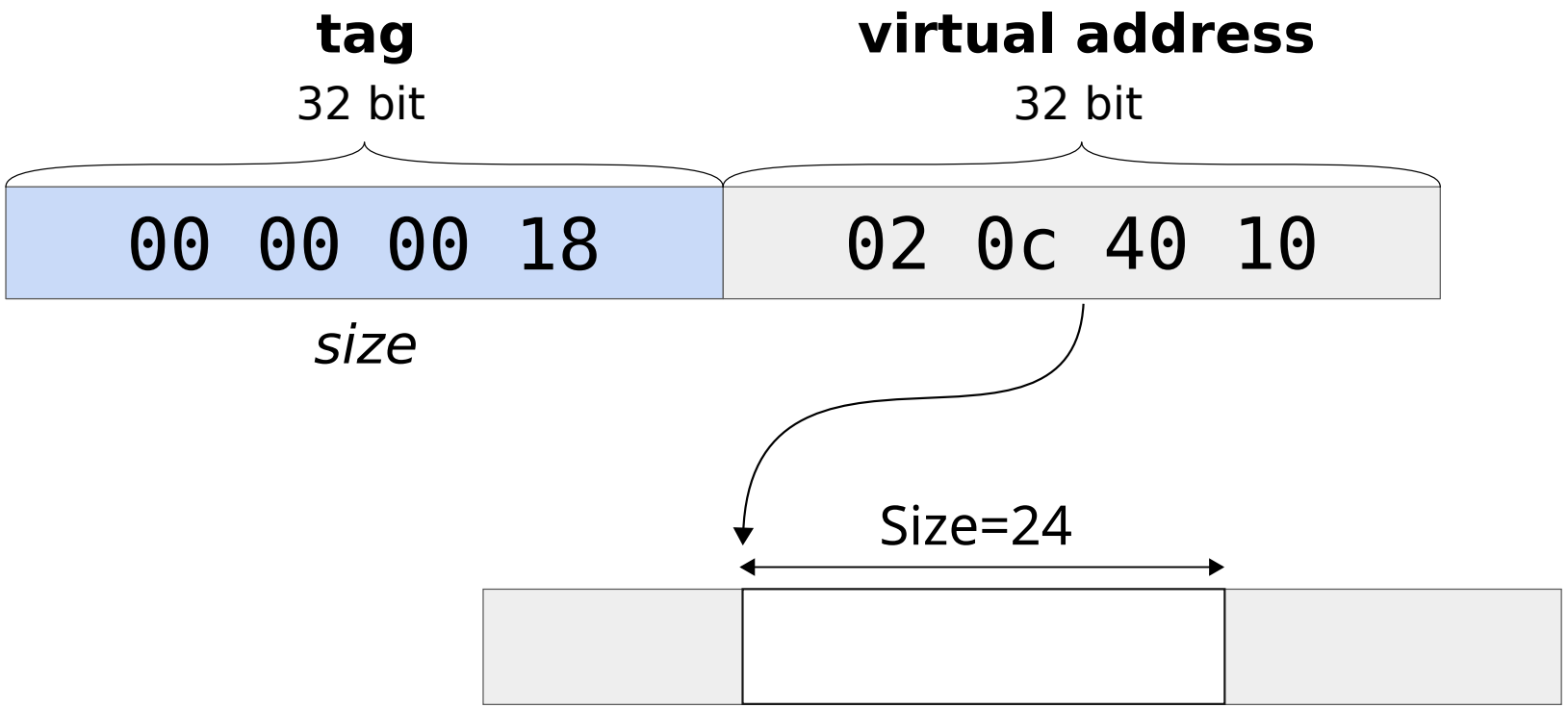
Tagged pointers



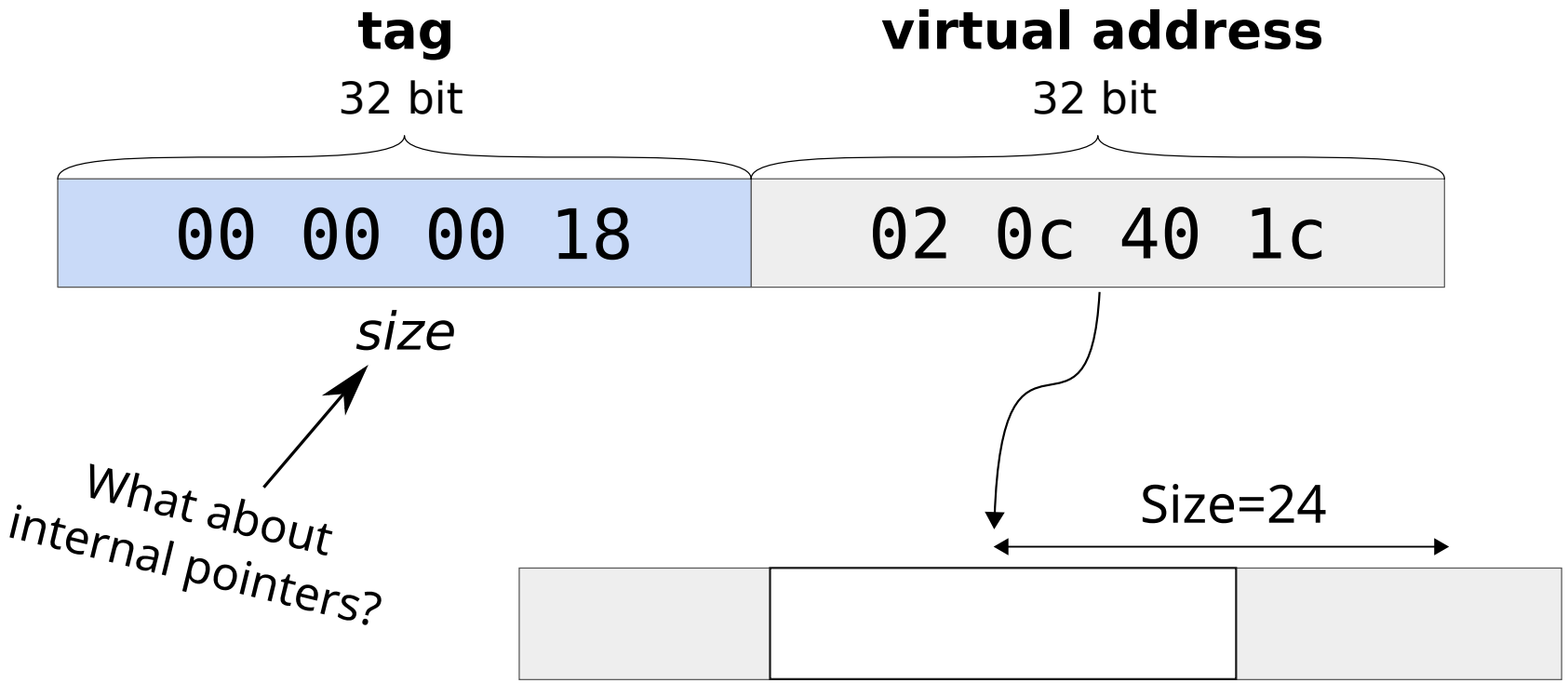
Tagged pointers



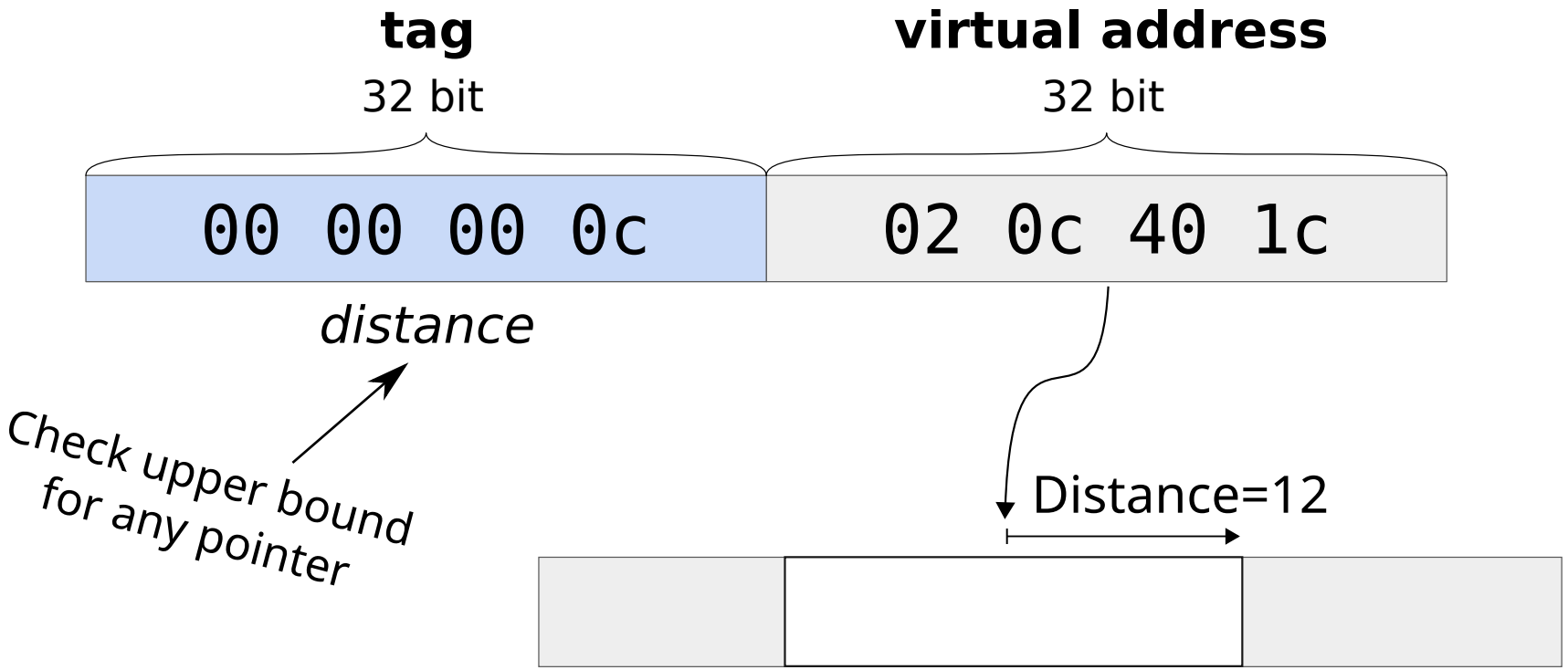
Delta Pointers



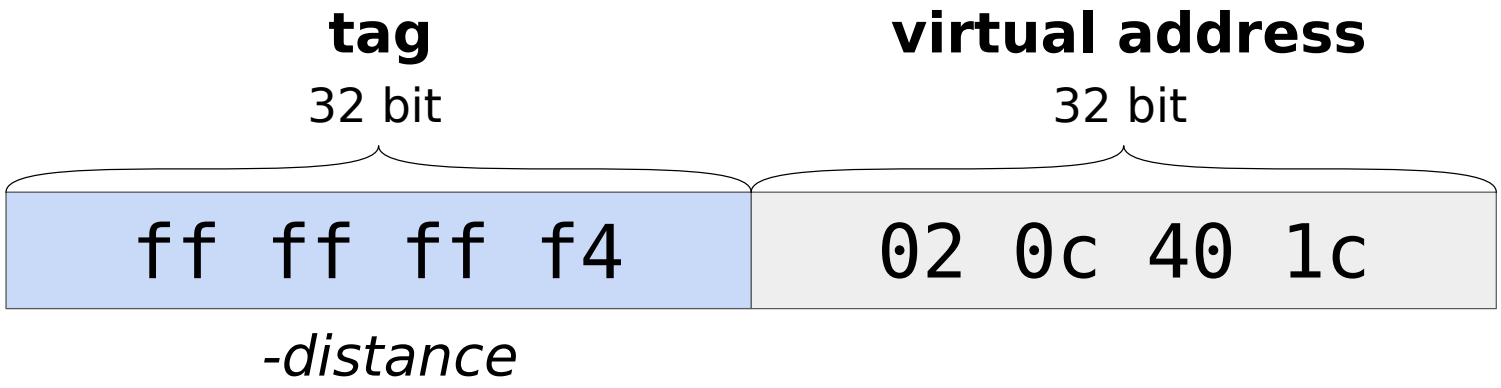
Delta Pointers



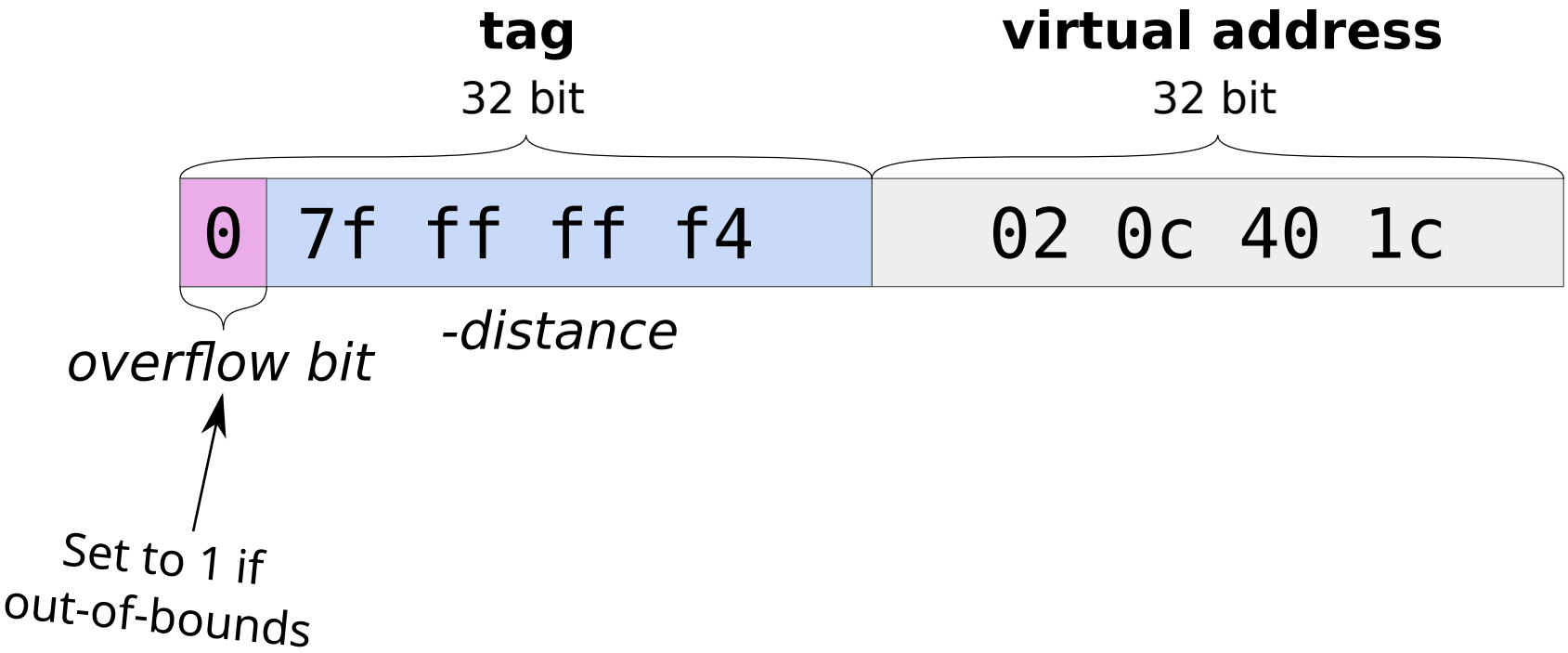
