IntTracer: Sanitization-aware IO2BO Vulnerability Detection across Codebases

Xiang Chen
Shanghai Jiao Tong University
Shanghai, China
cascades@sjtu.edu.cn

ABSTRACT

Integer Overflow to Buffer Overflow (IO2BO) vulnerability represents a common vulnerability pattern in system software and can be detected by various program analysis methods. Mainstream static approaches apply taint analysis to find source-sink pairs and then submit those suspicious bug traces to dynamic instrumentation or static encoding.

However, previous works utilizing those methods either fail to handle sanitization code well or cannot generalize across codebases. In this paper, we present IntTracer, which is enhanced with interval domain to model the effect of sanitization code in IO2BO bug trace and can find recurring vulnerabilities across different codebases. IntTracer can prevent false positives under 8 cases while keeping an overhead of 6.3% compared to previous work Tracer.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**.

KEYWORDS

Integer Overflow, Taint Analysis, Recurring Vulnerability, Interval Analysis

ACM Reference Format:
Xiang Chen. 2024. IntTracer: Sanitization-aware IO2BO Vulnerability Detection across Codebases. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3639478.3641223

1 INTRODUCTION

Integer Overflow to Buffer Overflow (IO2BO) [4] is the most prevalent and harmful pattern of Integer Overflow (IO) [3]. IO2BO bug manifests in two stages. Typically, when an **integer overflow** occurs in a variable from external input and that variable is later used as the parameter for memory allocation functions (like `malloc`), the actual allocated memory becomes significantly smaller than expected. Subsequent operations accessing this memory may result in a **memory overflow**, even facilitating severe RCE exploits [2], with an average and maximum CVSS score of 7.32 and 9.8.

ICSE-Companion '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0502-1/24/04…$15.00
https://doi.org/10.1145/3639478.3641223

IO can be benign [5] under certain scenarios (like crypto functions), leading to a large number of false positives in bug detection. Luckily, empirical study [19] shows that the most significant difference between IO2BO and other IO vulnerabilities lies in that operations on the overflown integer in IO2BO are often unexpected and harmful. Based on that observation, many existing works [7, 10, 12–14, 16, 19] choose to focus on detecting IO2BO bugs.

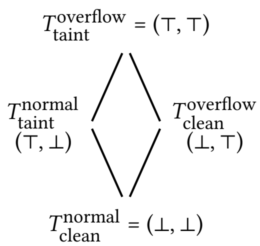
2 RELATED WORK

These works typically embark on IO2BO detection with static taint analysis, extracting the traces between integer variable inputs and memory allocation function calls. Subsequently, based on the approaches to handling tainted traces, these endeavors are categorized into dynamic instrumentation [12, 13, 19] and static encoding [7, 10, 14, 16]. Despite the absence of benign IO, the detection of IO2BO vulnerabilities still results in false positives due to the **oversight of sanitization code** [13] written intentionally by programmers, as shown in Listing 1.

```
1  // seq/aplaymidi/aplaymidi.c:477
2  static int num_tracks;
3  static int read_smf(void) {
4     // read from unbounded user input
5     num_tracks = read_int(2);
6     // pre-conditionally sanitize num_tracks to [1, 1000]
7     if (num_tracks < 1 || num_tracks > 1000) {
8        errormsg("invalid number of tracks (%d)");
9        return 0;
10    }
11    // num_tracks * sizeof(struct track) can not overflow
12    tracks = calloc(num_tracks, sizeof(struct track));
13 }
```

Listing 1: IO2BO sanitization in alsa-utils-1.2.9. `num_tracks` is sanitized by an if-guard (line 7) before the multiplication and allocation (line 12), thus can not overflow at run time.

