Poster: BSan: A Non-Intrusive and Comprehensive Binary-Level Memory Sanitizer

Yang Li[1], Yuan Li[2], Chenyang Li[1], Xinhui Han[1], Chao Zhang[3]
[1]Peking University, [2]Zhongguancun Laboratory, [3]Tsinghua University

***Abstract*—Memory safety vulnerabilities remain a critical threat to software security. Compiler-based sanitizers like AddressSanitizer (ASan) and HWASan require source code and recompilation, altering the program's memory layout. Existing binary-level solutions like QASan only support heap detection. We present BSan, a non-intrusive binary-level memory sanitizer using QEMU's dynamic binary translation. BSan detects heap vulnerabilities without external information, and extends to stack and global variables when debug information is available. Our evaluation demonstrates detection capabilities comparable to HWASan.**

I. Introduction and Motivation

Memory safety vulnerabilities, including buffer overflows and use-after-free (UAF) bugs, continue to be the root cause of numerous security exploits. According to Microsoft [1] and Google [2], approximately 70% of their security vulnerabilities stem from memory safety issues. Recent research shows that non-linear memory accesses (e.g., `buf[n] = x`) have surpassed linear accesses in heap corruption vulnerabilities, making traditional boundary-checking sanitizers less effective.

Existing memory sanitizers such as ASan [3] and HWASan [4] have proven highly effective at detecting these vulnerabilities during development. However, they share fundamental limitations: **(1) they require source code access and recompilation**, which alters the program's memory layout and may mask certain bugs; **(2) HWASan relies on hardware memory tagging features (TBI) only available on AArch64**. These constraints exclude a significant portion of real-world software from analysis, including closed-source commercial software, legacy binaries, third-party libraries distributed only as binaries, and firmware images.

Binary-level solutions like QASan [5] attempt to address the source code requirement by instrumenting binaries through QEMU. However, QASan only supports heap detection and relies on ASan-style redzone checking, which cannot detect non-linear out-of-bounds accesses that skip over redzones.

We present **BSan** (Binary Sanitizer), a non-intrusive memory sanitizer that operates entirely at the binary level through QEMU's linux-user mode emulation. BSan implements HWASan-style pointer tagging via software TBI emulation, enabling detection of both linear and non-linear memory safety violations. Without any external information, BSan can detect heap vulnerabilities including use-after-free and inter-object overflows. To extend detection to stack and global variables, BSan provides a metadata interface that accepts variable boundary information from external sources. Currently, the most accurate approach is extracting this metadata from debug information (DWARF or symbol tables). BSan enables memory safety analysis for any x86-64 ELF binary without requiring source code, recompilation, or special hardware support.

II. Design and Implementation

A. Architecture Overview

Figure 1 illustrates BSan's architecture. BSan integrates into QEMU's Tiny Code Generator (TCG) to instrument memory operations during dynamic binary translation. The system supports independent configuration of heap, stack, and global detection. Key components include: a preloaded library (`libheapsan.so`) for heap allocation interception via LD_PRELOAD, GOT hooking for glibc function interception, shadow memory for tag storage, and TCG instrumentation for memory access validation.

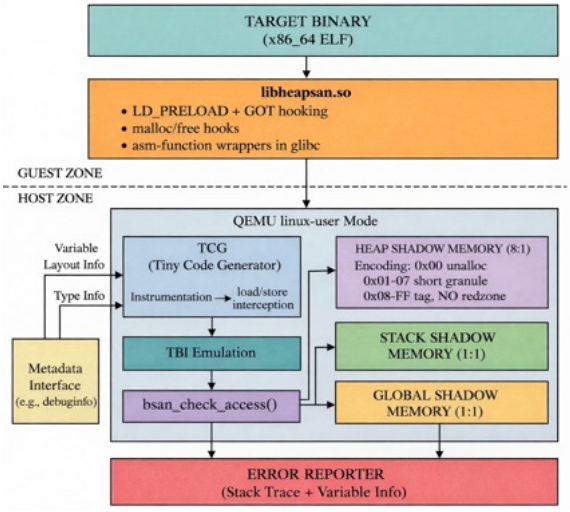


Fig. 1. BSan Architecture Overview

B. Non-Intrusive Shadow Memory

BSan stores shadow memory in QEMU's host memory space, completely separate from the guest program. The 8:1 ratio for heap and 1:1 ratio for stack/globals accommodate the program's original memory layout without modification. For heap, BSan implements short granules: when an allocation is not 8-byte aligned, the last shadow byte stores the allocation size modulo 8, enabling sub-granule overflow detection.

C. Software TBI Emulation

Unlike AArch64, x86-64 does not natively support Top Byte Ignore (TBI). BSan implements software TBI by storing 8-bit tags in pointer bits 56–63. During memory access, syscalls, and atomic operations, BSan masks the top byte to obtain the canonical address while using the tag for validation against shadow memory.