Delta Pointers



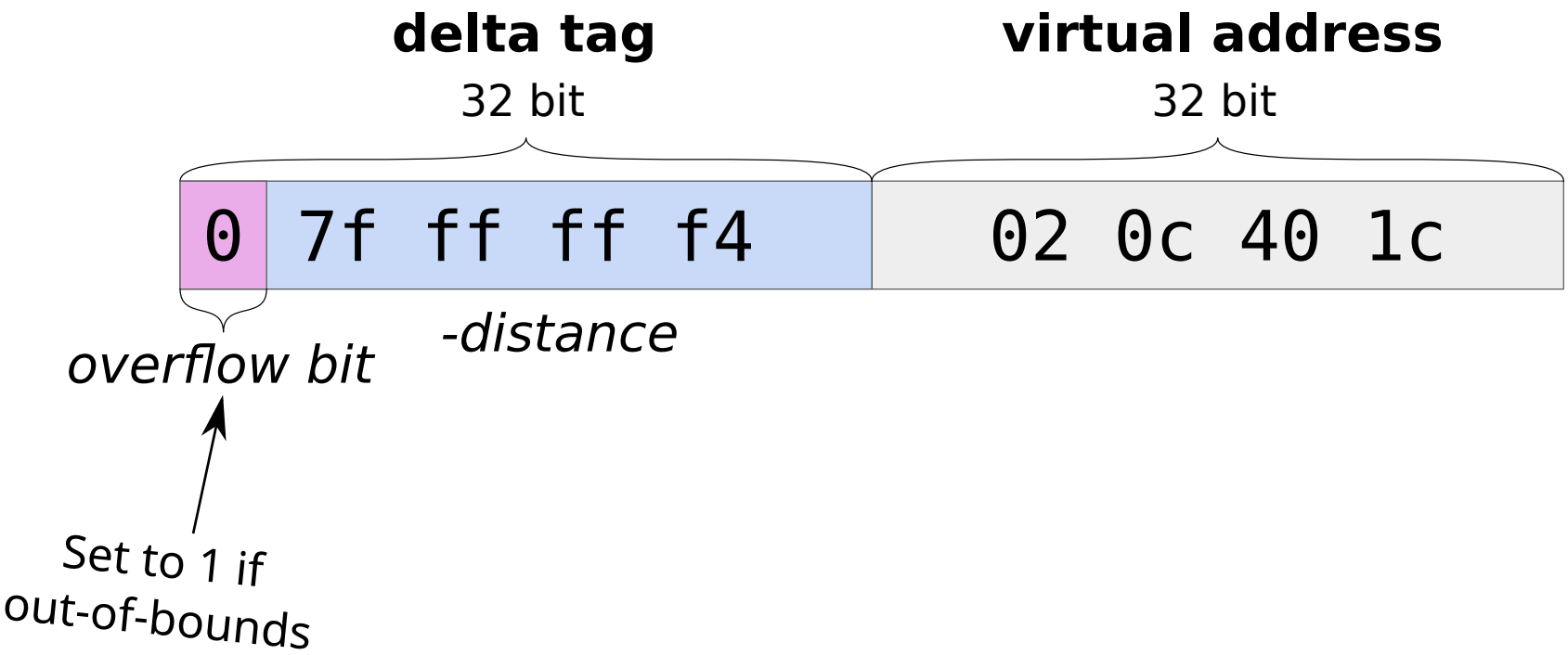
Delta Pointers



Delta Pointers



Delta Pointers



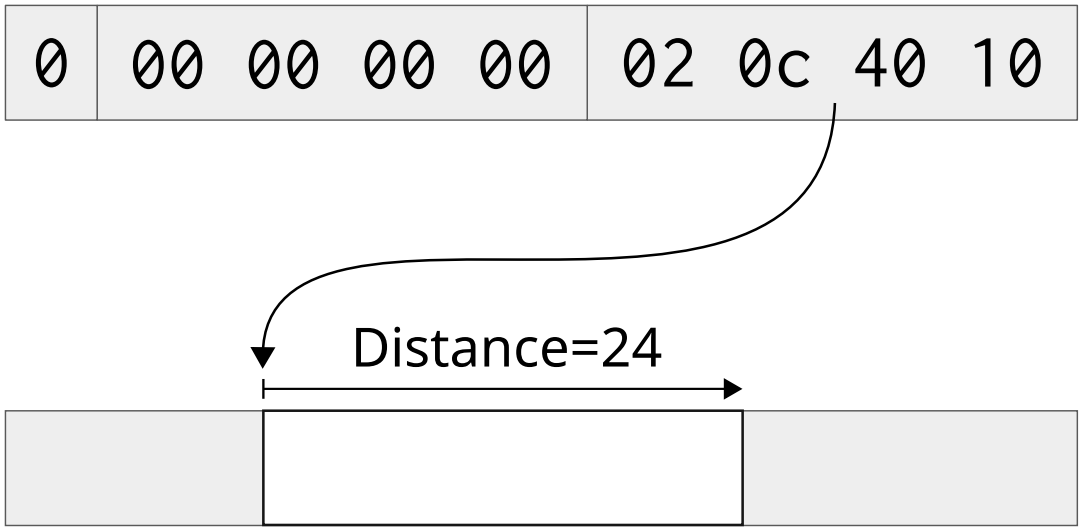
Instrumentation

```
char *p = malloc(24);
```

0	00 00 00 00	02 0c 40 10

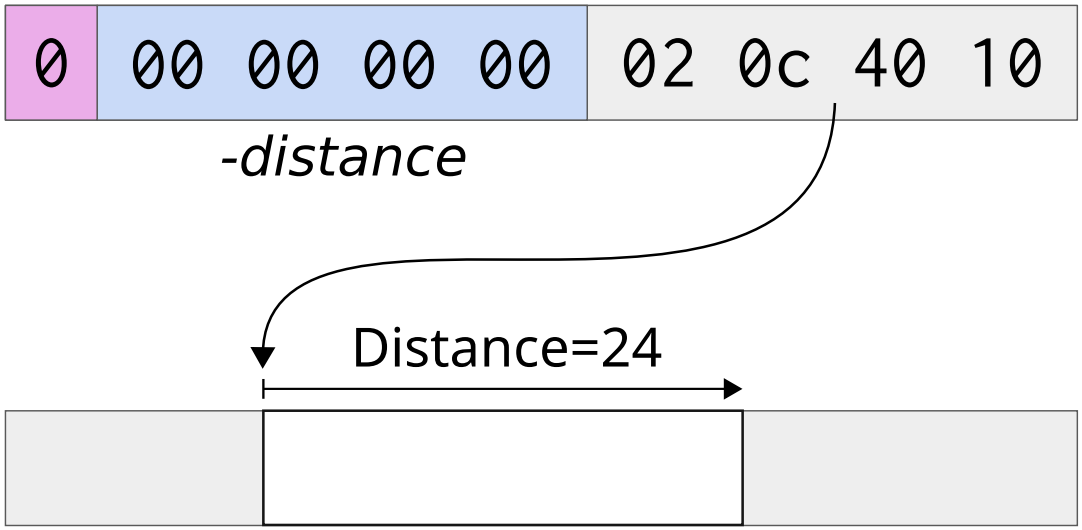
