Binary lifting for Object-oriented code: How far are we?

XIANG CHEN, The Hong Kong University of Science and Technology, China

CCS Concepts: • **Security and privacy → Software reverse engineering**.

Additional Key Words and Phrases: Type Recovery, Control-flow Recovery, OOP, Static Bug Detection

1 INTRODUCTION

Binary lifting [8], the process of translating low-level assembly (e.g., x86-64, Arm) code to mid-level Intermediate Representation (e.g., LLVM IR) code, is a major technique used in binary analysis. The lifted binary code can facilitate several downstream applications in software security (e.g., static bug detection, retrofitting, decompilation) when source code is not available, and the quality of lifted binary code affects the performance of those tasks significantly. Thus, improving the capability and performance of binary lifters is of great importance to security research.

Over the past 10 years, dozens of binary lifters [1–3, 13, 15, 16, 18, 23–25, 28, 29, 35, 40, 44] have been proposed for lifting binaries compiled from C/C++ [30], the most prevalent system programming languages in creating infrastructural software. The C programming language has the most closed semantics to assembly language and is usually programmed in the imperative way, which makes its binary not hard to lift [4]. However, C++ has introduced many high-level Object-Oriented programming semantics (e.g., encapsulation, inheritance, polymorphism, and exception handling) and is still introducing new features nowadays [38]. These high-level OOP abstractions reduce the burden on programmers at the cost of bringing much more complex binary code. Even to this day, no binary lifter claims to be capable of handling all features of C++, and consequently, it is hard for researchers/engineers to choose/develop the appropriate lifter when dealing with C++ binary code.

To gain a deeper understanding of the state-of-the-art of lifting OOP binary code and find future research opportunities, this review includes efforts from both academia (10 papers) and industry (5 tools). The result of the review is given in Section 2, and we summarize the contribution from this research area in Section 3.

2 REVIEW

The review is given from both academic and industry perspectives to find the gaps between them.

2.1 State-of-the-art of Lifting OOP features

Generally speaking, a binary lifter completes two tasks in the process of lifting: (1) control flow recovery and (2) type recovery. Given the disassembled assembly code, the first task aims to recover both possible intra- and inter-procedural control flow at runtime, and the second task aims to recover type information for both local and global variables, especially when without compiler metadata. These two tasks form the two main aspects of evaluating the lifter's capability. In the context of Object-Oriented programming, SmartDec [21], probably the first well-round work on C++ lifting, proposed 4 corresponding OOP features regarding the two lifting tasks as shown in Table 1.

2.1.1 Lifting OOP features in Academia. In the academic scenario, the definition of binary lifting is broad in terms of the finally lifted IR and its downstream application. Most of the research ideas are implemented on common-off-the-shelf IR (Vex IR [31, 37], QEMU TCG [2, 15], P-Code [9, 22, 32, 39]), while others use domestic IR [10, 17, 26, 35, 41]. Due to the diversity of IR usage, this review mainly focuses on the principle technique learned from the literature rather than implementation.

Table 1. Object-Oriented Programming features and their corresponding lifting tasks

OOP Features \ Lifting Tasks	Control Flow Recovery	Type Recovery
Encapsulation	✓	
Inheritance		✓
Polymorphism	✓	✓
Exception handling	✓	✓

Encapsulation modularizes the code structure by providing additional layers of function interface, resulting in a more complex control-flow graph (CFG). Specifically, it requires the lifter to recognize the precise boundary of all functions in the disassembling phase, which has proven a difficult problem on stripped binaries [4] for both C and C++. ByteWeight [6] automatically learns key features for recognizing functions and can therefore easily be adapted to different platforms, new compilers, and new optimizations; Nucleus [5] proposes a new function detection algorithm without any learning phase or signature information, and is capable for difficult cases such as non-contiguous or multi-entry functions.

Inheritance is implemented in C++ as class relationship, which exists in the form of compiler metadata (RTTI Info, virtual table/vtable, etc.) in the binary. Technically, it requires the lifter to precisely identify the address and length of vtable and reconstruct the class hierarchy. Marx [33] utilize 6 heuristics in vtables' binary layout, but they are restricted to certain C++ ABI; OOAnalyzer [35] identify the vtable via the dataflow path of the object pointer and constructor and support prolog rules to recover the class attributes; DeClassifier [19] consider the constructor inlining brought by compiler optimization and design a more robust lifter under multiple optimization settings; VirtAnalyzer [20] prove the common existence of virtual inheritance and recover a more complete class hierarchy.

Polymorphism allows C++ objects to call different functions with the same function name at run-time. Further to lifting inheritance, it requires the lifter to infer the indirect calling relationship on top of the class hierarchy. BPA [27] presents a binary-level points-to-analysis framework to construct sound and high-precision CFG but can not deal with C++ binaries; CALLEE [45] combine transfer learning and contrastive learning to find indirect calls and supports C++ functions.

