Using Binary Analysis Frameworks: The Case for BAP and angr

Chris Casinghino[1], JT Paasch[1], Cody Roux[1], John Altidor[2], Michael Dixon[3], and Dustin Jamner[4]

[1] Draper
{ccasinghino,jpaasch,croux}@draper.com
[2] Cambridge Semantics Inc.
john.altidor@cambridgesemantics.com
[3] Los Alamos National Laboratory
mdixon@lanl.gov
[4] Northeastern University
jamner.d@husky.neu.edu

Abstract. Binary analysis frameworks are critical tools for analyzing software and assessing its security. How easy is it for a non-expert to use these tools? This paper compares two popular open-source binary analysis libraries: BAP and angr, which were used by two of the top three teams at the DARPA Cyber Grand Challenge. We describe a number of experiments to evaluate the capabilities of the two tools. We have implemented a value-set analysis and a call graph comparison algorithm with each tool, and report on their performance, usability, and extensibility for real-world applications.

Keywords: BAP · angr · binary analysis · differential analysis · cyber security.

1 Introduction

If you want to analyze the version of your program that actually gets executed, you may need to examine its binary code directly. There are a variety of tools to help with this task. Some of these tools are general libraries that can help you build your own custom program analyses.

In this paper, we compare two popular, open-source binary analysis libraries: BAP [5] and angr [13]. We examine how each library constructs call graphs (CGs) and control flow graphs (CFGs). We have implemented a value-set analysis (VSA) and an algorithm to compare call graphs in both BAP and angr, and assess how easy it is to build real-world program analyses using each.

Our contributions include the following:

This work is sponsored by ONR/NAWC Contract N6833518C0107. Its content does not necessarily reflect the position or policy of the US Government and no official endorsement should be inferred.

– We detail some technical differences in the way BAP and angr identify function starts, as well as how they construct CGs and CFGs.
– We provide a first-hand account of building custom analyses with these libraries, and we profile the tools we built.
– We conclude by identifying the strengths and weaknesses of each tool, and give our impression of their suitability for building sound, static program analyses.

The data from our analyses is publicly accessible at https://github.com/draperlaboratory/cbat_tools/tree/master/bap-angr.

2 BAP and angr Overview

BAP and angr both begin by lifting a binary program to an intermediate representation (IR), and then analyzing that IR. BAP lifts to its own IR, the BAP Intermediate Language (BIL), while angr lifts to VEX, which is the IR used by Valgrind. The differences between BIL, VEX, and other potential IR choices are not the focus of this paper, but have been studied elsewhere [9].

Once a binary has been lifted to the IR, you can use built-in BAP or angr program analyses, or write your own tools to explore the lifted program. BAP is written in OCaml and angr is written in Python; it is easiest to write your own tools in the host language.

The idiomatic use of each tool is similar: first you load a binary into a "project," and then perform your own analysis. For example, you might begin by generating a CFG. In angr:

```
import angr

exe = "/bin/true"
project = angr.Project(exe)
cfg = project.analyses.CFGFast()
# Now do something with the CFG...
```

In BAP, the process is similar. In the following example, we select byteweight [4] to identify function starts, then we load the program into a project, retrieve the lifted IR program, and generate a CG:

```
open Core_kernel.Std;;
open Bap.Std;;

let exe = Project.Input.file "/bin/true";;
let byteweight = Rooter.Factory.find "byteweight";;
let Ok proj = Project.create exe ?rooter:byteweight;;
let lifted_prog = Project.program proj;;
let cg = Program.to_graph lifted_prog;;
(* Now do something with the CG... *)
```

Both libraries are easy to use in a REPL. For instance, you can import angr in a Jupyter console to explore a particular binary, and you can import BAP into `utop`, or the `baptop` REPL that BAP provides.

For batch mode, angr analyses can be written as straight-forward Python scripts that import `angr` and proceed from there. BAP offers a modular plugin architecture: each plugin makes a pass over the program, where it extracts information, alters the IR, or performs other tasks. Passes can be chained together.

Both tools offer a reasonably easy point of entry into programmatic binary analysis, with library functions for common tasks such as generating a CG or CFG. The communities for both projects are extremely helpful and responsive, to the extent that most of our technical questions about the tools were immediately answered.

For the experiments below, we worked on an Ubuntu 16.04.4 VM (Linux 4.4.0-87 and GCC 5.4.0) with 16Gb of memory and eight 2.2 GHz Broadwell family 6, 61 processors. We report results for angr 7.8.9 with vanilla Python 2.7, BAP 1.5.0 with OCaml 4.05.0. We also experimented with running angr with PyPy 6.0 rather than Python. We found PyPy to be less efficient for small programs and more efficient for larger ones. We ran BAP with a `--no-cache` flag, but normally BAP caches disassembly and other information, so repeat runs are significantly faster.

We estimated each library's resource overhead by loading an empty C program into a new project. On average, BAP took a half second with a max resident set size (RSS) of 84MB, while angr took one second with a max RSS of 82MB.