Instrumentation

```
char *p = malloc(24);
```



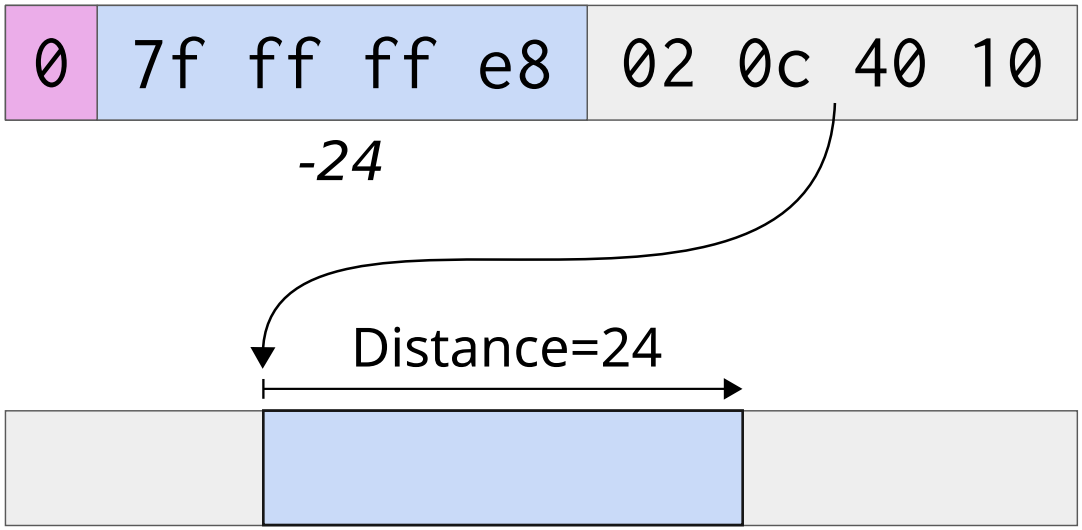
Instrumentation

```
char *p = malloc(24);
```



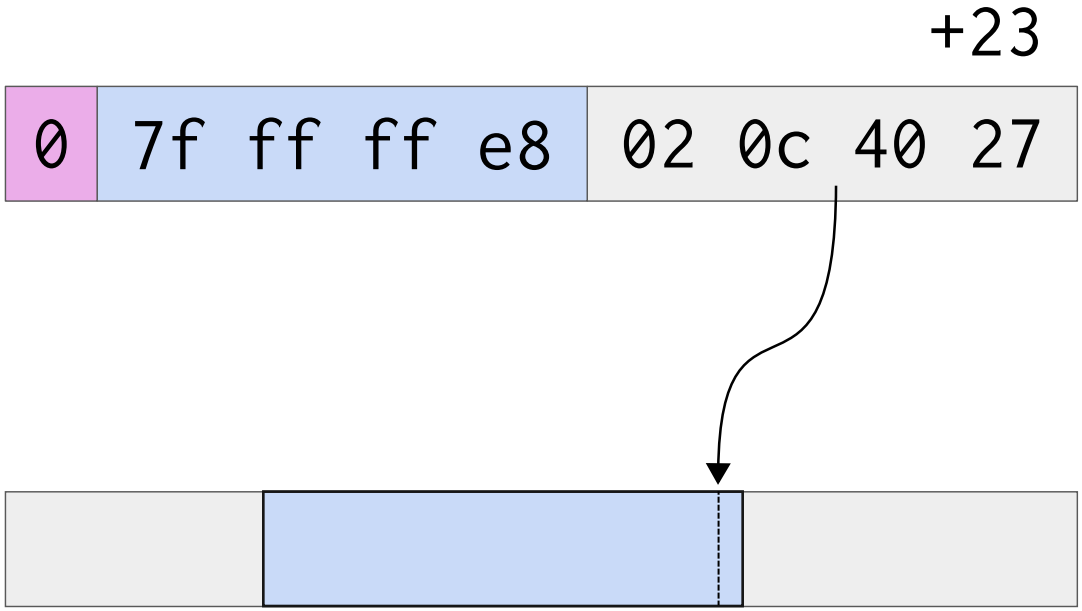
Instrumentation

```
char *p = malloc(24) | (-24 << 32);
```