Exception handling is a more complicated mechanism compared to previous features as it requires the lifter to know not only the type of exception but also the address of landing pads in stack unwinding. bin-CFI [43] firstly consider the exception handling control flow to the landing pad, but in a very conservative way; ARMore [7] parses and rewrites the landing pad table for binary rewriting, but still suffers from the variety and complexity of dwarf encodings; Bockenek Et al. [9] leverage symbolic execution to emulate the error handling process statically and finally construct an exceptional interprocedural control flow graphs for x86-64 binaries.

It is also worth noting that, those OO features together facilitate the majority of vulnerabilities in C++ (Bad type casting, Code reuse, Control flow hijacking, etc.). To defend against these attacks, researchers from the security community aim at hardening binary with control-flow integrity (CFI) [11] check, which is also the main downstream application of lifting C++. Although this article only showcases a few works on CFI, there are clearly many more insightful ideas in this area.

2.1.2 Binary lifters in the Industry. In the industrial scenario, engineers mainly focus on binary lifters that produce compiler-compatible (LLVM/GCC) IR, as they (1) can provide rich type information [18] and thus perform better on various downstream applications; (2) have abundant libraries

and tools as developing infrastructure. There have also emerged unified lifting frameworks [34, 36] recently, but they have not been deployed widely yet.

Binary lifters in the early age [1, 3, 13, 23, 28] produce emulation-style, low-level IR and only focus on C binaries. For example, their lifted IR still contains the register usage pattern. Thus, it can not relate to the variable type in the original C/C++ source code. In even worse cases, they may generate broken IR from peculiar binary code. Besides, those lifters can neither catch up with the fast development of LLVM nor maintain well, so there is little usage of them in the industry nowadays.

As for mainstream binary lifters nowadays, we classify them as follows:

- McSema [1] [16] can lift exception handling code using the landing pad address given by IDA Pro. McSema can also read vtable information from IDA Pro but does not process it;
- RetDec [29] and Plankton [44] use similar heuristics as Marx to recover vtable and class hierarchy. Plankton can output nicer class types in lifted IR with compiler metadata (debug info, symbols);
- Mctoll [40] fail to process executable compiled from C++ code [30];
- Revng [15] considers vtable when recovering the control flow graph in its original paper, however, it now focuses on decompiling binaries to C;
- Lisc [24], BinRec [2], Reopt [25] does not report to support C++ lifting.

2.2 Research Opportunities

As shown above, **developing a robust, scalable, and effective binary lifter for C++ binaries at the industry level is still an open problem.** Depending on the downstream application, engineers can tailor and adapt those ideas from academia into real-world binary lifters. For instance:

- For static bug detection, a more complete whole-program callgraph can be built when including the virtual call relationship;
- for static bug detection, special care can be taken of the class hierarchy in third-party, stripped libraries. A summary-based lifting is practical for those common C++ system libraries;
- For recompilation, reverse engineers can fully utilize the class hierarchy and virtual function signature to produce more idiomatic, OOP-style C++ source code.

Sometimes, there may emerge cases that are overlooked by academia. For example, relative virtual table [12] is a new OOP feature provided by the Clang compiler to reduce memory consumption at linking time. This new feature certainly deactivates the vtable discovery methods previously proposed by academia and needs further research endeavors on recovering normal vtable mixed with relative vtable.

Beyond those technical opportunities, engineers may inevitably encounter thousands of edge cases (memory/CPU usage, flaky bugs, etc.) when lifting C++ code at the industry scale. Once the patterns and root causes are found in those edge cases, either fuzzing [42] or verification [14] techniques can be applied to ensure the quality and safety of binary lifter.

3 CONTRIBUTION

This section concludes the contribution of C++ binary lifting to both the research community and reverse engineers.

[1] McSema probably proposed the first C++ lifting question found on public internet: https://youtu.be/nW9bE5tUVYg?t=3720.

3.1 Equip downstream research fields with better binary lifter

Binary lifting can facilitate dozens of downstream research fields. With a stronger binary lifter that can translate C++ binaries to mid-level IR, researchers can ideally treat all C/C++ binaries uniformly to save lots of engineering efforts. Besides, as the quality of the binary lifter grows, it is expected to output lifted IR equivalent to the corresponding compiled IR, which further bridges the gap between source analysis and binary analysis communities.

3.2 Unveiling the nature of OOP in binary code

For other programming languages (e.g., Rust, Go, OCaml) with OOP features, the general principles of lifting their binaries should be similar to what we have reviewed from lifting C++. These principles can (1) transfer to reverse engineering tools with further engineering efforts and (2) guide reverse engineers to manually analyze binaries with unknown source programming languages.