3 Extracting and Using Control Flow Data

A basic requirement for analyzing or transforming code in any non-trivial manner involves getting data and control flow information. For binary code, this can be a complex operation, and both BAP and angr offer built-in support. In this section, we compare the CFGs and CGs recovered by each tool, and describe a CG-based analysis that we implemented in both BAP and angr as a comparison of their capabilities and performance.

3.1 Control Flow Graphs

Both tools make CGs and CFGs easy to generate and manipulate. However, they make different choices about how to lift various binary constructs, making a direct comparison challenging.

First, angr generates a CFG for the whole program, while BAP generates one per function. Additionally, the two tools represent binary control flow differently. BAP's CFGs include "dummy nodes" at branch points that do have a direct analogue in the original binary but are created to make uplifiting more convenient. angr does not create similar nodes, but sometimes coalesces basic blocks. Neither angr nor BAP resolve most indirect jumps, with the notable exceptions of jump tables in angr, which are resolved using a heuristic. Some of

these issues, and a detailed analysis of the accuracy of CFG construction for several binary analysis tools including (older versions of) BAP and angr is explored in detail by Andriesse *et al* [1].

3.2 Call Graphs

We compared BAP and angr's features for working with CGs in two ways. First, we developed a script to directly compare the CGs produced by each tool, and report here on their similarity. Second, we selected a CG-based program analysis from the literature and implemented it twice, using each tool as a library.

Comparing CG Accuracy Both tools make it simple to recover a program's CG and output it in the DOT graph description language. We implemented a simple algorithm for comparing this output:

- Start with the program entry point of both graphs.
- Recursively fetch the reachable nodes from that point, excluding already seen nodes.
- Compare the reachable nodes at step n as sets between the graphs.

While the tools agree well on small examples, differences appear quite early in the CGs of larger programs. For example, we get around 6% difference 1 step below `main` in the CG for the `grep` executable, and the errors snowball at lower levels up to a significant fraction. The cause for these discrepancies is unclear, but may be related to disagreements between what the tools consider to be reachable function calls during CFG construction (see again [1]).

Implementing a CG-based Program Analysis One common use of CFGs and CGs is to judge the similarity of two programs [6]. As a basis on which to evaluate the usability and performance of each tool, we selected a well-regarded algorithm for estimating the similarity of two CGs [7] and implemented it both as a BAP plugin and as an angr script.

Implementation of this algorithm was mostly straightforward. One obstacle was that the BAP's plugin interface is designed to manipulate a single program at a time. However, BAP does support saving a program's `Project` data structure to disk. Thus, we designed our plugin to take one binary from the command line and compare with a previously saved `Project` structure.

For evaluation, we took 11 GNU applications of varying sizes and compiled them on two optimization levels (`-O0` and `-O1`). We used the analysis to compare the two versions of each program. Table 1 contains the results. Each column lists BAP's and angr's results respectively, separated by a slash. A long dash indicates that the analysis did not complete within 35 minutes.

The results show that our BAP OCaml implementation runs approximately 15% faster than our angr Python implementation on average, despite constructing larger CGs. Profiling revealed that the running time in both cases is dominated by a standard graph matching algorithm that the analysis uses, and thus

Table 1. CG construction performance (BAP/angr)

Exe	Time (secs)	Max RSS (Kb)	Graph size
bison	1181/824	15182/16847	7717/6078
gawk	158/2004	20253/25680	5760/8661
grep	89/581	7184/7528	3339/4002
gnuchess	158/82	20253/10815	5760/868
gzip	58/162	7391/6122	2065/1706
less	113/—	3741/—	4142/—
make	313/552	15812/10440	4835/4436
nano	729/454	8060/10620	6500/4618
screen	699/964	12980/12054	7466/6094
sed	27.6/—	4536/—	2320/—
tar	—/1321	—/8139	—/6520

speaks more to differences in the efficiency of OCaml and Python code than to differences in BAP and angr. The running time scales with the size of the graphs (reported as a sum of the number of nodes and edges). Substantial differences in graph sizes are a result of the discrepencies in CG recovery described above, and the similarity scores computed by the algorithm also differed as a result.

4 Value-Set Analysis

As an example of a standard, more complex use of a binary analysis toolkit, we experimented with value-set analysis (VSA) in both BAP and angr [2, 3]. The angr tools include an experimental Value Flow Graph (VFG) module that performs a VSA. It annotates the CFG with sets of values that registers and memory locations can take on at various points during execution. At the time of writing, BAP does not ship with a comparable module, so we implemented our own VSA plugin using BAP's built-in support for abstract interpretation.

Both implementations perform abstract interpretation, but use slightly different abstract domains. Our VSA plugin for BAP uses *circular linear progressions* [8, 12]. The implementation found in angr uses an extension of *wrapped strided intervals* [3, 10, 11]. These two representations are similar, and the distinction made little difference for our purposes.

To evaluate the two VSA implementations, we used them to resolve indirect jumps that BAP and angr CFG construction missed. We profiled runs on four small test programs that contain indirect jumps that require some insight to resolve. The results are in Table 2.