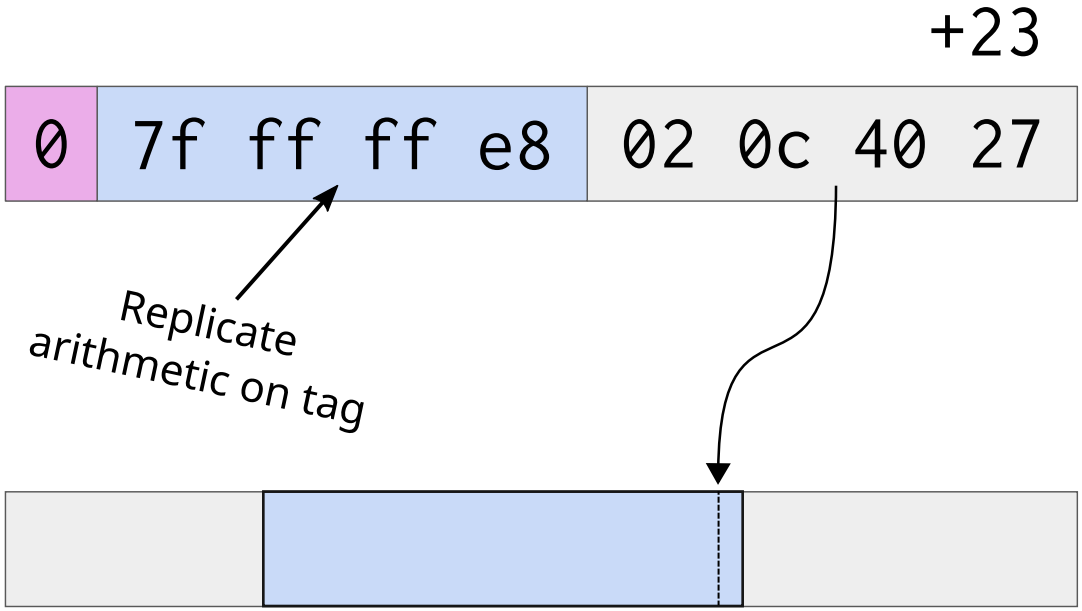
Instrumentation

```
p += 23;
```



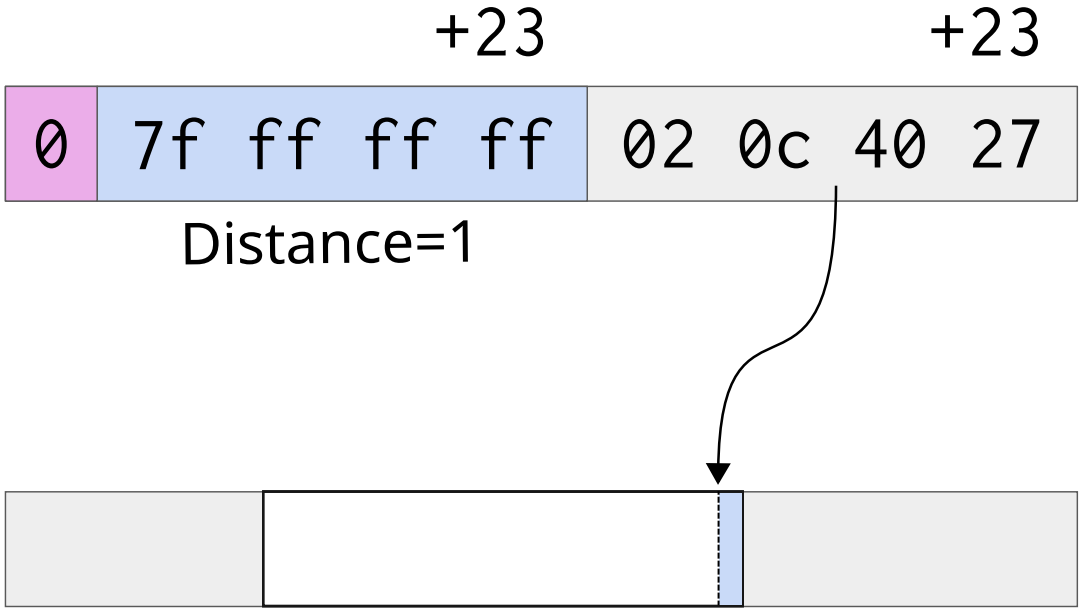
Instrumentation

```
p += 23;
```



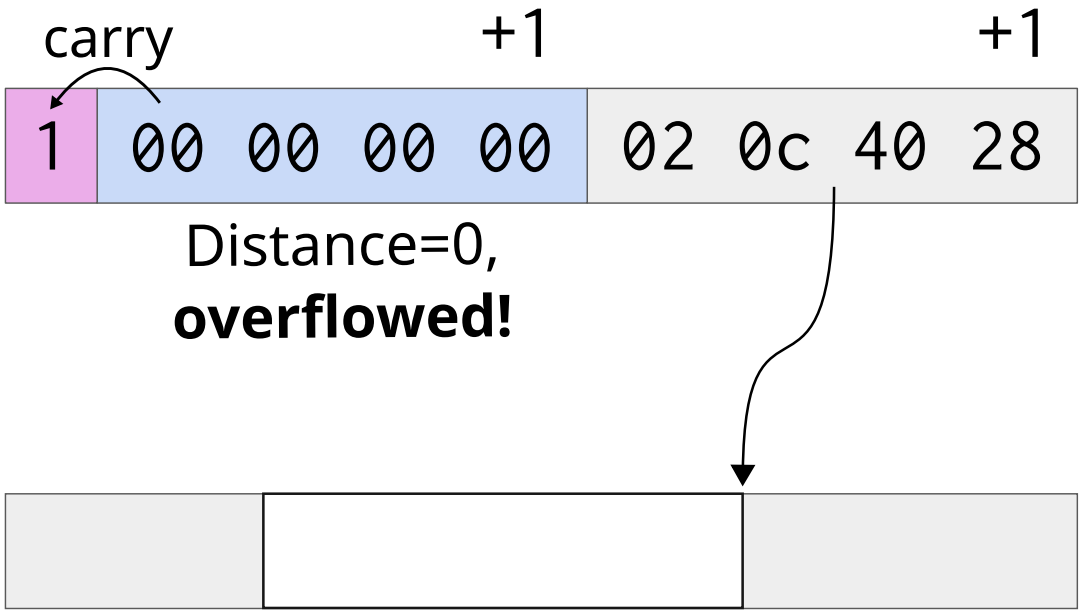
Instrumentation

```
p += 23 + (23 << 32);
```



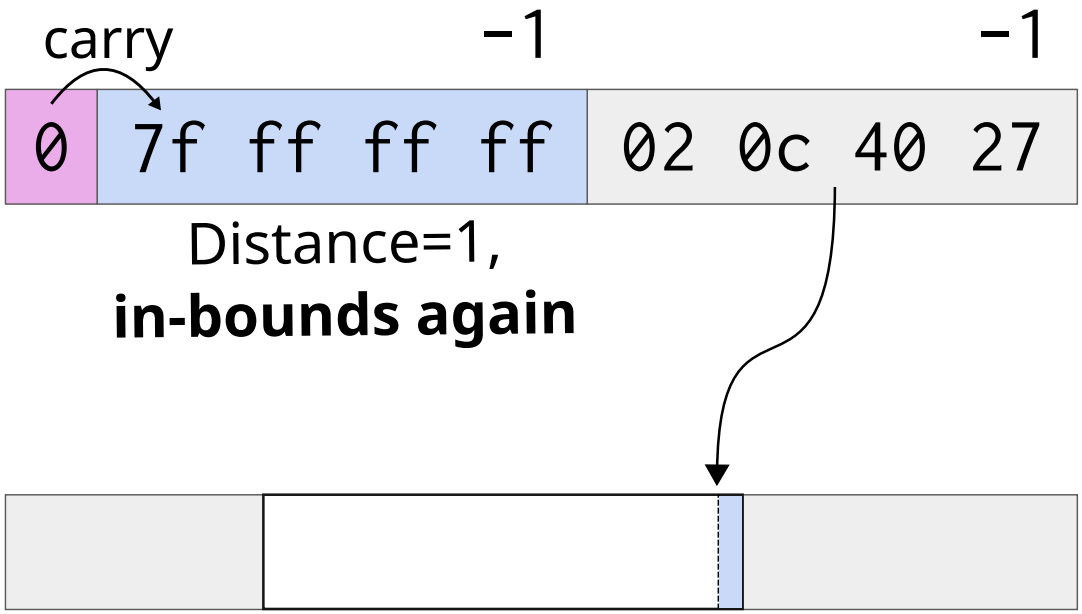
Instrumentation

```
p += 1 + (1 << 32);
```



