Automatically Repairing Binary Programs Using Adapter Synthesis

Vaibhav Sharma
Department of Computer Science and Engineering
University of Minnesota - Twin Cities
Minneapolis, MN, USA
vaibhav@umn.edu

***Abstract*—Bugs in commercial software and third-party components are an undesirable and expensive phenomenon. Such software is usually released to users only in binary form. The lack of source code renders users of such software dependent on their software vendors for repairs of bugs. Such dependence is even more harmful if the bugs introduce new vulnerabilities in the software. Automatically repairing security and functionality bugs in binary code increases software robustness without any developer effort. In this research, we propose development of a binary program repair tool that uses existing bug-free fragments of code to repair buggy code.**

***Index Terms*—automated program repair, adapter synthesis, binary analysis, symbolic execution**

I. Introduction

Bugs are a common and expected occurrence in software. The amount of money lost in the US in 2018 due to poor software quality was conservatively estimated at $1.1 trillion, with $479 billion of that amount lost due to effort spent on finding and fixing bugs [1]. While it is desirable to release software that is guaranteed to be bug-free, it is practically impossible to provide such claims to users during commercial software releases. On encountering undesirable behavior in commercial software, for example the software abruptly crashes on some inputs, users have to either search for a workaround on their own, wait patiently for the encountered issues to be fixed, or revert the software back to a previously functional, but less useful, version. This leads to a considerable loss in productivity for users. Unexpected behavior, that compromises security of the software, puts sensitive information of users at risk. At such times, users often wish they could identify workarounds or fixes for such issues automatically without having to wait for their software vendor to fix it. However, it is common for commercial software manufacturers to distribute their software only in its binary form. As opposed to source code, binary code is not meant to be human-modifiable and is optimized for execution by a processor. Repairing issues in the binary code of commercial software can be a daunting task for non-expert users since reverse engineering semantic knowledge from binary code is usually beyond the scope of their technical knowledge. Ensuring the correctness of such repairs involves not just gaining an accurate understanding of the binary code but also testing it to see if it causes other undesirable behaviors. While it is possible to expend expert human effort into repairing bugs at the binary level, this approach does not scale with a constantly increasing number of bugs, making automatic repair of binary code desirable.

Once the exact location of a fault is known, one approach often used in automated program repair is to change the fragment of code around it that produces the undesirable behavior. While this approach is closer to how human programmers fix bugs in software, it is difficult to generalize to the diverse kinds of changes needed for fixing the same bug in different software. Another approach is to (1) find a semantic clone of the original buggy code fragment such that the semantic clone provides the same functionality as the original fragment but does not have the undesirable behavior, (2) replace the original buggy fragment with its non-buggy semantic clone in the binary. Automatically repairing bugs in binary code by performing a scalable search for such semantic clones of buggy code fragments is the major thrust of this research proposal. The smallest such buggy binary fragment would likely have many reference clones that come close to its functionality. Apart from searching for semantic clones, this research also attempts to explore the trade-off between the size of the buggy binary fragment being replaced and the scalability of this approach.

Implementations of cryptographic functionality are constantly being scavenged for security bugs by attackers. OpenSSL is one of the most popular software suites that lies in this category. In March 2016, CVE-2016-2108 [2] was reported in OpenSSL. The bug was specific to applications that parsed and re-encoded X509 certificates. This security vulnerability was possible due to a buffer underflow in OpenSSL's implementation that converts a OpenSSL `ASN1_INTEGER` structure to its DER encoding and writes it to a buffer. This conversion involves computing the two's complement of a value represented as an arbitrary number of bytes in big endian format. Figure 1 shows a slice of the `i2c_ASN1_INTEGER` method in OpenSSL that performs this conversion. The National Vulnerability Database classified this bug as having "high" severity since it was found to be exploitable. A clever attacker who could cause a webserver to read their specially hand-crafted file could trigger this bug and potentially write arbitrary data to the webservers memory. However, this conversion implementation is not unique to OpenSSL. Botan is an independently developed library similar to OpenSSL that has code that converts an arbitrarily-sized big-endian value to its twos complement form which is shown in Figure 2. The

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00149

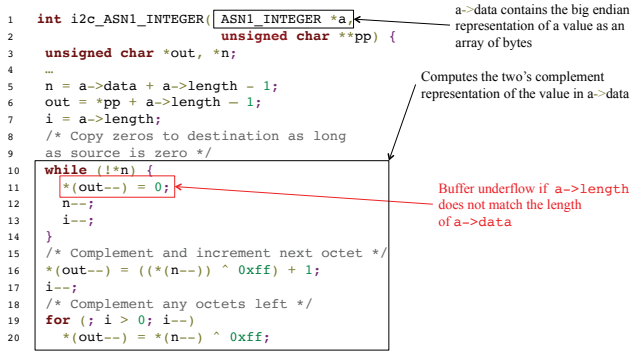

Fig. 1. Implementation of two's complete of a value represented as a sequence of bytes in big-endian representation in OpenSSL's `i2c_ASN1_INTEGER` method

