Poster: L4 (Long Long Long Long) Pointer to Prevent Buffer Overflow*

Seong-Kyun Mok
The Division of Computer Convergence
Chungnam National University
Daejeon, Republic of Korea
mok7764@cnu.ac.kr

Eun-Sun Cho
The Division of Computer Convergence
Chungnam National University
Daejeon, Republic of Korea
eschough@cnu.ac.kr

***Abstract*—This paper introduces a new method to ensure spatial memory safety using vector operations supported by an existing architecture. The proposed scheme with "L4 Pointers" uses special 128-bit pointers to support the safety of pointer arithmetic operations. It uses vector operations in the commonly used architecture to encode the metadata of each memory chunk. We also introduce a prototype of the L4 pointer based on the LLVM. Experimental results show that our L4 pointer method shows better performance than existing spatial memory safety supports, while maintaining the atomicity of the pointer arithmetic operations.**

I. Introduction

Buffer overflows are well-known vulnerabilities of C/C++ programs. Although the security problems caused by buffer overflow seem too old to consider these days, they are still notorious [1]. Researchers and practitioners have investigated various approaches to support spatial memory safety to prevent buffer overflow, but some issues remain to be solved. First, the overhead of updating pointers with safety support is not negligible for practical usage. Second, updating pointers and bound information should be atomic, because they would lead to false negative errors in multi-threading environment, otherwise. For instance, Intel MPX [2] has a false negative problem in a multithreading environment because with the separate bound information table, it fails to support atomic updates of pointers and bound information. Third, finding an effective way to store the appropriate bound information is not straightforward. We call these various forms of bound information "metadata."

This study proposes a new method called L4 Pointers to prevent a buffer overflow, focusing on the above issues. The L4 Pointer uses similar approaches to fat [3] and delta pointers [4] to resolve these issues. A fat pointer is a pointer enlarged with the corresponding metadata such as bound information, while a delta pointer embeds metadata in the pointer itself without increasing the size of the pointer variable.

However, traditional fat pointers are limited in that they do not support the atomicity of the update operation; updating both the pointer and metadata, and performing memory bound checks, inherently require multiple operations, which will cause problems in multi-thread applications. To overcome this nonatomic update problem, recently proposed fat pointers used special operations of newly introduced hardware architectures. However, currently, such a new hardware has not been widely accepted, thus it is difficult to envision that this approach will be dominant in practice in the near future.

However, delta pointers support atomic updates of a pointer with its metadata. However, the delta pointer uses only the upper 32 bits of a pointer for metadata, while the lower 32 bits are used for a virtual address, which is not sufficient for either; 32 bits for bound information allow only upper bound checks, and 32 bits for a virtual address might cause compatibility problems.

The proposed L4 Pointer method mainly uses the delta pointer approach to support atomicity in pointer updates and multiple-thread. Based on 128-bit pointers, the L4 Pointer reserves sufficient space to store metadata and the pointer itself. Depending only on the vector operations and 128-bit registers that most hardware architectures support [5], we expect the L4 Pointer to be preferred in real-world usage. Basically, our contributions include the following.

- This paper presents the design and implementation of prototype of L4 Pointer, which provides sufficient meta data storage space, to support both upper and lower bound checks against buffer overflows.
- The proposed L4 Pointer provides atomic operations of the pointer and metadata.
- The proposed L4 Pointer works on commonly used architectures such as Intel processors and ARM processors. The target architecture of the prototype of L4 Pointer is Intel x86.

II. L4 Pointer

To use the L4 Pointer, a normal pointer of 64 bits is expanded to consecutive 128 bits. Metadata is stored in the upper 64 bits, of which the upper and low bounds evenly share. It is significantly similar to delta pointers [4], except that the metadata of a delta pointer is 32 bits long, which is too short to hold both the upper and lower bounds. To handle 128-bit long pointers and manipulate metadata and pointers atomically, the L4 pointer uses vector operations and vector registers

This work was supported by Institute for Information communications Technology Planning & Evaluation(IITP) grant funded by the Korea government (MSIT)(No.2019-0-01343, Training Key Talents in Industrial Convergence Security)

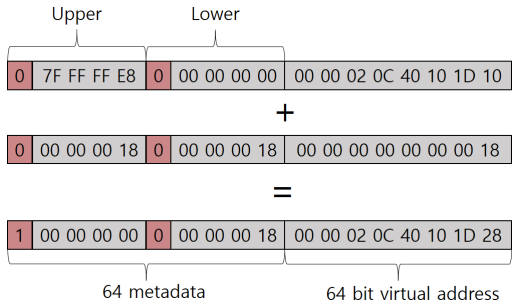

Fig. 1. Layout of L4 Pointer and process of pointer arithmetic in L4 Pointer

```
UPPER_BOUNDS' = UPPER_BOUNDS & 8000000000
LOWER_BOUNDS' = LOWER_BOUNDS & 8000000000
BOUNDS' = UPPER_BOUNDS' | LOWER_BOUNDS'
TAG_RESULT = BOUNDS' << 32
RESULT_ADDRESS = VIRTUAL_ADDRESS | TAG_RESULT
REG = LOAD RESULT_ADDRESS
```

Fig. 2. Process of LOAD operations in L4 Pointer

[5]. Fortunately, the most commonly used architectures such as X86 and ARM provide these facilities. More details are provided below.