Instrumentation

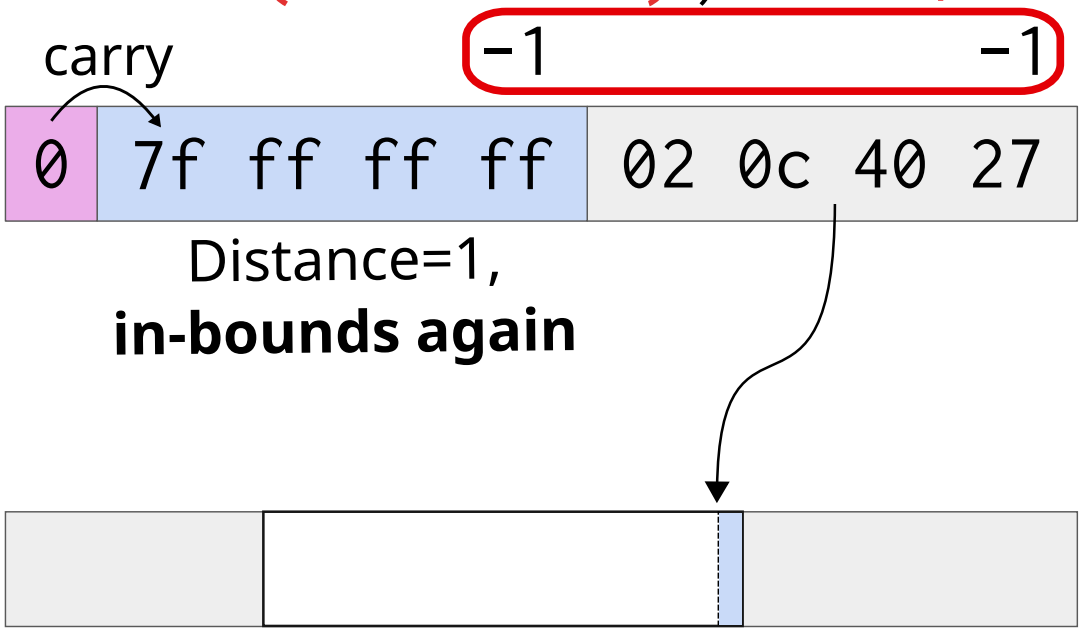
```
p += -1 + (-1 << 32);
```



Instrumentation

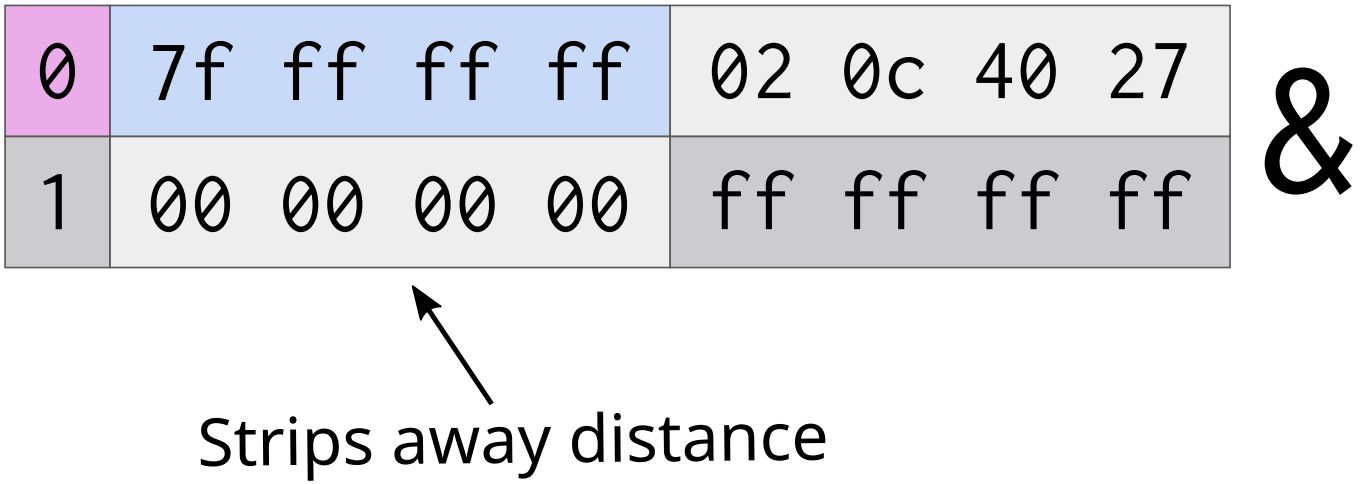
```
p += -1 + (-1 << 32);
```

one operation!