Table 2. Indirect jump resolution via VSA (BAP/angr)

Exe	Time (secs)	Max RSS (Mb)	Resolved Jumps
Prog A	0.73/1.21	124/88	5 of 5 (100%) / 4 of 5 (80%)
Prog B	0.72/1.59	124/91	8 of 8 (100%) / 7 of 8 (88%)
Prog C	0.71/1.85	124/93	8 of 8 (100%) / 7 of 8 (88%)
Prog D	0.70/4.20	124/104	8 of 8 (100%) / 8 of 8 (100%)

We found that our BAP VSA plugin resolved all jump targets, while angr's missed one in all but the last case. On further inspection, it looks like angr's VFG module has a bug that causes it to discard the contents of previous value sets after successive iterations, thereby resulting in an under approximation. By stopping after each iteration, we were able to observe that angr actually resolved some of the missing jumps before discarding the results for the next iteration.

The BAP plugin runs faster, but uses more memory at a constant level for our toy programs, while angr runs more slowly, but uses less memory. Neither implementation scales well to larger programs. When run on the GNU utilities described in the previous section, we typically encountered issues ranging from memory exhaustion to unsupported constructs before the analysis completes.

As implementors, we found that BAP gave us more confidence in the VSA results than angr. The simple Python interface and VFG module in angr made it easy to get started and obtain initial results. However, the lack of documentation and the presence of apparent bugs made it difficult to verify the correctness of the analysis we built on angr's capabilities. By contrast, since BAP ships with no VSA, it was a fair amount of work to build our own. Nevertheless, BAP's module-based documentation and the static checking provided by its use of the OCaml type system gave us more confidence that we were using it correctly.

5 Conclusion

Both BAP and angr enable analysis of binaries, providing a convenient interface that hides the technical details of the binary formats and ISAs. In addition, they each supply a suite of pre-built analyses to jump start the process.

We compared these tools in several ways. We described the process of implementing program analyses using them, and differences in the call graphs and control flow graphs they recover from binary programs. We implemented two representative program analyses using each tool, and examined their usability and performance.

In terms of resource usage, BAP is often more efficient, but not drastically so. We found that angr was easier than BAP to pick up quickly and begin experimenting with, and includes more-built in analyses. By contrast, BAP required us to do more work to get started, but its comprehensive module-based documentation gave us more confidence that we were using the tool correctly, even as new users.

References

1. Andriesse, D., Chen, X., van der Veen, V., Slowinska, A., Bos, H.: An in-depth analysis of disassembly on full-scale x86/x64 binaries. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 583–600. USENIX Association, Austin, TX (2016)
2. Balakrishnan, G., Reps, T.: Analyzing memory accesses in x86 executables. In: International Conference on Compiler Construction (CC). pp. 5–23. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
3. Balakrishnan, G., Reps, T.: Wysinwyx: What you see is not what you execute. ACM Trans. Program. Lang. Syst. **32**(6), 23:1–23:84 (Aug 2010)
4. Bao, T., Burket, J., Woo, M., Turner, R., Brumley, D.: Byteweight: Learning to recognize functions in binary code. In: USENIX Security Symposium (2014)
5. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: Bap: A binary analysis platform. In: Proceedings of the 23rd International Conference on Computer Aided Verification. pp. 463–469. CAV'11, Springer-Verlag, Berlin, Heidelberg (2011)
6. Chan, P.P.F., Collberg, C.: A method to evaluate cfg comparison algorithms. In: 2014 14th International Conference on Quality Software. pp. 95–104 (Oct 2014)
7. Hu, X., Chiueh, T.c., Shin, K.G.: Large-scale malware indexing using function-call graphs. In: Proceedings of the 16th ACM Conference on Computer and Communications Security. pp. 611–620. CCS '09, ACM, New York, NY, USA (2009)
8. Källberg, L.: Circular linear progressions in SWEET. Tech. rep., Mlardalen University, Embedded Systems (2014)
9. Kim, S., Faerevaag, M., Jung, M., Jung, S., Oh, D., Lee, J., Cha, S.K.: Testing intermediate representations for binary analysis. In: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering. pp. 353–364. ASE 2017, IEEE Press, Piscataway, NJ, USA (2017)
10. Lee, J., Avgerinos, T., Brumley, D.: Tie: Principled reverse engineering of types in binary programs. In: Network and Distributed Systems Security Syposium (NDSS). Internet Society (01 2011)
11. Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: Signedness-agnostic program analysis: Precise integer bounds for low-level code. In: Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings. pp. 115–130 (2012)
12. Sen, R., Srikant, Y.N.: Executable analysis using abstract interpretation with circular linear progressions. In: Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign. pp. 39–48. MEMOCODE '07, IEEE Computer Society, Washington, DC, USA (2007)
13. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Krügel, C., Vigna, G.: SOK: (state of) the art of war: Offensive techniques in binary analysis. In: IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016. pp. 138–157 (2016)