A. Layout of L4 Pointer

The layout of L4 pointer is illustrated in Fig. 1. As aforementioned, the upper 64 bits of the L4 pointer are used to store metadata, whereas the lower 64 bits are used for the virtual address. The upper 64 bits are divided in half, the upper 32 bits are used for upper-bound storage, and the lower 32 bits are used for lower-bound storage. In each of the 32 bits, bound information is stored only in 31 bits, and the highest 1 bit is used as a flag.

B. Pointer Arithmetic

We assumed that the vector operations and 128-bit registers are provided by the architecture to enable the operation of the 128-bit pointers. For instance, an integer value is added to a pointer (like 'p+0x18' where p is a pointer), the integer value (for instance, 0×18 in 'p+0x18') is also expanded to 128 bits to align to the 128-bit long pointer p. Note that the value to be added is copied to the upper and lower bounds as well as to the virtual address area. Because the newly created L4 Pointer cannot load/store from/to the address by itself, conversion between an L4 Pointer and a normal pointer is necessary, as depicted in Fig. 2. If overflow/underflow occurs, the highest bit of 'RESULT_ADDRESS' is set to 1, and the memory management unit (MMU) generates an exception.

C. Example of L4 Pointer

The pointer arithmetic operations use special registers. The prototype of the L4 Pointer is currently operating on an Intel x86 [5]. Special registers called the XMM registers of Intel x86 are 128-bit long, and used for vector or float calculations. Fig. 3 shows the pointer operation process in the

```
movaps XMM1, XMMWORD PTR [rbp-0x20]
movaps XMM0, 0x10000000100000000000000001
paddq  XMM1, XMM0
movaps XMMWORD PTR [rbp-0x30], XMM1
```

Fig. 3. Example assembly codes that adds an offset to a pointer in Intel x86 Architecture used L4 Pointer

TABLE I
THE RESULT OF DEDUP OF PARSEC FOR L4 POINTER

	The Number of Threads			
	1	2	3	4
L4 Pointer	26.2754s	14.1814s	7.7172s	5.2550s
Baseline	26.1704	14.2386s	7.6964s	5.2376s

x86 instruction. The instructions 'movaps' and 'paddq' are used instead of 'mov' and 'add' and are used for the XMM registers. The calculated value was then set in the XMM0 register, and the next operation is performed using the 'paddq' instruction. The result of the operation is stored in the XMM1 register and memory.

III. IMPLEMENTATIONS

Currently, the prototype of the L4 pointer is implemented using the LLVM 12.0.0 [6]. However, it cannot support double pointer and structure members that are of the pointer type yet, which we are currently working on. We use the XMM registers and related operations in the Intel x86 architecture.

IV. EXPERIMENTS AND CONCLUSIONS

We measured the performance overhead by running the Dedup PARSEC benchmark package [7]. We used Intel i9-7900X machines with 10 cores at 3.30 GHz and 64 GB of memory. The results are presented in I, depicting that the L4 Pointer does not incur significant overhead during execution. Currently, it does not cover all pointers, which means that even a smaller overhead is expected in the future.

REFERENCES

[1] M. Miller, "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape," presented at the BlueHat IL., February 2019.

[2] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack", Proceedings of the ACM on Measurement and Analysis of Computing Systems, 2(2):28, 2018

[3] D. Song et al., "SoK: Sanitizing for Security," 2019 IEEE Symposium on Security and Privacy (SP), 2019, pp. 1275-1295, doi: 10.1109/SP.2019.00010.

[4] T. Kroes et al., "Delta pointers: buffer overflow checks without the checks," in European Conference on Computer Systems (EuroSys), 2018, pp. 1-14, doi:10.1145/3190508.3190553

[5] Intel 64 and IA-32 Archtectures Software Developer's Manual Volume 1: Basic Architecture, Intel, Accesed: September 2016, Available: https://www.intel.com/content/dam/www/public/us/en/documents /manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf

[6] The LLVM Compiler Infrastructure, LLVM, [Online]. Available: https://llvm.org/

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in the 17th international conference on Parallel architectures and compilation techniques (PACT '08), 2008, pp 72–81. doi:10.1145/1454115.1454128