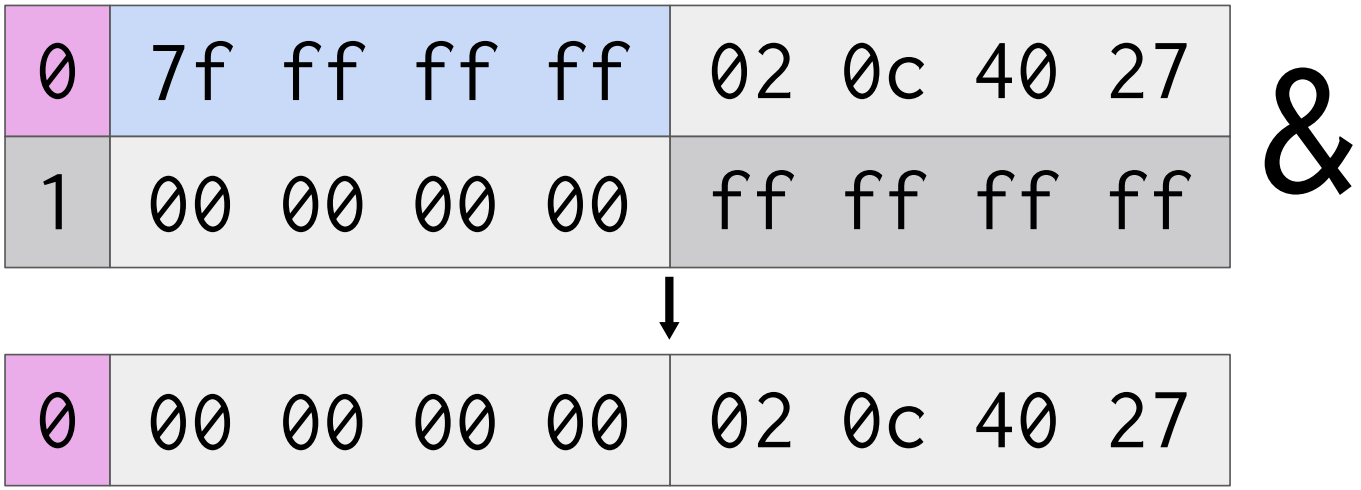


Dereferencing an in-bounds pointer

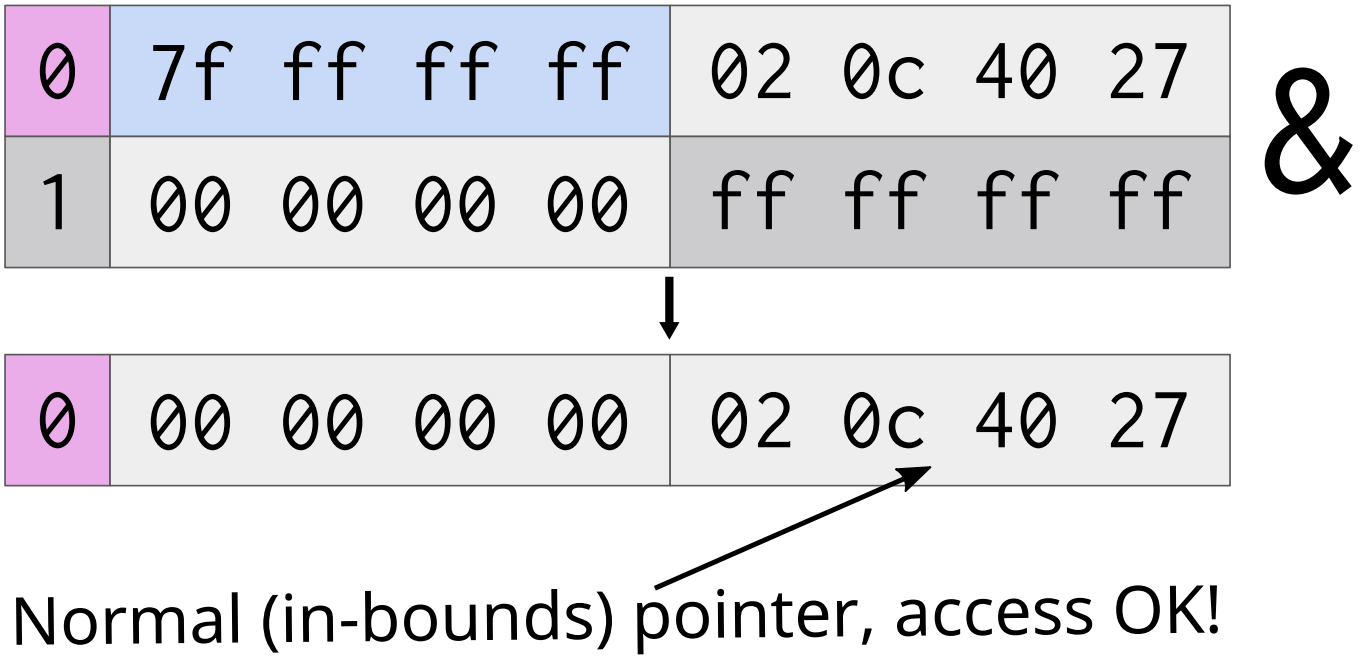
Dereferencing an in-bounds pointer



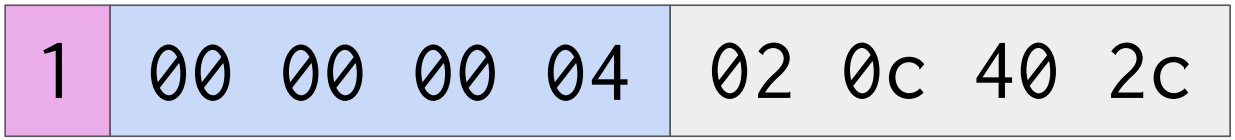
Dereferencing an in-bounds pointer



Dereferencing an in-bounds pointer



Dereferencing an out-of-bounds pointer

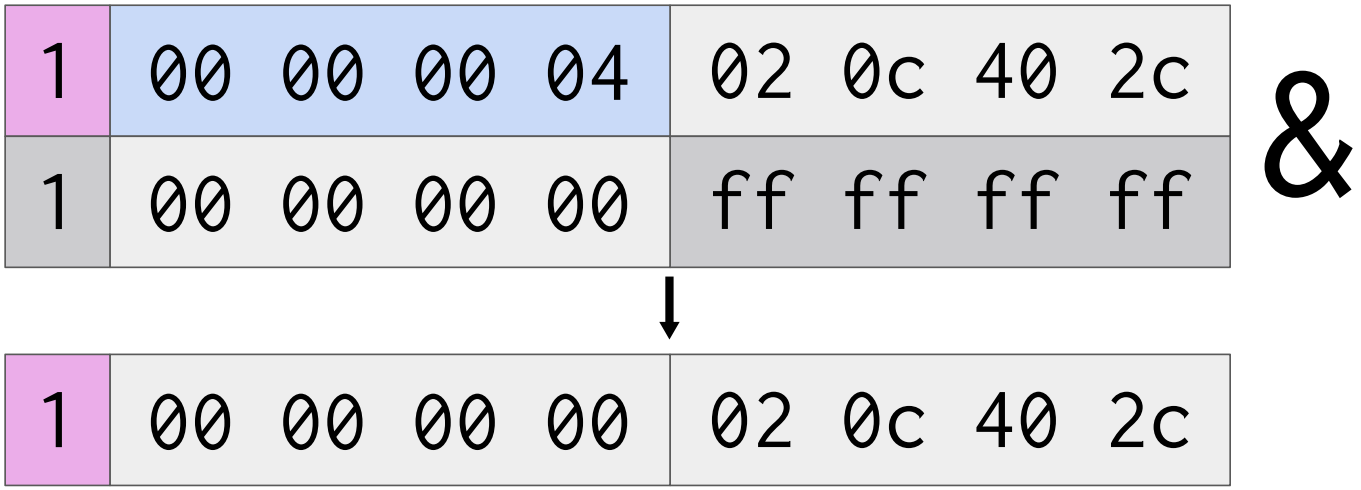


Dereferencing an out-of-bounds pointer

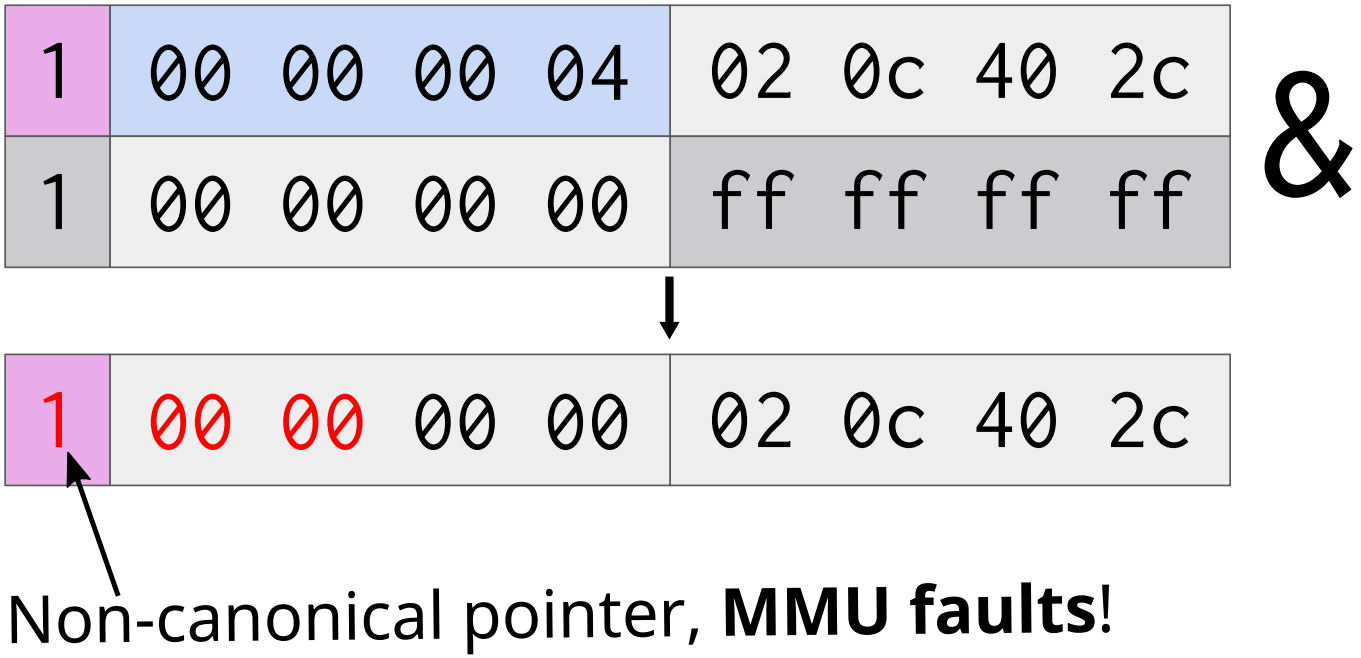
1	00 00 00 04	02 0c 40 2c
1	00 00 00 00	ff ff ff ff

&

Dereferencing an out-of-bounds pointer



Dereferencing an out-of-bounds pointer



Implementation

- LLVM based prototype for C/C++
- Stack + heap + globals
- 32-bit address $\rightarrow$ 4GB address space
- 31-bit distance $\rightarrow$ 2GB allocations
- Instrument NULL pointer with *distance* $= -1$
- Optimizations: omit instrumentation on in-bounds pointers

Pointer tagging breaks things

- Uninstrumented libraries

```
//strdup(ptr);
TAG(strdup(MASK(ptr)));
```

- Non-zero NULL pointer
- Subtraction, addition, multiplication, vectors, etc.
- Incomplete type information (e.g., unions)
- Compiler quirks
- . . . and more
  - Solved with TBAA + def-use chain analysis
  - Details in paper

Pointer tagging breaks things

- Uninstrumented libraries

```
//strdup(ptr);
TAG(strdup(MASK(ptr)));
```

- Non-zero NULL pointer
- Subtraction, addition, multiplication, vectors, etc.
- Incomplete type information (e.g., unions)
- Compiler quirks
- . . . and more
  - Solved with TBAA + def-use chain analysis
  - Details in paper

