A Brief Survey of Specification Inference in Static Program Analysis

Chuansen Chai[1, a], Xuexiong Yan[1, a], Qingxian Wang[1, a], Shukai Liu[1, a], Yajing Sun[1, a] and Shuai Yi[1, a]

[1] State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, 450001, China

[a]xdccs@foxmail.com

Keywords: Survey; Specification Inference; Static Analysis

Abstract. Nowadays, many applications are too big to keep their own security. Static analysis is a technique which can be used for ensure the safety of large programs. However, frequent calls to libraries and frameworks bring a lot of difficulties to the analysis. Researchers propose the specification inference to solve the problem. We survey the recent advances in specification inference techniques and divide them into three groups by the methods they adopt which are domain specific language (DSL), data mining and abductive inference, respectively. Then we take a detailed look at the representative results in each category. It is our hope that we can see more creative achievements in this field by doing this work.

Introduction

Static program analysis is a kind of analysis for computer software that can be performed without actually executing programs and it is proved to be a useful technique for program verification and security. Previous work has shown that the static analysis can detect errors and vulnerabilities in programs efficiently. The precision and scalability of this technique rely on the ability that all source codes can be analyzed. Nevertheless, many of today's prevalent software have a huge number of codes interacting with libraries which may use low-level language or dynamic features such as reflection. The hard-to-analyze codes may lead the existing work into trouble. Many existing static analysis tools handle the missing parts by using either pessimistic but imprecise or optimistic but unsound assumptions. Some static analysis tools ignore the library calls and lead to false negatives. We call these ways as ad hoc methods which overlook the issues rather than solve them.

The initial approach to handle the problem is by writing specifications by human auditors. However, this suffers from scalability and efficiency, as mentioned in [1], they spent over one year analyzing Android malware, but only 179 library classes' specifications were written. This is unacceptable in real-world working environment.

At present, there hasn't been a specific reported survey about the techniques for specification inference. In this paper, we try to classify the existing research results in this field into three categories by the methods they adopt, respectively specification language [4,5,6,7], Date mining [8,9,10,11,12] and adductive inference [1,2,3]. Due to the limit of space, we only give detailed discussion on the representative of the results in recent years for each category.

Domain Specific Language

An immediate method to solve the problem is by assisting developers describing the behavior of libraries [4,5,6,7]. Some researchers design domain specific languages which help the developer to describe the behavior of libraries. This way wants to model the specific semantics which is hard for normal static analysis to handle and transform those parts into easy-to-analysis codes. Sridharan's [4] approach is representative among similar methods. It presents a system for taint analysis of framework-based web applications called F4F (framework for framework). The framework focuses on the reflection feature in Java which makes trouble for analysis.

Many of modern frameworks often use reflection to provide higher-level abstraction for changing common tasks such as configuration files. Unfortunately, the information that configuration files provide can be precisely handled via normal code analysis. F4F's practice is to automatically generate specifications for reflections used in frameworks by defining a DSL called WAFL (for Web Application Framework Language). This language can be easily integrated into the existing taint analysis system and its grammar is simple enough to guarantee the developers learning quickly as shown in Fig. 1.

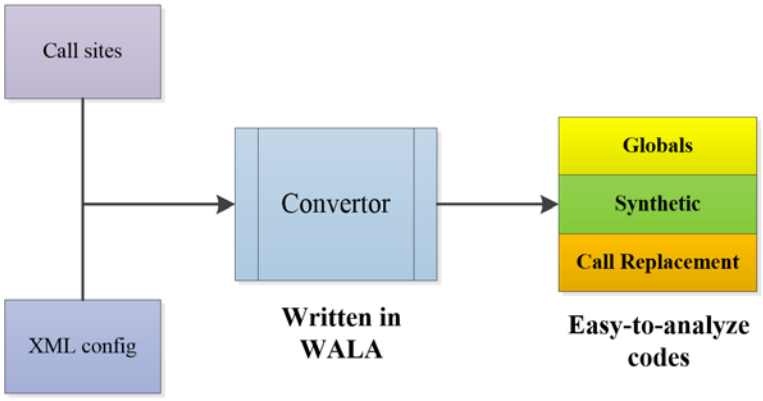

Figure 1. The basic process of F4F

WAFL consists of three key elements, respectively is global declarations, synthetic methods, and the call replacements. The authors build these three elements based upon analyzing the semantics of reflections.

The global declarations is used for model the disparate parts of the applications. Some attributes such as request and session in Java. Web applications exist in different scope. Global declarations significantly reduce the efforts that subsequent taint analysis.