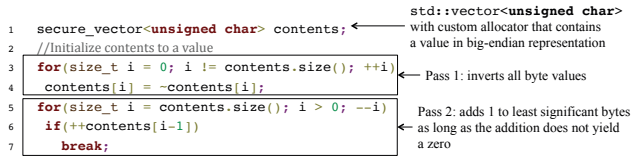

Fig. 2. Another implementation of two's complement of a value represented as a `std::vector` in Botan

buffer underflow bug present in the OpenSSL implementation does not exist in the Botan implementation. In fact, the Botan implementation was available for more than 2 years before the bug in the OpenSSL implementation was discovered. With the Botan implementation being a natural clone of the OpenSSL implementation without the buffer underflow bug, an obvious direction to repair this bug is to substitute the buggy OpenSSL implementation with the bug-free Botan implementation. But, even if we can automatically sift through the code of all the other independently developed cryptographic tools and find this clone in Botan, such substitution is not straightforward. First, we need to identify the interfaces to the buggy fragment we are trying to substitute, and the bug-free fragment we are trying to substitute the buggy fragment with. In this example, we would have to find that (1) that the buggy binary fragment we are trying to substitute spans lines 5-20 of Figure 1, (2) the bug-free binary fragment we should find is encapsulated by the implementation shown in Figure 2. An interface would clearly specify the inputs and outputs of each of these two fragments. We would also need to reconcile any differences between these two interfaces by wrapping around one or both interfaces. This wrapper would make the interfaces match and make an exact substitution possible to complete the repair. This wrapper is called an adapter in a previously developed technique called adapter synthesis [3], [4].

II. Related Work

A. Adapter Synthesis

Our proposed approach to binary program repair builds on existing research done by Sharma et al. [3], [5]. Given a target code fragment, adapter synthesis ensures that the adapted reference fragment's side-effects and outputs match those of the target fragment's for all inputs to the target fragment. Adapter synthesis needs to be extended for program repair along two directions. (1) We need to extend adapter synthesis to perform adaptation modulo an undesirable behavior. (2) While Sharma et al. [5] find an adaptable substitution for all possible inputs to the target implementation, program repair needs to only repair uses of the target implementation. In other words, program repair needs to take the context of usage of the target implementation into account.

B. Program Synthesis

Program synthesis is an active area of research that has many applications including generating optimal instruction sequences [6], [7], automating repetitive programming, filling in low-level program details after programmer intent has been expressed [8], and even binary diversification [9]. Programs can be synthesized from formal specifications [10], simpler (likely less efficient) programs that have the desired behavior [6]–[8], or input/output oracles [11]. Our proposed research takes the second approach to specification, treating existing target code as specification when synthesizing a repair using an adapted reference implementation.

C. Program Repair

The area of automated program repair has seen a tremendous interest in the research community in the last few years. A comprehensive survey of this research area was recently presented by Gazzola et al. [12]. The closest area of research for generating a fix automatically comes from the set of tools that use a *generate-and-validate* approach. These tools [13], [14], [15], [16] use an iterative approach that is similar to adapter synthesis. They first generate a candidate fix to a repair problem in the generate step. Next, they validate the correctness of the candidate fix in a validate step by running a set of test cases. The correct fix should pass all available tests. A reference implementation-based repair tool shares other similarities too with these tools. Code Phage [14] attempts to fix bugs in binary code by automatically transferring code from a reference (referred to as *donor*) into a target (referred to as *recipient*) implementation. A key difference between our proposed research and Code Phage is our use of synthesis of a wrapper around the reference implementation that makes our adapter synthesis-based program repair more flexible in its choice of reference implementations it can work with. Another tool with which we share our intuition in proposing the use of existing reference implementations for program repair is SCRepair [15]. SCRepair makes use of metrics that establish the similarity and differences between code fragments. We plan to reinterpret their metrics for binary code and use them for ranking candidate reference implementations to address scalability challenges involved with reference implementation-based binary program repair. The second approach to automated program repair is semantics-driven repair. This approach makes use of a formal specification of a repair

problem and uses it to find a solution that is guaranteed to solve the repair problem. A number of semantics-driven repair tools [17], [18] make use of oracle-guided program synthesis. Our proposed approach shares a similar intuition with these tools and makes use of a target implementation as the oracle. More specifically, our proposed approach shares the intuition with Angelix [19] that symbolic execution can be used to derive semantic information from a target implementation. But, we also make use of a reference implementation to restrict the space of the program synthesis problem. Our proposed approach lies in the middleground between the *generate-and-validate* and *semantics-driven* approaches.