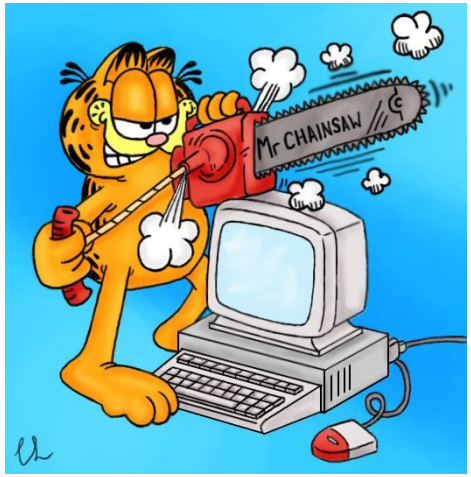
Pointer tagging breaks things

- Uninstrumented libraries

```
//strdup(ptr);
TAG(strdup(MASK(ptr)));
```

- Non-zero NULL pointer
- Subtraction, addition, multiplication, vectors, etc.
- Incomplete type information (e.g., unions)
- Compiler quirks
- . . . and more
  - Solved with TBAA + def-use chain analysis
  - Details in paper

Pointer tagging breaks things

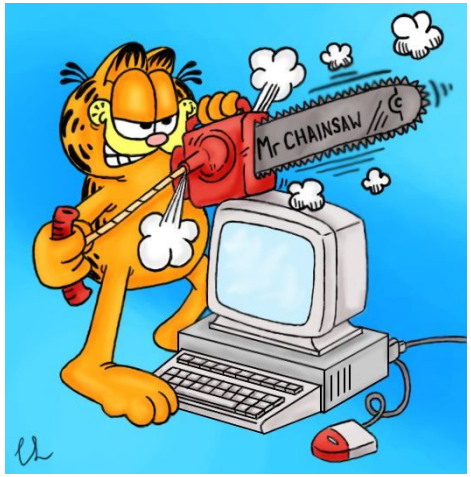


- Uninstrumented libraries

```
//strdup(ptr);
TAG(strdup(MASK(ptr)));
```

- Non-zero NULL pointer
- Subtraction, addition, multiplication, vectors, etc.
- Incomplete type information (e.g., unions)
- Compiler quirks
- . . . and more
  - Solved with TBAA + def-use chain analysis
  - Details in paper

Pointer tagging breaks things

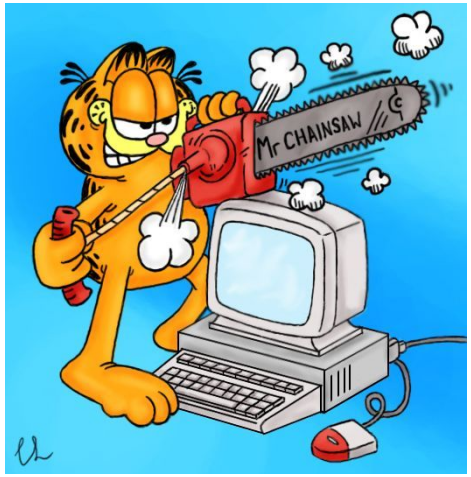


- Uninstrumented libraries

```
//strdup(ptr);
TAG(strdup(MASK(ptr)));
```

- Non-zero NULL pointer
- Subtraction, addition, multiplication, vectors, etc.
- Incomplete type information (e.g., unions)
- Compiler quirks
- . . . and more
  - Solved with TBAA + def-use chain analysis
  - Details in paper

Pointer tagging breaks things



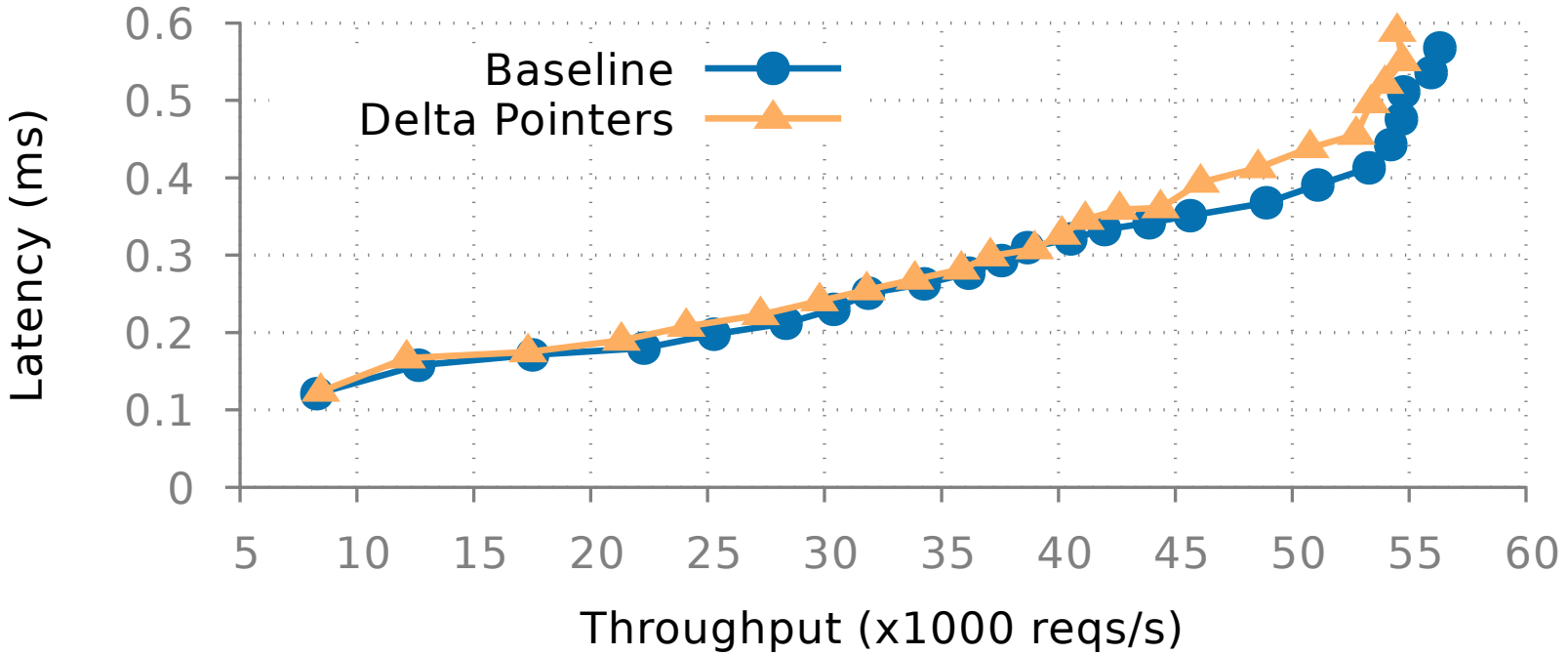
- Uninstrumented libraries

```
//strdup(ptr);
TAG(strdup(MASK(ptr)));
```

- Non-zero NULL pointer
- Subtraction, addition, multiplication, vectors, etc.
- Incomplete type information (e.g., unions)
- Compiler quirks
- . . . and more
  - Solved with TBAA + def-use chain analysis
  - Details in paper

