How can dynamic analysis features be used to improve static Abstract Interpretation?

Marvin Weiler
https://weiler.rocks
marvin.weiler@tu-dortmund.de

Abstract

Modern trends in software engineering such as dynamic programming features, extensive use of third-party libraries or various different execution environments are posing hard challenges for static analysers. Therefore the use of Abstract Interpretation is limited. Dynamic analysis methods are often not constrained by these practices. Therefore, it seems obvious to combine Abstract Interpretation with different dynamic analysis methods to mutually compensate each disadvantages. This elaboration will recap on a few ways to combine Abstract Interpretation and dynamic analysis. Also problems that still need to be solved are shown.

2012 ACM Subject Classification

Keywords and phrases static Abstract Interpretation, dynamic program analysis

1 Introduction

Modern applications are no longer created like conveyor belts that have a linear execution flow, use only simple data structures and a small enclosed set of standard libraries. To improve productivity and portability, software engineers interact with large, complex libraries, frameworks and vastly different runtimes to provide the scaffolding on which an application can be created. Traditionally these frameworks and libraries are using a lot of dynamic language features which are known to not work well with Abstract Interpretation. In some cases the libraries are not even written in the same languages as the applications using them. Often Abstract Interpretation finds itself in a quandary, ignoring some of the dynamic features violates the soundness assumption, analysing them increases the calculation time by large factors and often also produces a lot of false positives. Therefore dynamic analysis methods can be used to refine the Abstract Interpretation process and help with solving those burdens. This elaboration will give a short insight how dynamic program analysis works and then show some methods on how to combine Abstract Interpretation and dynamic analysis.

2 Dynamic analysis

Dynamic analysis is the analysis of the properties of a running program. In contrast to static analysis where textual program representation is over-aproxiamted, dynamic analysis derives properties that are valid for at least one program execution by inspection of the running program. The program can be analysed by observing e.g its memory state or transitions. Dynamic analysis is sometimes also called *input-centric* or *program-centric* [3] due to the stark relation of program inputs to the resulting analysed program behaviour. The analysis precision is also better than with static analysis methods because all found properties are from actual executions of the program, resulting in less false positives. As a dynamic analysis usually inspects one long program trace through the whole program semantic dependencies widely separated in this trace can be found. In addition dynamic analysis is often much faster than static analysis due to the reduced scope of analysed program traces and properties.

3 Combining static and dynamic analysis methods

In the following section a variety of techniques to refine Abstract Interpretation with dynamic analysis methods, working around some shortcomings of Abstract Interpretation, are presented.

3.1 Opaque code analysis

In modern software development most of the code that gets bundled in a shipped application is not written by the actual developers but included from third party libraries. These libraries are quite often several orders of magnitude larger than the actual code written by the developers. Abstract interpreters need to decide to either include the library's code in the analysis and as a result increase the computation time significantly, or to not analyse the library code and, as a result of missing some code-behaviour, produce possibly unsound results.

A obvious workaround for this problem is to provide an abstract model of the public accessible library functions. The clear disadvantage of this approach is the ginormous overhead that is introduced by modelling libraries by hand due to their vast size and the huge amount of libraries used. There are also various approaches to create this models automatically from specification of the code behaviour [18, 21]. This is unfortunately not always possible due to the lack of appropriate specification or sometimes no usable documentation at all.

Another approach is to learn the expected behaviour via dynamic execution [15, 7], but these methods are either unsoundly modelling a subset of the executions or are confronted with an explosion of possible traces to analyse. Therefore, specific analysers for specific Libraries are needed. This concludes a lot of manual work too.

In Javascript the built-in standard library is provided by the host environment e.g. webbrowsers. This library is for the most parts not implemented in Javascript and the different host environments use a multitude of different languages to implement the standard library. Googles v8[1] ist written in C++, Nashorn [2] is written in Java and Mozillas Spidermonkey [3] is written in C++ and Rust. An abstract interpreter is in need to understand all of these quite different languages in order to soundly abstract a Javascript program which uses the standard library.