Several benchmarks [20], [21], [22], [23] have also been developed used for comparing different automated program repair tools. While some of these (such as the ManyBugs and IntroClass [23]) can be compiled to binary code and used to evaluate binary program repair tools, the evaluation would be susceptible to compiler optimizations and choice of compiler. We plan to establish similar benchmarks that can evaluate binary program repair tools against challenges unique to the binary program analysis domain.

III. Proposed Research

A. Motivating Example

Returning back to the example presented in Figures 1, 2, it is clear that we can repair the buffer underflow in `i2c_ASN1_INTEGER` in OpenSSL (the target implementation) by substituting its insecure implementation of two's complement with the secure one in Botan (the reference implementation). A detector can be placed in the adapter search step so that it does not try to find substitutions on control-flow paths in `i2c_ASN1_INTEGER` that have a buffer underflow. This change in adapter search lies at the core of our proposed approach to reference implementation-based program repair. It weakens the equivalence between the target and reference implementation so that undesirable behaviors in the target implementation are excluded when it is substituted with the reference. With a specification of undesirable behavior in the target implementation that should not be substituted, adapter synthesis will search for an adapter that repairs the undesirable behavior in the target while substituting it with the reference. Figure 3 shows the interfaces of the insecure fragment for two's complement implementation in OpenSSL and the secure fragment of two's complement implementation in Botan. Figure 3 also shows the interface adaptation needed to substitute the insecure fragment with the secure one. This substitution requires us to convert a buffer represented as a sequence of `unsigned char` bytes specified in the OpenSSL implementation in `a->data` with the buffer size specified in `a->length` to a `std::vector` class object in the object named `contents` in the Botan implementation. The `_M_impl._M_start` and `_M_impl._M_finish` pointers point to the beginning and end of the buffer internal to `std::vector` class object respectively.

Another adaptation operator needed to make this substitution possible is one that maps all writes to buffer in a reference fragment to a corresponding buffer in the target fragment. The interface to the reference implementation in this example uses its arguments for both input and output, whereas the the target implementation interface has a separate output parameter (`out`). Such repair can be done by extending adapter synthesis [3]. This example shows the need to extend adapter synthesis to allow adaptation modulo undesirable behavior to make it practically useful for binary program repair. Sharma et al. implement adapter synthesis using symbolic execution and search for adaptable substitutability by treating the inputs to the target implementation as symbolic inputs. In this example, treating the `a->length` input as symbolic would allow the input to be of any arbitrary size. Even when adaptation modulo undesirable behavior finds the right adapter, proving its correctness for inputs of arbitrary length can be challenging in a limited time budget. In such cases, it can be useful to bring in context-specific information from the target as part of the repair process. In this example, it is likely that most applications use the `i2c_ASN1_INTEGER` method on values up to 8 bytes long. Using this pre-condition on the `a->length` input parameter to the target implementation can make the correctness-checking step of adapter synthesis faster. Using more of the target context can help us discover such pre-conditions and perform the adaptation only for useful inputs instead of all possible inputs to the target implementation. Figuring out the right buggy target fragment is another extension needed to adapter synthesis.

We plan to extend our previous work by considering program repairs that take more target context into account when doing the repair. While our assumptions would allow the fault to be located precisely to a single instruction, it would be unclear how much of the preceding binary fragment should be considered as the target. We plan to develop techniques that allow substitution of arbitrary target code fragments with a reference. For a large-scale evaluation, we plan to use programs from the DARPA Cyber Grand Challenge (CGC). The DARPA CGC binaries [24] consist of a set of vulnerable programs written by experts in the Cyber Grand Challenge competition hosted by DARPA. The programs are written to resemble real-world software and represent a mix of difficulties with finding vulnerabilities. The DARPA CGC benchmark set also includes inputs to trigger each vulnerability in every program. Since the objective of binary program repair is not to find vulnerabilities but to fix them, the DARPA CGC benchmark set serves as a good starting point for evaluating binary program repair.

IV. Conclusion

The purpose of this research is to build a binary program repair tool that uses existing fragments of binary code as reference to repair functional and security bugs. This research builds on a previously developed technique called adapter synthesis to allow for differences between the interfaces of the target code we want to substitute and the reference code we wish to substitute it with. We plan to evaluate our technique on the DARPA CGC benchmark set of binaries. This benchmark set approximately reflects the complexity seen in real-world