D. TCG Instrumentation

BSan instruments multiple instruction types during TCG translation: (a) load/store operations validate pointer tags against shadow memory; (b) SUB instructions handle pointer subtraction by cleaning tags from both operands to ensure correct arithmetic; (c) LEA instructions inject tags into computed stack and global addresses; (d) CALL/RET instructions track stack frame lifetimes for use-after-return detection.

E. Detection Mechanisms

Heap: `libheapsan.so` intercepts malloc/free via LD_PRELOAD and communicates with QEMU via hypercalls. Each allocation receives a random tag; UAF detection works by re-tagging shadow memory on free.

Stack: CALL/RET interception tracks frame lifetimes: on CALL, a new frame is created with a random tag applied to shadow memory; on RET, the frame's shadow is poisoned for use-after-return detection. LEA instructions inject tags from shadow memory into computed stack addresses. For memory operands without explicit LEA (e.g., `mov al, [rbp+rax-0x7]`), BSan applies base-tagging: the base+displacement is tagged separately before adding the index, preserving the origin tag for overflow detection.

Global: During ELF loading, BSan initializes 1:1 shadow memory with per-variable tags provided by the metadata interface. LEA instructions targeting global addresses read tags from shadow memory and inject them into the resulting pointer.

F. Handling glibc Assembly Functions

A major source of false positives comes from glibc's hand-written assembly functions. Two issues arise: (1) assembly code performs pointer arithmetic that violates C/C++ standards, causing tag pollution where one pointer's tag is transferred to another; (2) SIMD-optimized functions read 16/32 bytes at once even when the allocation is smaller. BSan addresses this through two-layer interception: LD_PRELOAD intercepts external calls by symbol overriding, while IRELATIVE GOT hooking intercepts libc-internal calls (e.g., `printf` calling `strlen`). Wrapper functions validate boundaries, call the real function with untagged pointers, and re-tag the return value.

III. Evaluation

We evaluated BSan on compatibility and detection capability.

Compatibility: We tested common applications installed via apt on Ubuntu 24.04, including vim, python, bash, xeyes, coreutils, etc. Ubuntu's dbgsym packages provide convenient access to debug symbols for stack and global detection.

Detection Capability: We used the MSET benchmark suite [6], which covers linear/non-linear out-of-bounds access, use-after-free/realloc, double-free, misuse-of-free, and type confusion vulnerabilities.

TABLE I
MSET Detection Results (O0 compilation)

Region	Detected	Total	Rate
Heap	120	120	100%
Global	86	90	95%
Stack	120	122	98%
Cross-region	164	164	100%

Intra-object overflows are excluded from the statistics, as memory tagging approaches inherently cannot detect them—tags are assigned per allocation/variable, not per struct field. This matches HWASan's behavior [4]. Note that detection is probabilistic due to potential tag collisions, with approximately 1/256 probability of missing a violation.

IV. Conclusion and Future Work

BSan demonstrates the feasibility of comprehensive memory sanitization at the binary level, achieving detection capabilities comparable to HWASan for inter-object vulnerabilities without requiring source code or special hardware.

Key contributions: (1) a comprehensive binary-level memory sanitizer supporting heap, stack, and global detection, (2) software TBI emulation for x86-64, (3) non-intrusive shadow memory architecture, and (4) a metadata interface framework for extending detection to stack and global variables.

Future work: extending support to additional architectures (AArch64, RISC-V), performance optimization through selective instrumentation, real CVE detection validation, and integration with fuzzing frameworks for automated vulnerability discovery.

References

[1] G. Thomas, M. Miller *et al.*, "A proactive approach to more secure code," in *BlueHat IL 2019*, 2019, microsoft Security Response Center.

[2] The Chromium Projects, "Memory safety," https://www.chromium.org/Home/chromium-security/memory-safety/, 2020, accessed: 2026.

[3] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Annual Technical Conference, USENIX ATC 2012, Boston, MA, USA, June 13-15, 2012*, G. Heiser and W. C. Hsieh, Eds. USENIX Association, 2012, pp. 309–318. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[4] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov, "Memory tagging and how it improves C/C++ memory safety," *CoRR*, vol. abs/1802.09517, 2018. [Online]. Available: http://arxiv.org/abs/1802.09517

[5] A. Fioraldi, D. C. D'Elia, and L. Querzoni, "Fuzzing binaries for memory safety errors with qasan," in *IEEE Secure Development, SecDev 2020, Atlanta, GA, USA, September 28-30, 2020*. IEEE, 2020, pp. 23–30. [Online]. Available: https://doi.org/10.1109/SecDev45635.2020.00019

[6] E. Q. Vintila, P. Zieris, and J. Horsch, "Evaluating the effectiveness of memory safety sanitizers," in *IEEE Symposium on Security and Privacy, SP 2025, San Francisco, CA, USA, May 12-15, 2025*, M. Blanton, W. Enck, and C. Nita-Rotaru, Eds. IEEE, 2025, pp. 774–792. [Online]. Available: https://doi.org/10.1109/SP61157.2025.00088

Poster: BSan: A Non-Intrusive and Comprehensive Binary-Level Memory Sanitizer

Yang Li[1], Yuan Li[2], Chenyang Li[1], Xinhui Han[1], Chao Zhang[3]

[1]Peking University, [2]Zhongguancun Laboratory, [3]Tsinghua University

Introduction and Motivation

Memory safety vulnerabilities remain critical—**70%** of security issues at Microsoft/Google stem from memory safety [1,2].