3.1.1 Sample-Run-Abstract

J. Park et al. propose an automatic modelling methods for opaque Javascript code [11]. Although the mechanism cannot guarantee a sound over-approximation for all of the builtin library functions, they are able to compute models for a large part of the Javascript standard library. Compared to the previous state of the art analysis methods based on manually created models, they increased the analysis precision and also found errors in the manual generated models. They named their new model the Sample-Run-Abstract (SRA) model. For SRA to work, three preconditions for an abstract analyser A and a dynamic analyser $\mathfrak{Z}$ must be given:

- An abstract state domain $\hat{S}$ which forms a complete lattice.

[1] https://v8.dev/
[2] https://openjdk.org/projects/nashorn/
[3] https://spidermonkey.dev/

- An abstraction function α and a concretization function γ.
- Either A or $\mathfrak{Z}$ are required to identify the sourcecode corresponding to a given program point.

The analysis then starts as a usual Abstract Interpretation with the entrypoint of the program. Abstract Interpretation continues until the execution reaches a program point $c \in C$ where a library function is used.

Sample $\hat{S} \to \alpha(S)$ At this point the current abstract state $\hat{s} \in \hat{S}$ is converted to a finite subset of elements $s \subseteq \gamma(\hat{s})$. If $\hat{s}$ represents a concrete value or a small set of values $s = \gamma(\hat{s})$ otherwise a *Sample* heuristic chooses representative values of all the possible values in $\gamma(\hat{s})$.

Run $C \times S \to S$ For each *Sampled* value and the current program point, executable code is generated. The dynamic interpreter executes this code, takes a snapshot of the program state after execution and returns this state.

Abstract $\alpha(S) \to \hat{S}$ For each of the given concrete state α generates a corresponding abstract state. These states are joined and one abstract state representing a program point after the library code is returned.

If *Sample* has to choose a set of values from the abstract state then the dynamic analyser executes only a subset of the possible program traces. Therefore, the resulting states after *Run* might be a under-approximation of possible states. The *Abstract* step then consequently produces an unsound abstract state. To cope with this problem J. Park et al. recommend a optional *Broaden* [4] heuristic similar to the widening operators that should be applied to cope with the unsoundness.

3.1.2 Mostly-Concrete Interpretation

The *SRA* aproach ist suitable for library code that is called from the application but some problems are present in applying *SRA* to (enterprise) frameworks such as Spring [5] or Struts [6]. Frameworks are typically using a lot of programming techniques that are troublesome for Abstract Interpretation e.g. reflections, dynamic dispatch or embedded DSLs. Commonly the framework-code calls into functions located in the application code, defined in some sort of configuration. In most frameworks this configuration is statically parseable and not dependent on run-time values [19]. Another commonality in projects that use frameworks is a design-pattern called *state separation* "which requires that the application state is opaque to the framework implementation and similarly for framework state and application code"[19, p.4]. As a consequence the program state can be partitioned into a framework state R and an application state $\hat{R}$, forming a complete lattice with concretization function $\gamma(\hat{R})_A$ and abstraction function $\alpha(R)_A$ forming the Galois connection $R \underset{\alpha_A}{\overset{\gamma_A}{\leftrightarrows}} \hat{R}$.

Based on this two assumptions John Toman and Dan Grossman developed the *Concerto* framework [19]. *Concerto* soundly combines a dynamic concrete interpretation with Abstract Interpretation. This combination fits well, as a specific framework routine combined with an application specific configuration usually lead to a single execution trace through the frameworks code. This trace is analysed with a concrete interpreter and as it is executing the

[4] Broaden is similar to a widening operation, as it widens the possible abstract values. For an example of a Broaden heuristic see [11, p. 52].
[5] https://spring.io
[6] https://struts.apache.org/

code with concrete values, the mentioned dynamic language features are no problem. The application code which often contains loops and nondeterministic user input is analysed by Abstract Interpretation. So both analysis methods compensate for each other's weaknesses. To convert between the framework-state and application-state the abstraction function α and concretization function γ could be utilised. This conversion is necessary every time the execution context switches between framework-code and application-code. Given the preconditions this approach is sound but unfortunately infeasible to implement as the concretization and abstraction functions are required to work with possibly infinite large sets of values. To overcome this limitation the authors extended the concrete interpretation to *mostly-concrete interpretation* by allowing only finite sets of values. Additionally, the concretization and abstraction functions are replaced by *domain transformers* defined as the equation 12 and 13 in [19] which are weaker that a Galois connection but can still assure a sound transfer of state between the two domains.

