# Poster: Challenges and next steps in binary program analysis with angr

Christophe Hauser
*Information Sciences Institute*
*University of Southern California*
*hauser@isi.edu*

Yan Shoshitaishvili
*Arizona State University*
*shoshitaishvili@asu.edu*

Ruoyu Wang
*University of California, Santa Barbara*
*fish@cs.ucsb.edu*

***Abstract*—In the past decade, academic interest for binary program analysis models, tools and techniques has received increasing interest. As a result, recent advances have been pushing the limits forward, as demonstrated by the Cyber Grand Challenge, a competition organized by DARPA, as well as recent academic work in the field. However, despite this progress, a number of challenges remain to be addressed in order to solve current and future problems related to security and privacy in binary programs and applications. We give a description of some of these challenges, along with practical suggestions for future work and improvement over the current state of the art.**

## 1. Introduction

Despite the prominence of interpreted languages and open-source projects, the ability to assess the security of programs and applications based on the analysis of their binary executable form remains an important problem for solving vulnerability discovery and reverse engineering challenges. Reasons motivating binary analysis include the presence of (or dependence on) proprietary code, abstract differences between source code and assembly representations, and the existence of bugs in compilers and tool chains, among others. Increasing academic interest over the last decade, led to a number of open-source binary analysis platforms freely available on the internet [1], [2], [3], [4], [5], [6], among which `angr` is a framework that we built in order to experiment with new and existing analysis techniques in a composable way, and to increase the reproducibility of academic models and prototypes [7], [8].

However, while recent advances pushed the limits forward, a number of challenges remain to be addressed in order to apply those new techniques on real-world programs in an extensible and scalable manner. Such challenges include the analysis of third party libraries, low-level programmatic concepts such as those found in kernel and embedded code, as well as modeling high level abstractions necessary to reason about certain classes of bugs.

### 1.1. Incremental vulnerability discovery

The vast majority of software projects rely on third party code and libraries, and by design, need to trust this external code. However, even widely used and tested projects and libraries suffer from security bugs, whether these are benign (*i.e.,* have been inserted accidentally) or malicious (*i.e.,* were voluntarily inserted in the code by an attacker). Current approaches to detect security bugs are typically involved *after* the software release, and in some cases, it takes up to *years* until critical security bugs are found, because of the difficulty of manual analysis, and the inherent limitations of automated tools, such as the trafe-offs one has to make between accuracy and coverage. Closing the gap between code size/complexity and the level of accuracy that can be achieved with automated tools is therefore a research direction of interest. Meanwhile, reasoning about large code bases such as the code of external libraries remains a daunting task without a better understanding of the possible execution contexts or code paths of interest within the binary. Attempts to improve coverage based on fuzzing, symbolic execution and mixed static/dynamic analysis [9], [10], while successful in some cases, are inherently bound to the same coverage limits. One way to move forward is to reduce the analysis scope, either in terms of space, or time in the development life-cycle of a software product. For instance, an important aspect to consider is that most vulnerabilities are introduced by single commits affecting only isolated parts of the modified code. In this context, incremental software testing can be leveraged to help model the effects of increments (*i.e.,* relatively small changes in the code, such as the effects of a patch) over the execution environment of the evaluated software, and to derive new techniques and heuristics to detect dangerous code changes as part of complex code bases.

### 1.2. Analyzing closed-source embedded firmware

The current state-of-the-art in binary firmware analysis leaves a number of open problems untouched. For instance, embedded devices sometimes suffer from incorrect implementations of cryptographic functionalities. Similarly, memory corruption bugs are still regularly found and exploited by attackers, and firmware is no exception. Automatically detecting vulnerabilities in embedded firmware remains challenging, because of the low-level nature of the software abstractions involved, but also because of the specificity of the hardware in use. For example, reasoning