The two categories of endeavors employ different approaches to address the challenge posed by the sanitization code. For example, IntPatch [19] designs a binary bottom-top domain (Figure 1a) for maintaining the taint and overflow tag and later inserts dynamic checks when a variable with both tags is used in memory allocation. The instrumented code needs to run on each specific codebase while the run-time environment remains a big issue. KINT [16] encodes path and overflow conditions along the taint trace as SMT constraints and relies on the SMT solver, while it only deals with system-level codebase written in C. As IO2BO remains a common bug pattern in various development scenarios (multimedia processing, file parser/converter, etc.), **these methods still fall short in detecting similar IO2BO bugs across codebases.**



(a) Binary bottom-top domain for taint and overflow

$$\textbf{Itv} \triangleq \textbf{Bound} \times \textbf{Bound} \tag{1}$$

$$\begin{cases} \textbf{Bound} \triangleq \mathbb{Z} + max/min(\textbf{Bound}) \\ \textbf{Bound} \triangleq max/min(\textbf{Bound}) \\ \textbf{Bound} \triangleq \mathbb{Z} + \textbf{Bound} \times \textbf{Bound} \end{cases} \tag{2}$$

$$\begin{cases} \textbf{Bound} \triangleq \pm\infty \mid \mathbb{Z} + \textbf{LinearSym} \\ \textbf{LinearSym} \triangleq \sum \textbf{Symbol} \cdot \mathbb{Z} \end{cases} \tag{3}$$

(b) Interval domain for integer value, following InferBO's design and implementation

$$\textbf{(overflow tag)}\ \bar{\mathbb{I}} = \{\bot_o, \top_o\} \tag{4}$$

$$\mathcal{V}(n)(m) = \mathcal{V}(\text{source})(m) = \bot_o \tag{5}$$

$$\textbf{(may)}\ \mathcal{V}(E_1 \oplus E_2)(m) = \top_o, \text{if } ub > ub_e \text{ or } ub = +\infty \tag{6}$$

$$\textbf{(must not)}\ \mathcal{V}(E_1 \oplus E_2)(m) = \bot_o, \text{if } ub \leq ub_e \tag{7}$$

$$\text{where } \textbf{Itv}(E_1 \oplus E_2)(m) = [lb, ub]$$

$$\mathcal{V}(E_1 \oplus E_2)(m) = \mathcal{V}(E_1)(m) \sqcup \mathcal{V}(E_2)(m) \tag{8}$$

(c) Abstract operational semantics of overflow tag generation and propagation in binary expression, $\oplus \in \{+, -, \times, \div, \text{«}, \text{»}\}$

Figure 1: Abstract domain and semantics used in IntTracer, taking overflow as an example

On top of IntPatch, Tracer [7] proposed a method for extracting and comparing IO2BO vulnerability signatures, capable of identifying recurring IO2BO vulnerabilities in different codebases. Though Tracer has somewhat encoded sanitization code to the signature, it blindly relies on the similarity score. For example, Tracer computes the similarity score of 0.86 between Listing 1 and CVE-2017-16612 [11], which is above the reporting threshold of 0.85, resulting in a false positive. We even found that the bug report in grass-7.82 in Tracer's paper was actually a false positive.

3 DESIGN

To strike a balance between sanitization code awareness and cross-codebase analysis, we conducted an empirical study [2] of real-world IO2BO vulnerabilities collected from previous works [13, 15, 16, 19] and historical CVEs. The study indicated that the generation, sanitization, and propagation of overflow tags are critical to IO2BO bug reports. However, both Tracer and IntPatch assign overflow tags on each binary expression for safe approximation.

We then propose IntTracer (**Int**erval-assisted **Tracer**), which is an amalgamation of Tracer's abstract domain (Figure 1a) and InferBO's interval domain (Figure 1b) with a more precise operational semantics from AbsIntIO [9]. The interval domain (Eq 1) is defined as a pair of lower and upper Bound $[lb, ub]$ to represent the possible range of an integer variable. Bound is both recursively (Eq 2) and linearly (Eq 3) defined to support inter-procedural and symbolic analysis. IntTracer performs the interval analysis along with taint analysis and uses interval value to check the generation, sanitization, and propagation of overflow tag (Eq 4).