REFERENCES

[1] Ahmed, B., Geoffroy, A., Pierre, C., Thomas, C., Jonathan, S., and Amaury, d. l. V. Dagger: Decompiling to ir, 2013.

[2] Altinay, A., Nash, J., Kroes, T., Rajasekaran, P., Zhou, D., Dabrowski, A., Gens, D., Na, Y., Volckaert, S., Giuffrida, C., Bos, H., and Franz, M. Binrec: dynamic binary lifting and recompilation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (New York, NY, USA, 2020), EuroSys '20, Association for Computing Machinery.

[3] Anand, K., Smithson, M., Elwazeer, K., Kotha, A., Gruen, J., Giles, N., and Barua, R. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, Association for Computing Machinery, p. 295–308.

[4] Andriesse, D., Chen, X., van der Veen, V., Slowinska, A., and Bos, H. An In-Depth analysis of disassembly on Full-Scale x86/x64 binaries. In *25th USENIX Security Symposium (USENIX Security 16)* (Austin, TX, Aug. 2016), USENIX Association, pp. 583–600.

[5] Andriesse, D., Slowinska, A., and Bos, H. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)* (2017), pp. 177–189.

[6] Bao, T., Burket, J., Woo, M., Turner, R., and Brumley, D. BYTEWEIGHT: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 845–860.

[7] Bartolomeo, L. D., Moghaddas, H., and Payer, M. ARMore: Pushing love back into binaries. In *32nd USENIX Security Symposium (USENIX Security 23)* (Anaheim, CA, Aug. 2023), USENIX Association, pp. 6311–6328.

[8] Basque, Z. L. Program Lifting - Decompilation Wiki, 2024.

[9] Bockenek, J., Verbeek, F., and Ravindran, B. Exceptional interprocedural control flow graphs for x86-64 binaries. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 21st International Conference, DIMVA 2024, Lausanne, Switzerland, July 17–19, 2024, Proceedings* (Berlin, Heidelberg, 2024), Springer-Verlag, p. 3–22.

[10] Brumley, D., Jager, I., Avgerinos, T., and Schwartz, E. J. Bap: a binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (Berlin, Heidelberg, 2011), CAV'11, Springer-Verlag, p. 463–469.

[11] Burow, N., Carr, S. A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., and Payer, M. Control-flow integrity: Precision, security, and performance. *ACM Comput. Surv. 50*, 1 (Apr. 2017).

[12] Chan, L. Relative vtables in c++, 2021.

[13] Chipounov, V., and Candea, G. Enabling sophisticated analyses of ×86 binaries with revgen. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops* (USA, 2011), DSNW '11, IEEE Computer Society, p. 211–216.

[14] Dasgupta, S., Dinesh, S., Venkatesh, D., Adve, V. S., and Fletcher, C. W. Scalable validation of binary lifters. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2020), PLDI 2020, Association for Computing Machinery, p. 655–671.

[15] Di Federico, A., Payer, M., and Agosta, G. rev.ng: a unified binary analysis framework to recover cfgs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction* (New York, NY, USA, 2017), CC 2017, Association for Computing Machinery, p. 131–141.

[16] Dinaburg, A., Com, T., Ruef, A., and Com, T. McSema: Static Translation of X86 Instructions to LLVM, 2014.

[17] Djoudi, A., and Bardin, S. BINSEC: binary code analysis with low-level regions. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint*

Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (2015), C. Baier and C. Tinelli, Eds., vol. 9035 of *Lecture Notes in Computer Science*, Springer, pp. 212–217.

[18] Engelke, A., and Schulz, M. Instrew: leveraging llvm for high performance dynamic binary instrumentation. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2020), VEE '20, Association for Computing Machinery, p. 172–184.

[19] Erinfolami, R. A., and Prakash, A. DeClassifier: Class-Inheritance Inference Engine for Optimized C++ Binaries. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security* (Auckland New Zealand, July 2019), ACM, pp. 28–40.

[20] Erinfolami, R. A., and Prakash, A. Devil is Virtual: Reversing Virtual Inheritance in C++ Binaries. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event USA, Oct. 2020), ACM, pp. 133–148.

[21] Fokin, A., Derevenetc, E., Chernov, A., and Troshina, K. Smartdec: Approaching c++ decompilation. In *2011 18th Working Conference on Reverse Engineering* (Oct 2011), pp. 347–356.

[22] Freek, V., Nico, N., and Binoy, R. Verifiably correct lifting of position-independent x86-64 binaries to symbolized assembly. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2024), CCS '24, Association for Computing Machinery.

[23] Félix, C. fcd: An optimizing decompiler., 2017.

[24] Hasabnis, N., and Sekar, R. Lifting assembly to intermediate representation: A novel approach leveraging compilers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2016), ASPLOS '16, Association for Computing Machinery, p. 311–324.

