In-Fat Pointer: Hardware-Assisted Tagged-Pointer Spatial Memory Safety Defense with Subobject Bound Granularity Protection

Shengjie Xu, Wei Huang, and David Lie

University of Toronto

1. Motivation

Unsafe programming languages like C and C++ do not strictly enforce memory safety, and may allow an attacker to exploit defects in pointer arithmetic to cause unintended memory corruption or disclosure. One important aspect is *spatial* memory safety, where the pointer points to an unintended memory location because of bad pointer arithmetic. When the bad pointer arithmetic only moves the pointer in the same top-level object, it can still cause *intra-object overflow* if the result address points to the memory of a different subobject (e.g., struct member in C), which is harder to detect but equally dangerous. These vulnerabilities can be exploited by attackers to become a powerful primitive for further attacks, for example, code reuse attacks [12] and the more recent data-only attacks [5]. Currently, no existing or proposed defense can comprehensively provide fine-grained checking, low overhead, and binary compatibility at the same time.

2. Limitations of the State of the Art

While many solutions have been proposed to improve memory safety for C and C++ code, some schemes do not protect against *intra-object overflow*, which allows a buffer overflow inside a struct member to overwrite the memory of other members in the same struct. Fine-grained memory safety defenses aiming to catch these overflows will need to associate a pair of legitimate address as pointer bounds to each pointer subject to bad arithmetic [14].

Traditional *fat pointer* based approaches associate the pointer bounds to each pointer persistently throughout the lifetime of the pointer [11, 16, 8]. Such schemes either incur high overhead if the bounds are maintained in a separate memory region, or are incompatible with legacy code if the pointer bounds are stored inline with the pointers. They are also common to have memory overhead of 2× or higher [11].

Recent *tagged-pointer* schemes store metadata bits, known as *tags*, in the unused high address bits on pointers. Prior works have explored using pointer tags to lookup in-memory object metadata and then *recomputing* pointer bounds on-demand, which avoids the high memory overhead [9, 1, 2, 6]. However, none of the existing hardware-based schemes can both detect intra-object overflow and fit all the per-pointer metadata on the tag without increasing pointer size. EffectiveSan [2] is a software-only sanitizer that achieves both but has a high performance overhead of 115% for bound checking.

3. Key Insights

For a tagged pointer scheme using object metadata, in order to detect intra-object overflow, the recomputed bound should have *subobject* granularity. This means that when a pointer to a subobject is derived from a pointer to the parent, the bound on the derived pointer must be *narrowed* to the memory range of the subobject. We propose In-Fat Pointer which solves the following two challenges:

- Develop means for object metadata lookup schemes to consume less pointer tag bits so that In-Fat Pointer can encode information about current subobject.
- Develop an efficient approach to compute the most narrow bounds to the current subobject with help from additional pointer tag bits

The first insight is that by including multiple metadata schemes, each of which is *designed* for certain objects in the program, object metadata lookup can take fewer pointer tag bits. Previous works using tagged pointers only use a single metadata scheme general enough to accommodate all possible objects. This insight guides the design of three metadata schemes in In-Fat Pointer which solves the first challenge.

The second insight is that carrying information on pointer representing the current subobject can significantly simplify subobject bound recomputation. EffectiveSan [2] needs to perform a search on possible subobjects to retrieve the correct bound. We find that by storing a *subobject index* on pointer and updating it when deriving pointer to subobjects, the retrieval of the current subobject can be simplified. This insight guides the design of layout table in In-Fat Pointer which solves the second challenge.

With the two challenges solved by two insights above, In-Fat Pointer can achieve finer-grained protection without increasing pointer size than previous tagged-pointer schemes.

4. Main Artifacts

In-Fat Pointer is an instruction set extension and compiler instrumentation that provides spatial memory safety at subobject granularity. We implemented In-Fat Pointer prototype on CVA6 [17], a 64-bit in-order RISC-V processor, and evaluated it on an FPGA board. We modified Linux kernel to support additional user-level states introduced by In-Fat Pointer, and proper handling of pointer tags when a pointer from a user-level program is passed to the kernel. We implemented the compiler instrumentation on Clang/LLVM [7].

As introduced in Section 3, the ISA is extended with three metadata schemes and layout tables for object metadata access and subobject bound computation. The hardware provides a new `promote` instruction to recompute the pointer bound from a pointer value, which first determines the metadata scheme from the pointer tag, then accesses the object metadata according to the scheme design, and finally computes the pointer bounds. The compiler instruments the program so that each pointer subject to bad pointer arithmetic has an accompanying bound, and a bound check is performed before the pointer is dereferenced.

In-Fat Pointer introduces the following three metadata schemes, which trade-off placement constraints, object size constraints, and scalability. Constraints can limit compatibility with certain types of objects, while scalability limits the number of objects that can be supported. We reserve 2 bits from a 16-bit tag to indicate the scheme and 2 *poison bits* to track the validity of the pointer.

The Local Offset scheme is designed for small objects that have object placement constraints, such as stack-allocated objects. It appends object metadata directly after the object, aligns both of them at power-of-2 sized *granule*, and for each pointer it stores the distance from the current address to the metadata address in terms of multiple of granule size on the pointer tag. Therefore, 6-bit offset on pointer tag and 16-byte granule size can support objects at most ~1KB in size.

The Subheap scheme supports heap allocations. It requires the memory allocator to group objects with the same type and size into power-of-2 sized blocks, and share a common metadata per block among all objects in the block. The prototype uses 4 bits from the pointer tag to locate the common metadata, with the help of 16 control registers specifying the block size and offset of metadata that is indexed by the 4 bits.