As for constant or integer from external input (Eq 5), IntTracer initiates them with no overflow tag. When handling binary expression, IntTracer assigns an overflow tag only when the interval value may (Eq 6) exceed its expected range, removes tag when the interval value must not (Eq 7) exceed the expected range and joins tags by default (Eq 8). The expected range $[lb_e, ub_e]$ of an integer variable is obtained from its integer type width, e.g. `[INT_MIN, INT_MAX]` for `int num_tracks` in Listing 1. IntTracer also designs a symmetric check scheme for integer underflow. These checks can be divided into two categories, corresponding to sanitization code occurring before and after the overflow behavior respectively:

- **Pre-check** Apply Eq 6, Eq 7, and Eq 8 to each binary expression and their child expression recursively.
- **Post-check** Apply Eq 7 to each memory allocation argument.

As for other instructions, IntTracer propagates the tags by doing join and widen operations on the domain in Figure 1a.

4 EVALUATION

IntTracer is implemented on top of Tracer [7] with two kinds of checks in ~600 lines of OCaml code. The interval analysis component is supported by InferBO [18] checker in Facebook's Infer analyzer [1]. We have made some minor adjustments to achieve higher interval precision and make it consistent with Figure 1c.

To evaluate its ability to detect cross-codebase vulnerability, we manually select 47 C/C++ OpenWrt packages in 8 development areas, together with 273 Debian packages in Tracer's evaluation as the dataset. Theoretically, IntTracer can support any programming languages (like Obj-C and Java) adopted by Infer's front-end.

- **RQ1: IO2BO Detection and Overhead** IntTracer has detected all 5 CVEs found by Tracer on Debian packages, as true positives. Besides, IntTracer has discovered 3 new IO2BO bugs in nmap-7.93 and syslog-ng-4.2.0, all of which have been fixed by developers. 2 of the newly detected bugs are similar to historical CVEs from packages in different categories. The average analysis overhead (6.3%) that comes from the two checks is acceptable.
- **RQ2: False Positive Reduction** IntTracer has successfully avoided 8 IO2BO bug reports from different packages (nmap-7.93, alsa-utils-1.2.9, ipmitool-1.8.18, ImageMagick-7.0.9-5, monit-5.26.0), including the motivating example in Listing 1, while Tracer reports them all as false positives.

5 CONCLUSION AND EXPECTATION

In this work, we present the prototype of IntTracer, a sanitization-aware IO2BO bug detection tool, which can detect vulnerabilities across codebases while largely reducing false positives.

The design behind IntTracer can also be applied to detecting Buffer Overflow [8] and integer-related logical bugs [6, 17]. The future works are (1) modeling sources and sinks for new vulnerability types in taint analysis, (2) conducting larger-scale experiments on different codebases, and (3) validating the exploitability of IO2BO bugs and updating the vulnerability signature database.

6 ACKNOWLEDGEMENT

This paper is under Prof. Yue Wu's supervision and supported by the National Key R&D Program of China (No.2020YFB1807504).

REFERENCES

[1] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *Proceedings of the Third International Conference on NASA Formal Methods* (Pasadena, CA) *(NFM'11)*. Springer-Verlag, Berlin, Heidelberg, 459–465.

[2] Xiang Chen. 2023. *Real-world CWE680: Integer Overflow to Buffer Overflow (IO2BO) vulnerability collections*. GitHub. https://github.com/cascades-sjtu/rw-io2bo

[3] The MITRE Corporation. 2023. *CWE-190: Integer Overflow or Wraparound*. The MITRE Corporation. Retrieved October 26, 2023 from https://cwe.mitre.org/data/definitions/190.html

[4] The MITRE Corporation. 2023. *CWE-680: Integer Overflow to Buffer Overflow*. The MITRE Corporation. Retrieved October 26, 2023 from https://cwe.mitre.org/data/definitions/680.html

[5] Will Dietz, Peng Li, John Regehr, and Vikram Adve. 2012. Understanding Integer Overflow in C/C++. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Zurich, Switzerland, 760–770.