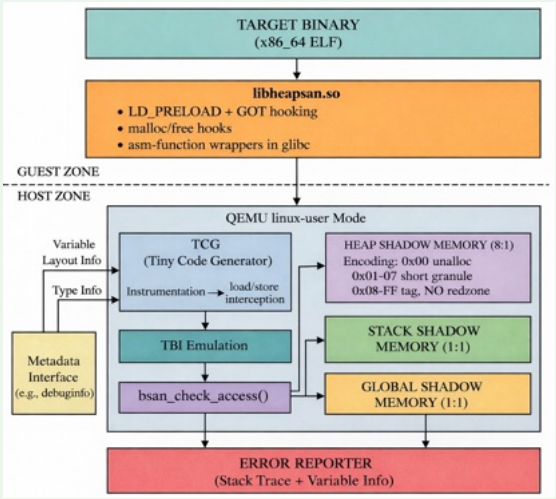
Limitations of existing sanitizers:

- ASan/HWASan require source code and recompilation
- HWASan relies on AArch64 TBI hardware feature
- QASan only supports heap detection with redzone

BSan is a non-intrusive binary-level memory sanitizer:

- Uses QEMU's dynamic binary translation
- HWASan-style pointer tagging via software TBI
- Detects heap, stack, and global vulnerabilities
- No source code or special hardware required

Design and Implementation



Key Components:

- **Shadow Memory:** 8:1 for heap, 1:1 for stack/globals
- **Software TBI:** 8-bit tags in pointer bits 56–63
- **TCG Instrumentation:** Load/Store, SUB, LEA, CALL/RET
- **glibc Hand-written ASM:** LD_PRELOAD + GOT hooking to avoid false positives

Detection Mechanisms:

- **Heap:** malloc/free interception, random tags, UAF
- **Stack:** Variable-level tracking, use-after-return detection
- **Global:** ELF metadata, per-variable tags

Metadata Interface: Flexible sources for stack/global variable boundaries:

- **Debug info:** DWARF, symbol tables, compiler plugins (most accurate)
- **Static analysis:** Ghidra, IDA Pro, angr for stripped binaries
- **Other sources:** Manual annotations, heuristics

Evaluation

1. Compatibility:
vim, python, bash, xeyes, coreutils, etc. (Ubuntu 24.04, with dbgsym packages)

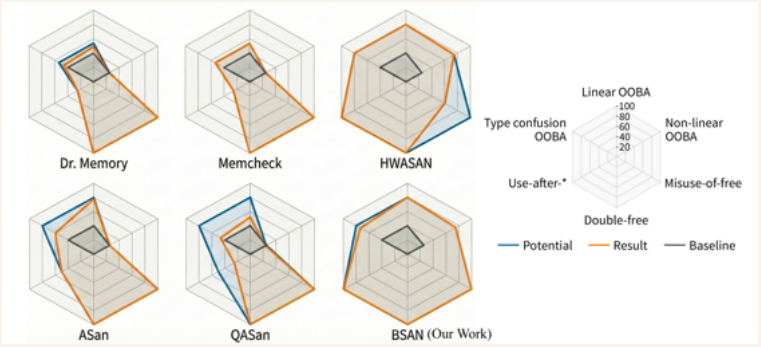
2. Detection Capability:
BSan achieves near-complete detection at binary level (∼1/256 tag collision).

*MSET Benchmark (*intra-object excl.)*

Region	Det/Tot	Rate
Heap	120/120	100%
Global	86/90	95%
Stack	120/122	98%
Cross	164/164	100%

**Memory tagging can't catch intra-object overflows (tag per allocation), as HWASan.*

Comparison on MSET benchmark [3]



BSan achieves HWASan-comparable detection across all memory regions and various vulnerability types (OOB, UAF, double-free, etc.).

Conclusion and Future Work

Key Takeaway:

BSan is the **first** binary-level sanitizer achieving **HWASan-comparable detection** (95–100%) for heap, stack, and global regions without source code or hardware dependencies.

Future Work:

- AArch64, RISC-V architecture support
- Performance optimization via selective instrumentation
- Real CVE detection validation
- Fuzzing framework integration

References: [1] Microsoft Security (BlueHat'19) [2] Chromium Memory Safety [3] MSET (S&P'25)