With finite domains, the two domain transformers and the state separation as prerequisites, *Concerto* can soundly combine the abstract and the mostly-concrete interpreter. In case the state separation is violated e.g. application code is accessing a frameworks internal data-structure directly, *Concerto* can still soundly continue the analysis with reduced precision by assuming accessed value as $\top$.

3.2 Dynamic Shortcuts through sealed execution

In recent years more and more applications allow the embedding of dynamic languages that interact with the rest of the program. Examples for this applications are Webbrowsers with their embedded Javascript engines however there are also some areas where the usage of dynamic languages was previously considered unusable. There are now IOT [7] platforms that allow embedding Javascript [8] or Python [9] code in the application. This new application areas boosted the usage of these dynamic languages.

There has been a lot effort put into the analysis of dynamic languages e.g. advanced string domains [1], loop sensitivity [9], or on demand backwards analysis [17]. All of these solutions need to find a trade-off between enough precision and analysis time. Researcher who are using dynamic analysis methods can rely on the already tremendous work put into optimising existing runtimes whereas static interpreters need to provide there own, often slower, abstract runtimes. This leads to a large performance gap between the two analysis approaches. In a comparison between the static analyser SAFE [10] and the dynamic analyser Jaling [14] using the subset of, by both analysable, tests from the SunSpider [10] benchmark the dynamic analyser turned out 34.8x as fast as SAFE [12, Fig. 1].

J. Park et al. worked out a way to instrument this speed advantage into Abstract Interpretation and preserve the soundness guarantee [12]. They are combining the Abstract Interpretation methods with sealed execution which is similar to dynamic symbolic execution. Instead of using symbolic variables to execute the program they are using *sealed values*. A *sealed value* is representing the existence of an abstract value in a program. The *sealed value* does especially not hold information about the content of the abstract value except the abstract value corresponds to exactly one concrete value. It can be seen as a symbol to track

[7] Internet of Things
[8] https://www.espruino.com
[9] https://micropython.org/
[10] https://webkit.org/perf/sunspider/sunspider.html

the use of an abstract value through the code. The run of the program with the concrete *sealed values* is then called a *sealed execution.* [12]

The Abstract Interpretation is then started at the program entry point. When an abstract interpreter encounters a program point which to analyse would reduce precision or possibly increase the analysis time by a large factor, the *sealed execution* can be utilised as a *Dynamic shortcut* through the program part.

Listing 1 Example code with access to a *sealed value*

```
var v = ... // an abstract value
var obj = { p1: v };
var x = 4 + obj.p1;
```

$\hat{D} \to D_\omega \times \hat{D}_\delta$ As a first step the current abstract state gets converted to a symbolic state where each variable in the program gets replaced by a *sealed value*, except its abstract value can be directly mapped to a single concrete value. To get a concrete value from an abstract value the concretization function γ [4] can be utilised.

Execute code Concrete code for the next instructions after the current program point and D_δ is generated and executed until a value represented by *sealed value* is accessed. On access of a *sealed value* the end of a *sealed execution* is necessary to guarantee overall soundness. Note that in listing 1 even though *obj* contains an abstract value v the second line is not an access to a *sealed variable* as the value of *v* is only passed on and not used. Line three is a *sealed access* as the value of *p1.obj (v)* is needed for the addition operation.

$D_\delta \to \hat{D}$ The symbolic state corresponding to the program point just before the access to the *sealed value* is converted back to an abstract state. Concrete values get mapped by applying the abstraction function α and as *sealed values* are just references to abstract values the referenced values from the previous abstract state can be copied.

The Abstract Interpretation then continues until the next opportunity of a dynamic shortcut is reached. The authors found during testing,that when using *dynamic shortcuts* the analyser outperformed the same analyser without *dynamic shortcuts* by 7.81x. Also they found that the number of false positive tests could be reduced by an average of 92 %.