about concurrency (*i.e.,* multithreading) or analyzing kernel code paths subject to preemptive behavior (*e.g.,* interruptions), among other problems, are challenging due to the complexity to model the required abstractions using current analysis techniques. Steps in this direction would extend the coverage of current automated analysis systems. Furthermore, modern embedded devices often rely on hardware implemented using System on Chip (SoC) devices. Such devices generally use known architectures, but incorporate additional circuitry which specifications often are not publicly available. The ability to model interactions with such hardware at the software level, without the need to interact with the actual hardware, is an important factor towards automated analysis of embedded firmware. Steps forward in this direction include more accurate modeling of asynchronous behavior by borrowing concepts from distributed computing [11] as well as the extension of current analysis platforms with firmware re-hosting capabilities, automated reasoning about instructions/library calls/system calls whose specifications are not available, and hardware/OS interface identification.

### 1.3. Reducing semantic gaps

Past and current research in vulnerability discovery mostly focus on modeling particular classes of bugs, such as *buffer overflows* or *use after free*, but in practice, fail to identify higher-level logic bugs due to their complexity, causing security vulnerabilities to remain undiscovered. This is especially true at the binary-level, because of the lack of semantic information in the recovered assembly, increasing the gap with higher-level program logic. Example of higher-level logic bugs include incorrect interactions between functions, information leakage, defects in cryptographic implementations and backdoors. On the one hand, attempts were made to bridge this semantic gap by recovering *e.g.,* type information from disassembly [12], while on the other hand, past works have successfully leveraged code similarity measurement techniques based on syntactic [13] and semantic [14] properties to find bugs across binaries of different architectures. With the advent of online communities, programmers leverage online forums to often exchange "code snippets" representing solutions to common algorithmic problems. Such code snippets typically represent only part of a function, and are inserted as-is, with very little to no modifications, in the final code. In practice, it is reasonable to expect that a subset of this code is subject to security vulnerabilities, and that it may be found in both open-source an proprietary code. Furthermore, similarities also intrinsically exist between bugs (*e.g.,* buffers size depending on user input). Drawing on the success of code similarity techniques, and by leveraging the large language and architecture support offered by the LLVM compiler under the same intermediate representation, further identification of semantic bug patterns "at scale" is a promising direction for extending the scope of vulnerability analysis models. In particular, by leveraging machine learning in combination with program analysis techniques[1], insecure coding patterns may be learned from communities and translated to an intermediate representation where the knowledge of such insecure patterns can later be used as input for binary program analysis, and to identify new insecure patterns in closed-source software.

## References


[1] "Unix-like Reverse Engineering Framework and Command-line Tools". http://radare.org.

[2] "angr, the Next Generation Binary Analysis Platform from UC Santa Barbara!". http://angr.io.

[3] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A Binary Analysis Platform," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 463–469.

[4] "BinNavi is a binary analysis IDE that allows to inspect, navigate, edit and annotate control flow graphs and call graphs of disassembled code". https://github.com/google/binnavi.

[5] C. Heitman and I. Arce, "BARF: A Multiplatform Open Source Binary Analysis and Reverse Engineering Framework," in *XX Congreso Argentino de Ciencias de la Computación (Buenos Aires, 2014)*, 2014.

[6] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: Binary Analysis for Computer Security," 2013.

[7] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "SOK:(State of) The Art of War: Offensive Techniques in Binary Analysis," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 138–157.

[8] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware." in *NDSS*, 2015.

[9] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[10] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," 2017.

[11] J. Hatcliff, J. Corbett, M. Dwyer, S. Sokolowski, and H. Zheng, "A Formal Study of Slicing for Multi-threaded Programs with JVM Concurrency Primitives," in *International Static Analysis Symposium*. Springer, 1999, pp. 1–18.

[12] J. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled Reverse Engineering of Types in Binary Programs," 2011.

[13] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient Cross-architecture Identification of Bugs in Binary Code," in *Proceedings of the 23th Symposium on Network and Distributed System Security (NDSS)*, 2016.

[14] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging Semantic Signatures for Bug Search in Binary Programs," in *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014, pp. 406–415.


1. Such as code reuse detection, code diffing, function inlining and automatic function summarization.