The Global Table scheme is a fallback scheme that supports objects that the other two schemes cannot support due to constraints. Each object is assigned a row in a global metadata table, and the index is stored in the pointer tag. The size of the table is limited by the width of the index, which is 12 bits in our prototype.

After one of the metadata schemes is accessed and the object granularity bound computed, the hardware executing `promote` instruction will access the layout table to narrow the bound to the currently pointed subobject. The layout table encodes the type hierarchy as a table, and a *subobject index* from pointer tag points the element in the table corresponding to the subobject, which the hardware uses to retrieve the narrowed subobject bound.

We synthesized the modified CVA6 processor on an FPGA board and (1) run all 5,572 applicable test cases from the NIST Juliet test suite for C/C++ [10] to test the functionality; (2) run 4 benchmarks to evaluate the performance and area overhead: bzip2 [13], 458.sjeng from SPEC2006 [4], CoreMark [3], and WolfCrypt's Diffie–Hellman benchmark [15]; (3) evaluate the area overhead by analyzing the FPGA synthesis reports.

5. Key Results and Contributions

In-Fat Pointer shows that it is practical for hardware-based tagged-pointer scheme utilizing object metadata to provide spatial memory safety at subobject bound granularity without breaking compatibility by increasing pointer size. It incurs performance overhead from 8.9% to 23.1% and memory overhead less than 17.2% across four benchmarks. In-Fat Pointer detects all 5,572 applicable vulnerabilities in the Juliet test suite with full accuracy.

Comparing with existing tagged-pointer schemes, In-Fat Pointer is the first hardware design that can achieve subobject granularity protection on par with fat pointers without breaking compatibility with legacy code. Comparing with existing fat pointer schemes that store pointer bounds in separate memory regions, In-Fat Pointer has a lower overhead.

This paper makes the following contributions:

- We present In-Fat Pointer, which uses three complementary metadata schemes that provide comprehensive, legacy code-compatible, and performance-efficient protection for hardware-based spatial memory safety defenses against memory corruption.
- We describe layout tables, a mechanism that In-Fat Pointer uses to provide protection against intra-object overflows by narrowing the bounds that pointers to subobjects within an object are checked against, so they are the most precise bounds possible.
- We implement a In-Fat Pointer prototype on an FPGA board. We evaluate its ability to detect memory safety violations using the Juliet test suite, and performance and memory overhead against a set of application benchmarks.

6. Why ASPLOS

In-Fat Pointer is a memory hardware-software safety proposal that and involves modifications and innovations in the processor architecture, compiler, and operating system. It illustrates how hardware and software can be co-designed to achieve results that were previously impossible.

7. Citation for Most Influential Paper Award

This paper presents In-Fat Pointer, the first hardware-accelerated tagged-pointer scheme that can achieve spatial memory safety at subobject granularity while maintaining compatibility with legacy code and imposing low performance overhead. In-Fat Pointer encodes both object metadata location and currently pointed subobject in pointer tags, and backs them with three complementary metadata schemes for deriving object metadata from the pointer tag. The work shows that this approach provides practical fine-grained spatial memory safety protection on par with fat pointers, but with full compatibility with legacy code and acceptable performance overhead.

References

[1] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. CUP: Comprehensive user-space protection for C/C++. In *Proceedings of the ACM ASIA Conference on Computer & Communications Security 2018*, ASIACCS 18, Incheon, Korea, June 2018.

[2] Gregory J. Duck and Roland H. C. Yap. EffectiveSan: Type and memory error detection using dynamically typed c/c++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 18, Philadephia, PA, June 2018.

[3] Shay Gal-On and Markus Levy. Exploring coremark a benchmark maximizing simplicity and efficacy. *The Embedded Microprocessor Benchmark Consortium*, 2012.

[4] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[5] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Porceedings of the 37th IEEE Symposium on Security and Privacy*, Oakland 16, San Jose, CA, May 2016.

[6] Gnanambikai Krishnakumar, Kommuru Alekhya REDDY, and Chester Rebeiro. ALEXIA: A processor with lightweight extensions for memory safety. *ACM Trans. Embed. Comput. Syst.*, 18(6), November 2019.

[7] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[8] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. WatchdogLite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 14, Orlando, FL, Feburary 2014.

[9] Myoung Jin Nam, Periklis Akritidis, and David J Greaves. FRAMER: A tagged-pointer capability system with memory safety applications. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC 19, San Juan, Puerto Rico, December 2019.

[10] NIST. Juliet test suite for C/C++. Accessed on: Dec 12, 2020. [Online]. Available: https://samate.nist.gov/SRD/testsuite.php.

[11] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX explained: A cross-layer analysis of the Intel MPX system stack. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, SIGMETRICS 18, Irvine, CA, June 2018.

[12] M. Prandini and M. Ramilli. Return-oriented programming. *IEEE Security Privacy*, 10(6):84–87, 2012.

[13] Julian Seward. bzip2: Home. Accessed on: Dec 12, 2020. [Online]. Available: https://www.sourceware.org/bzip2/.

[14] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *Porceedings of the 34th IEEE Symposium on Security and Privacy*, Oakland 13, San Francisco, CA, May 2013.

[15] WolfSSL Inc. Wolfcrypt embedded crypto engine. 2020. Accessed on: Dec 12, 2020. [Online]. Available: https://www.wolfssl.com/products/wolfcrypt-2/.

[16] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture*, ISCA 14, Minneapolis, MN, June 2014.

[17] Florian Zaruba and Luca Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, Nov 2019.