3.3 Determinacy analysis

The above mentioned methods all are similar in the way they are interleaving the Abstract Interpretation with the dynamic analysis part. Therefore, various different converter functions are required to convert between the different domains. *SRA* and the *dynamic shortcut* techniques also require some modifications to the used domains. This makes them harder to implement into existing interpreters. A different approach is not to interleave the static and dynamic analysis part but apply them in sequence. Schäfer et al. developed a method to soundly find determinate (non changing) variables based on dynamic flow analysis of a program [13].. "Roughly speaking, a variable x is determinate at a program point p if x must have the same value, say v, whenever program execution reaches p"[13, p. 1]. Although such information can also be detected statically utilising a call graph this is often not feasible due to dynamic language features such as *eval* [11] [16, 13]. Therefore, the dynamic approach is better suited.

[11] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval

In the methodology proposed by Schäfer et al. first a dynamic flow analysis is done to find determinate variables. To stay sound all possible program paths must be checked.

To guarantee this Schäfer et al. are using *counterfactual execution*. If the decision which branch in a program should be taken during a concrete execution is only dependent on determinate variables then only the chosen branch gets analysed. Otherwise the remaining possible branches are also analysed and after executing the effects of any writing to local variables or the heap are undone and the accessed variables are marked as indeterminate.

In the begin of the analysis all variables are considered determinate. "A variable becomes indeterminate when it is assigned an expression involving other indeterminate variables, but also when it is modified in code that is control dependent on an indeterminate condition and hence not necessarily run in every execution" [13, p. 3]. A variable also gets indeterminate if it is marked indeterminate during *counterfactual execution*.

Listing 2 Code taken from [8] showcasing a use of eval that can be replaced by static code

```
1  ivymap = window.ivymap || {};
2  function showIvyViaJs(locationId) {
3      var _f = undefined;
4      var _fconv = "ivymap[\\'"+locationId+"\\']";
5      try {
6          _f = eval(_fconv);
7          if (_f!=undefined) {
8              _f();
9          }
10     } catch (e) {}
11 }
12 showIvyViaJs ('pc.sy.banner.tcck.');
13 showIvyViaJs ('pc.sy.banner.duilian.');
```

After a full run *determinacy facts* are generated that can be used to transform the program to a simpler to analyse one. This information can be used to replace dynamic language features with static analysable code. In listing 2 the strings passed to the eval function are determinate and so the dynamic analysis produces the following *determinacy facts*:

$$[\![\_fconv]\!]_{14\to6} = \text{"ivymap['pc.sy.banner.tcck.']"} \tag{1}$$

$$[\![\_fconv]\!]_{15\to6} = \text{"ivymap['pc.sy.banner.duilian.']"} \tag{2}$$

The facts provide the information that every time the program reaches line $14_{(1)}$ / $15_{(2)}$ the variable _fconv on line 6 of listing 2 contains either: ivymap['pc.sy.banner.tcck.']$_{(1)}$ or ivymap['pc.sy.banner.duilian.']$_{(2)}$. Based on this information the call to *eval* on line 6 can be replaced with concrete code and easily analysed by Abstract Interpretation.

4 Conclusion

Abstract Interpretation has some well known problems that are no issues in the world of dynamic analysis. The combination of both techniques is a good way to compensate for weaknesses in both approaches. Still some combinations, as in *SRA*, provide possible unsound results but especially in the *SRA* case the pure abstract approach was also potentially unsound due to human errors. Also in some cases a unsound fast result is better than no result or a timeout. For sound combinations of dynamic analysis and Abstract Interpretation there is often no counterargument, except implementation work, to combine both as the precision and analysis speed is often improved. Frameworks such as *Concerto* can help to reduce the combination effort but for easy integration a standardised interface to communicate between different analysers might be helpful.

5 Further work

Shiyi Wei and Barbara G. Ryder and created a framework to combine static and dynamic taint analysis into *blended taint analysis* [20] increasing the detection rate over purely static analysis. Patrice Godefroid, Nils Klarlund and Koushik Sen created DART [5] a framework to automatically generate program tests, combining *random interpretation* [6] with several static and dynamic analysing methods. Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Möller present a technique to combine a non-relational static analysis soundly with a value refinement mechanism to increase precision on demand at precision critical locations [17]. Cyrille Artho and Armin Biere crated JNuke [2] which reduces the implementation cost of a combined interpreter by sharing the used algorithm between the concrete and abstract domains.