[6] Horn Jann. 2024. *Linux 5.6 io_uring Cred Refcount Overflow*. packet storm. https://packetstormsecurity.com/files/176649/Linux-5.6-io_uring-Cred-Refcount-Overflow.html

[7] Wooseok Kang, Byoungho Son, and Kihong Heo. 2022. TRACER: Signature-Based Static Analysis for Detecting Recurring Vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1695–1708. https://doi.org/10.1145/3548606.3560664

[8] Andreas D. Kellas, Alan Cao, Peter Goodman, and Junfeng Yang. 2023. Divergent Representations: When Compiler Optimizations Enable Exploitation. In *2023 IEEE Security and Privacy Workshops (SPW)*. 337–348. https://doi.org/10.1109/SPW59333.2023.00035

[9] Alexander Küchler, Leon Wenning, and Florian Wendland. 2023. AbsIntIO: Towards Showing the Absence of Integer Overflows in Binaries Using Abstract Interpretation. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security* (Melbourne, VIC, Australia) *(ASIA CCS '23)*. Association for Computing Machinery, New York, NY, USA, 247–258. https://doi.org/10.1145/3579856.3582814

[10] Fan Long, Stelios Sidiroglou-Douskos, Deokhwan Kim, and Martin Rinard. 2014. Sound Input Filter Generation for Integer Overflow Errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 439–452. https://doi.org/10.1145/2535838.2535888

[11] MITRE. 2017. *CVE-2017-16612 Detail*. NVD. https://nvd.nist.gov/vuln/detail/CVE-2017-16612

[12] Marios Pomonis, Theofilos Petsios, Kangkook Jee, Michalis Polychronakis, and Angelos D. Keromytis. 2014. IntFlow: Improving the Accuracy of Arithmetic Error Detection Using Information Flow Tracking. In *Proceedings of the 30th Annual Computer Security Applications Conference* (New Orleans, Louisiana, USA) *(ACSAC '14)*. Association for Computing Machinery, New York, NY, USA, 416–425. https://doi.org/10.1145/2664243.2664282

[13] Hao Sun, Xiangyu Zhang, Chao Su, and Qingkai Zeng. 2015. Efficient Dynamic Tracking Technique for Detecting Integer-Overflow-to-Buffer-Overflow Vulnerability. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (Singapore, Republic of Singapore) *(ASIA CCS '15)*. Association for Computing Machinery, New York, NY, USA, 483–494. https://doi.org/10.1145/2714576.2714605

[14] Hao Sun, Xiangyu Zhang, Yunhui Zheng, and Qingkai Zeng. 2016. IntEQ: Recognizing Benign Integer Overflows via Equivalence Checking across Multiple Precisions. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 1051–1062. https://doi.org/10.1145/2884781.2884820

[15] Wang Tielei, Wei Tao, Lin Zhiqiang, and Zou Wei. 2009. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*. The Internet Society, USA. https://www.ndss-symposium.org/ndss2009/intscope-automatically-detecting-integer-overflow-vulnerability-in-x86-binary-using-symbolic-execution/

[16] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Improving Integer Security for Systems with KINT. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 163–177.

[17] Wikipedia. 2024. *Year 2038 problem*. Wikipedia. https://en.wikipedia.org/wiki/Year_2038_problem

[18] Kwangkeun Yi. 2017. *Inferbo: Infer-based buffer overrun analyzer*. Meta Research. Retrieved Feburary 6, 2023 from https://research.facebook.com/blog/2017/02/inferbo-infer-based-buffer-overrun-analyzer/

[19] Chao Zhang, Tielei Wang, Tao Wei, Yu Chen, and Wei Zou. 2010. IntPatch: Automatically Fix Integer-Overflow-to-Buffer-Overflow Vulnerability at Compile-Time. In *Proceedings of the 15th European Conference on Research in Computer Security* (Athens, Greece) *(ESORICS'10)*. Springer-Verlag, Berlin, Heidelberg, 71–86. https://doi.org/10.5555/1888881.1888888