[25] Hendrix, J., Wei, G., and Winwood, S. Towards verified binary raising. In *Workshop on Instruction Set Architecture Specification (co-located with ITP 2019)* (2019), vol. 6.

[26] Jung, M., Kim, S., Han, H., Choi, J., and Cha, S. K. B2R2: Building an efficient front-end for binary analysis. In *Proceedings of the NDSS Workshop on Binary Analysis Research* (2019).

[27] Kim, S., Sun, C., Zeng, D., and Tan, G. Refining indirect call targets at the binary level. In *NDSS* (2021).

[28] Kirchner, K., and Rosenthaler, S. bin2llvm: Analysis of binary programs using llvm intermediate representation. In *Proceedings of the 12th International Conference on Availability, Reliability and Security* (New York, NY, USA, 2017), ARES '17, Association for Computing Machinery.

[29] Křoustek, J., and Matula, P. RetDec: An Open-Source Machine-Code Decompiler, 2017.

[30] Liu, Z., Yuan, Y., Wang, S., and Bao, Y. Sok: Demystifying binary lifters through the lens of downstream applications. In *2022 IEEE Symposium on Security and Privacy (SP)* (May 2022), pp. 1100–1119.

[31] Nethercote, N., and Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2007), PLDI '07, Association for Computing Machinery, p. 89–100.

[32] Nico, N., Freek, V., Sagar, A., and Binoy, R. Poster: Formally verified binary lifting to p-code. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2024), CCS '24, Association for Computing Machinery.

[33] Pawlowski, A., Contag, M., Van Der Veen, V., Ouwehand, C., Holz, T., Bos, H., Athanasopoulos, E., and Giuffrida, C. MARX: Uncovering Class Hierarchies in C++ Programs. In *Proceedings 2017 Network and Distributed System Security Symposium* (San Diego, CA, 2017), Internet Society.

[34] Schulte, E., Dorn, J., Flores-Montoya, A., Ballman, A., and Johnson, T. GTIRB: Intermediate Representation for Binaries, Apr. 2020. arXiv:1907.02859 [cs].

[35] Schwartz, E. J., Cohen, C. F., Duggan, M., Gennari, J., Havrilla, J. S., and Hines, C. Using Logic Programming to Recover C++ Classes and Methods from Compiled Executables. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto Canada, Oct. 2018), ACM, pp. 426–441.

[36] Scott, R. G., Boston, B., Davis, B., Diatchki, I., Dodds, M., Hendrix, J., Matichuk, D., Quick, K., Ravitch, T., Robert, V., Selfridge, B., Stefănescu, A., Wagner, D., and Winwood, S. Macaw: A Machine Code Toolbox for the Busy Binary Analyst, July 2024. arXiv:2407.06375 [cs].

[37] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., and Vigna, G. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy* (2016).

[38] Stroustrup, B. Thriving in a crowded and changing world: C++ 2006–2020. *Proc. ACM Program. Lang. 4*, HOPL (June 2020).

[39] Wen, H., and Lin, Z. Egg hunt in tesla infotainment: A first look at reverse engineering of qt binaries. In *32nd USENIX Security Symposium (USENIX Security 23)* (Anaheim, CA, Aug. 2023), USENIX Association, pp. 3997–4014.

[40] Yadavalli, S. B., and Smith, A. Raising binaries to LLVM IR with MCTOLL (WIP paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (Phoenix

AZ USA, June 2019), ACM, pp. 213–218.

[41] Yakdan, K., Eschweiler, S., Gerhards-Padilla, E., and Smith, M. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS* (2015).

[42] Yutong, Z., Fan, Y., Zirui, S., Ke, Z., Jiongyi, C., and Kehuan, Z. Liftfuzz: Validating binary lifters through context-aware fuzzing with gpt. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2024), CCS '24, Association for Computing Machinery.

[43] Zhang, M., and Sekar, R. Control flow integrity for COTS binaries. In *22nd USENIX Security Symposium (USENIX Security 13)* (Washington, D.C., Aug. 2013), USENIX Association, pp. 337–352.

[44] Zhou, A., Ye, C., Huang, H., Cai, Y., and Zhang, C. Plankton: Reconciling binary code and debug information. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (New York, NY, USA, 2024), ASPLOS '24, Association for Computing Machinery, p. 912–928.

[45] Zhu, W., Feng, Z., Zhang, Z., Chen, J., Ou, Z., Yang, M., and Zhang, C. Callee: Recovering Call Graphs for Binaries with Transfer and Contrastive Learning . In *2023 IEEE Symposium on Security and Privacy (SP)* (Los Alamitos, CA, USA, May 2023), IEEE Computer Society, pp. 2357–2374.

revised 22 Nov 2024