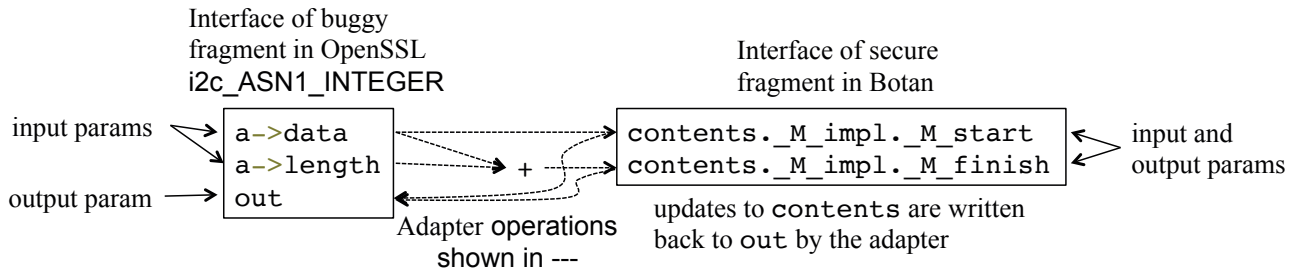

Fig. 3. Adaptation needed to substitute the insecure two's complement implementation in OpenSSL with a secure fragment of the same implementation in Botan

software. Allowing program repair at the binary-level not only reduces developer effort in fixing bugs but also frees end-users from depending on developers for bug fixes.

REFERENCES

[1] H. Krasner, "The cost of poor quality software in the us: A 2018 report," Consortium for IT Software Quality, Tech. Rep., 10 2018.

[2] "CVE-2016-2108." Available from MITRE, CVE-ID CVE-2016-2108., 1 2016. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2108

[3] V. Sharma, K. Hietala, and S. McCamant, "Finding substitutable binary code for reverse engineering by synthesizing adapters," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, April 2018, pp. 150–160.

[4] V. Sharma, K. Hietala, and S. McCamant, "Finding substitutable binary code by synthesizing adapters," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[5] V. Sharma, K. Hietala, and S. McCamant, "Finding substitutable binary code by synthesizing adapters," University of Minnesota – Twin Cities, Tech. Rep., 10 2017.

[6] H. Massalin, "Superoptimizer: A look at the smallest program," in *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ser. ASPLOS II. Los Alamitos, CA, USA: IEEE Computer Society Press, 1987, pp. 122–126. [Online]. Available: http://dx.doi.org/10.1145/36206.36194

[7] R. Joshi, G. Nelson, and K. Randall, "Denali: A goal-directed superoptimizer," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, ser. PLDI '02. New York, NY, USA: ACM, 2002, pp. 304–314. [Online]. Available: http://doi.acm.org/10.1145/512529.512566

[8] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat, "Combinatorial sketching for finite programs," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2006, pp. 404–415. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168907

[9] M. Jacob, M. H. Jakubowski, P. Naldurg, C. W. N. Saw, and R. Venkatesan, "The superdiversifier: Peephole individualization for software protection," in *Advances in Information and Computer Security: Third International Workshop on Security, IWSEC 2008, Kagawa, Japan, November 25-27, 2008. Proceedings*, K. Matsuura and E. Fujisaki, Eds. Berlin, Heidelberg: Springer, 2008, pp. 100–120.

[10] Z. Manna and R. Waldinger, "A deductive approach to program synthesis," *ACM Trans. Program. Lang. Syst.*, vol. 2, no. 1, pp. 90–121, Jan. 1980. [Online]. Available: http://doi.acm.org/10.1145/357084.357090

[11] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 215–224. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806833

[12] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[13] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, p. 54, 2012.

[14] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," *SIGPLAN Not.*, vol. 50, no. 6, pp. 43–54, Jun. 2015. [Online]. Available: http://doi.acm.org/10.1145/2813885.2737988

[15] T. Ji, L. Chen, X. Mao, and X. Yi, "Automated program repair by using similar code containing fix ingredients," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, June 2016, pp. 197–202.

[16] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "Does genetic programming work well on automated program repair?" in *2013 International Conference on Computational and Information Sciences*, June 2013, pp. 1875–1878.

[17] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 772–781. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486890

[18] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 448–458. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818811

[19] ——, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 691–701. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884807

[20] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 437–440. [Online]. Available: http://doi.acm.org/10.1145/2610384.2628055

[21] S. H. Tan, J. Yi, Yulis, S. Mechtaev, and A. Roychoudhury, "Codeflaws: A programming competition benchmark for evaluating automated program repair tools," in *Proceedings of the 39th International Conference on Software Engineering Companion*, ser. ICSE-C '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 180–182. [Online]. Available: https://doi.org/10.1109/ICSE-C.2017.76

[22] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, "How developers debug software the dbgbench dataset: Poster," in *Proceedings of the 39th International Conference on Software Engineering Companion*, ser. ICSE-C '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 244–246. [Online]. Available: https://doi.org/10.1109/ICSE-C.2017.94

[23] C. L. Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of c programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, Dec 2015.

[24] Trail of Bits, "DARPA Challenge Binaries on Linux, OS X, and Windows," 02 2019.