References

1 Roberto Amadini, Alexander Jordan, Graeme Gange, François Gauthier, Peter Schachte, Harald Søndergaard, Peter J. Stuckey, and Chenyi Zhang. Combining String Abstract Domains for JavaScript Analysis: An Evaluation. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 41–57, Berlin, Heidelberg, 2017. Springer. `doi:10.1007/978-3-662-54577-5_3`.

2 Cyrille Artho and Armin Biere. Combined Static and Dynamic Analysis. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 131:3–14, May 2005. `doi:10.1016/j.entcs.2005.01.018`.

3 Thoms Ball. The concept of dynamic analysis. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-7, pages 216–234, Berlin, Heidelberg, October 1999. Springer-Verlag.

4 Patrick Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, June 1996. `doi:10.1145/234528.234740`.

5 Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. *ACM SIGPLAN Notices*, 40(6):213–223, June 2005. `doi:10.1145/1064978.1065036`.

6 Sumit Gulwani and George C. Necula. Precise interprocedural analysis using random interpretation. *ACM SIGPLAN Notices*, 40(1):324–337, January 2005. `doi:10.1145/1047659.1040332`.

7 Stefan Heule, Manu Sridharan, and Satish Chandra. Mimic: Computing models for opaque code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 710–720, New York, NY, USA, August 2015. Association for Computing Machinery. `doi:10.1145/2786805.2786875`.

8 Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the eval that men do. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 34–44, New York, NY, USA, July 2012. Association for Computing Machinery. `doi:10.1145/2338965.2336758`.

9 Changhee Park and Sukyoung Ryu. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 735–756, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.ECOOP.2015.735`.

10 Jihyeok Park, Yeonhee Ryou, Joonyoung Park, and Sukyoung Ryu. Analysis of JavaScript Web Applications Using SAFE 2.0. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 59–62, May 2017. doi:10.1109/ICSE-C.2017.4.

11 Joonyoung Park, Alexander Jordan, and Sukyoung Ryu. Automatic Modeling of Opaque Code for JavaScript Static Analysis. In Reiner Hähnle and Wil van der Aalst, editors, *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 43–60, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-16722-6_3.

12 Joonyoung Park, Jihyeok Park, Dongjun Youn, and Sukyoung Ryu. Accelerating JavaScript static analysis via dynamic shortcuts. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1129–1140, Athens Greece, August 2021. ACM. doi:10.1145/3468264.3468556.

13 Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Dynamic determinacy analysis. *ACM SIGPLAN Notices*, 48(6):165–174, June 2013. doi:10.1145/2499370.2462168.

14 Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 488–498, New York, NY, USA, August 2013. Association for Computing Machinery. doi:10.1145/2491411.2491447.

15 Nastaran Shafiei and Franck van Breugel. Automatic handling of native methods in Java PathFinder. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, SPIN 2014, pages 97–100, New York, NY, USA, July 2014. Association for Computing Machinery. doi:10.1145/2632362.2632363.

16 Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation Tracking for Points-To Analysis of JavaScript. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, Lecture Notes in Computer Science, pages 435–458, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-31057-7_20.

17 Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. Static analysis with demand-driven value refinement. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):140:1–140:29, October 2019. doi:10.1145/3360566.

18 Bae SungGyeong, Cho Hyunghun, Lim Inho, and Ryu Sukyoung. SAFEWAPI: Web API misuse detector for web applications. https://dlnext.acm.org/doi/10.1145/2635868.2635916.

19 John Toman and Dan Grossman. Concerto: A framework for combined concrete and abstract interpretation. *Proceedings of the ACM on Programming Languages*, 3(POPL):43:1–43:29, January 2019. doi:10.1145/3290356.

20 Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for JavaScript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 336–346, New York, NY, USA, July 2013. Association for Computing Machinery. doi:10.1145/2483760.2483788.

21 Juan Zhai, Jianjun Huang, Shiqing Ma, Xiangyu Zhang, Lin Tan, Jianhua Zhao, and Feng Qin. Automatic model generation from documentation for Java API functions. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 380–391, New York, NY, USA, May 2016. Association for Computing Machinery. doi:10.1145/2884781.2884881.