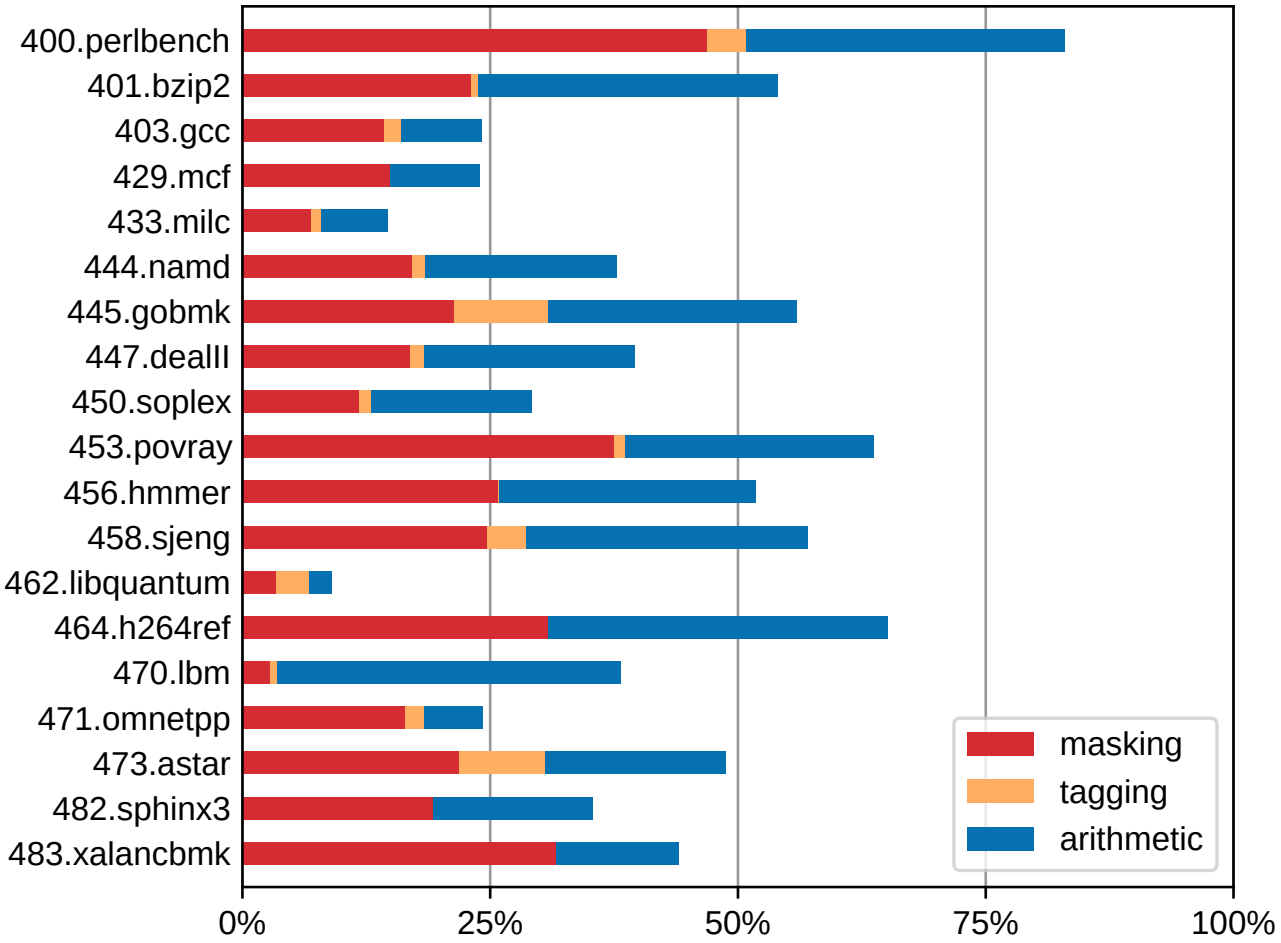
Evaluation

Nginx



3-6% (I/O bound)

SPEC CPU2006 (C/C++)



35% geomean with optimizations

Is that any good?

~~Is that any good?~~

Is it better than branches?

~~Is that any good?~~

Is it better than branches?

Branching implementation: 48% overhead

~~Is that any good?~~

Is it better than branches?

Branching implementation: 48% overhead $>$ 35%!

~~Is that any good?~~

Is it better than branches?

~~Branching implementation: 48% overhead $>$ 35%!~~

Yes

Conclusion

- Reliable pointer tagging implementation
- We can check (upper) bounds without checks
- Faster than existing solutions

`https://github.com/vusec/deltapointers`

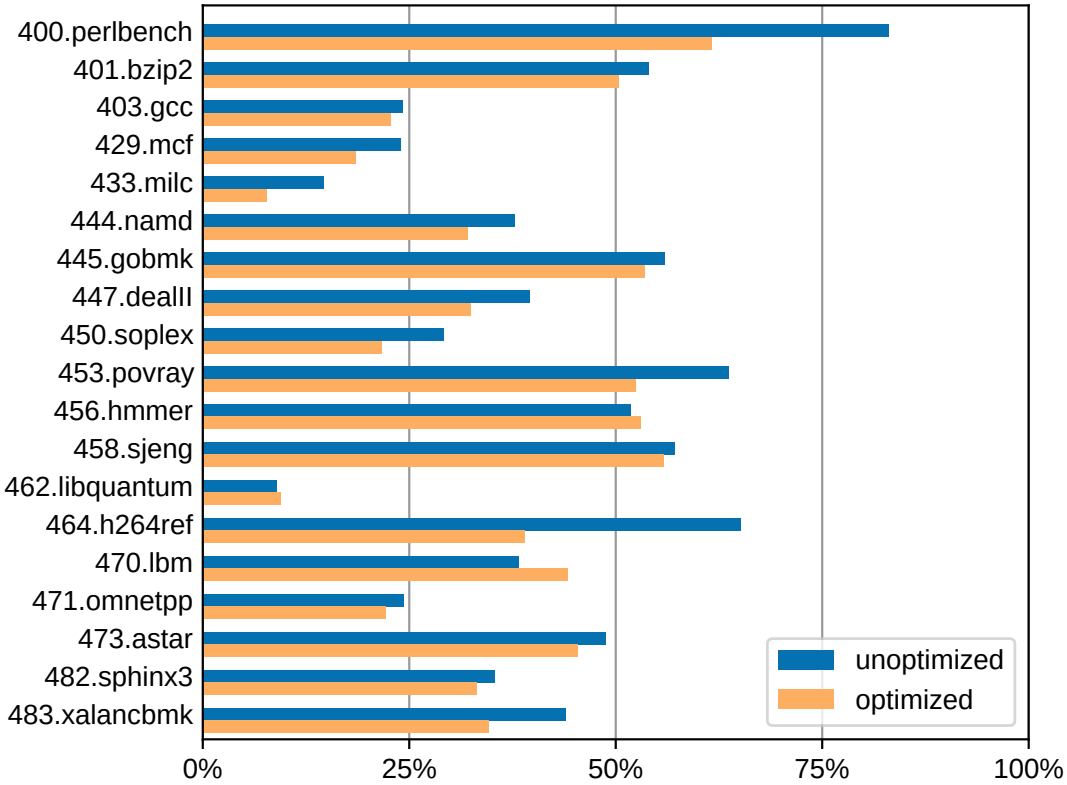
Related work

System	C++	Metadata	Checks	Passing OoB pointers	Non-linear	Runtime	Memory
Softbound	✗	Table	Deref	✓	✓	67%	64%
Baggy Bounds	✗	Layout	Arith	✓[a]	✓	72%	11%
PAriCheck	✗	Shadow	Arith	✓	✓[b]	96%	18%
LBC	✗	Shadow	Deref	✓	✗	22%	7.7%
ASan	✓	Shadow	Deref	✓	✗	80%	237%
Intel MPX	✓	Table	Deref	✓	✓	139%	90%
LowFat	✓	Layout	Deref	✗	✓	54%	5.2%
SGXBounds	✓	Tag	Deref	✓	✓	89%	0.1%
Delta Pointers	✓	Tag	—	✓	✓	35%	0%

[a] Only up to $alloc\_size/2$ on 32-bit.

[b] Unless wrap-around on 16-bit labels occurs.

Impact of optimization with static analysis



$41\% \Rightarrow 35\%$

Statistics

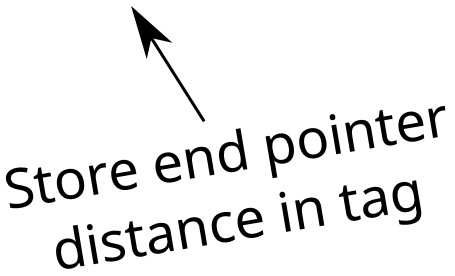
- 72% of SPEC offsets are dynamic
- 80% increase in code size with Delta Pointers

Branching implementation

```
void foo(int n) {
    char *buffer = malloc(24);
    char *p = buffer + n;
    *p = 'x';
}
```

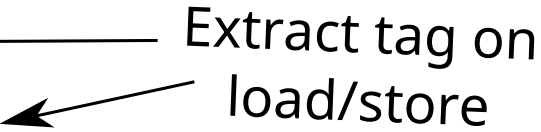
Branching implementation

```
void foo(int n) {
    char *buffer = malloc(24);
    buffer |= (buffer + 24) << 32;
    char *p = buffer + n;
    *p = 'x';
}
```

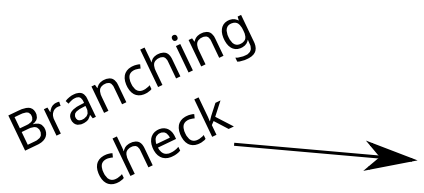


Branching implementation

```
void foo(int n) {
    char *buffer = malloc(24);
    buffer |= (buffer + 24) << 32;
    char *p = buffer + n;
    tag = p >> 32;
    p = p & 0xffffffff;
    *p = 'x';
}
```



Branching implementation




```
void foo(int n) {
    char *buffer = malloc(24);
    buffer |= (buffer + 24) << 32;
    char *p = buffer + n;
    tag = p >> 32;
    p = p & 0xffffffff;
    if (p >= tag)
        ERROR("overflow");
    *p = 'x';
}
```

Some pointer tagging challenges

- Some operations need masking to preserve semantics

```
char a[10];
//size_t len = &a[10] - &a[0];
size_t len = MASK(&a[10]) - MASK(&a[0]);
```

- Pointers that look like integers

```
union {
  char *buf;
  uint64_t foo;
} field;

field.buf += 42;  // should instrument
field.foo += 42;  // should NOT
```