Synthetic methods are designed for modeling how the library invokes codes in application. They present the original workflow of framework by code snippets which the taint analysis can handle. In frameworks such as Struts, part of the functionality is implements by using reflection to instantiate the classes named in the configuration file. This makes that the static analysis tools cannot get information about class presented in the configuration files. Moreover, even the control-flow is hidden. With the Synthetic methods, the pitfalls caused by reflection are deal can be avoided.

Call Replacements change the original call sites of library method with new codes.

This approach works well in the real-world applications but it suffers from the following disadvantages:

(1) It doesn't realize complete automation. Developers who need to write specifications have to master the DSL.

(2) This approach is for carrying out manipulation to limited kinds of problems such as reflection. When faced with more complicated situations, the language will expose inherent weaknesses.

(3) The validity of this way is based on the understanding of semantics of concrete codes. It needs all the library's codes being scanned.

Data Mining

Some researchers use data mining techniques to infer specifications. Data is a broad concept. Different researchers have various understandings and opinions about the definition of specification. For example, Zhou et al. [9] and livshits et al.[10] combines data mining and pointer analysis to infer the patterns about action pairs. Jagannathan et al. [11] focus on the inference of method preconditions in software systems while Whaley et al. [12] consider the interface specification of method calls.

As shown above, due to the inconsistent perspective, different the specific techniques are used in

the inference process. In this paper, we mainly introduce Merlin [8] and hope future researchers can get some insights from it.

Merlin mostly deals with the inference problem existing in explicit information flow by combining the probabilistic inference and the propagation graph. The propagation graph is a directed graph where the nodes represent for methods in program and an edge indicates that there is a data flow from the starting method to the end method. It is the basis for a set of probabilistic constraints which provide information for inference.

The goal of specification inference is to divide the nodes in the propagation graph into four types: sources, sinks, sanitizers and regular nodes. The type distribution of nodes forms the specification about the library. The probabilistic constraints are based on two assumptions:

(1) Most paths are secure

(2) The number of sanitizer nodes is small relative to the regular node.

The two assumptions mentioned above is coincident with the real situations according to the statistics. The authors add four auxiliary constraints for more precise. Then a set of probabilistic constraints can be generated from the propagation graph and the conjunction of constraints of all paths is the key indicator of the odds of the specific specification.

The joint probability distribution $p_i(x_i)$ over boolean $x_1, \ldots, x_N$ is defined as:

$$p_i(x_i) = \sum_{x_1} \cdots \sum_{x_{i-1}} \sum_{x_{i+1}} \cdots \sum_{x_N} p(x_1, \ldots, x_N) \tag{1}$$

However, the number of path is an exponential number, the formula can't be computed in practice. Therefore, this paper uses the factor graph [13] to compute the marginal probabilities. The Framework of Merlin is shown in Fig. 2.

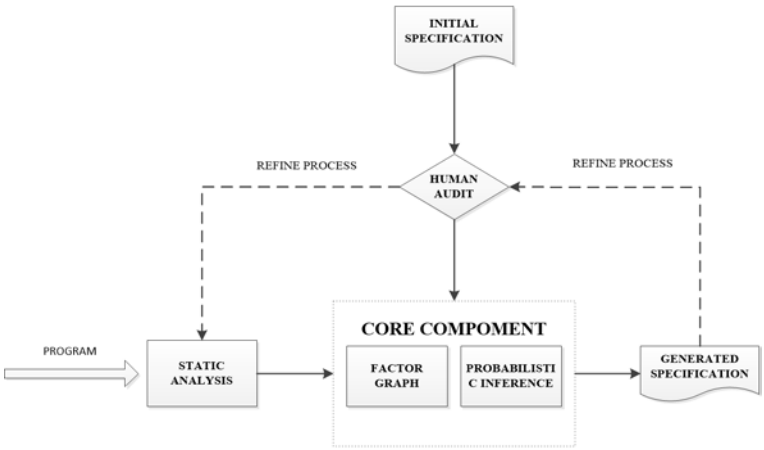

Figure 2. MERLIN system architecture

The core component consists of the following stages:

1) Construct a factor graph on the basis of the propagation graph.

2) Derive the most likely specification by performing probabilistic inference on the factor graph.

The bright side of this method is that the scalability is handled well since it approximates the exponential number of probabilistic constraints to a cubic number of triple constraints.

But the selection of assumption is skillful and needs trying over and over again and it faces severe tests when applying to more complex situations.

Abduction Inference

Abductive reasoning (also called abductive inference) is a form of logical inference which goes from an observation to a theory which accounts for the observation, ideally seeking to find the simplest and most likely explanation [14], adductive reasoning can be flexible to integrate to the static analysis. Many researchers have already tried this way. Some researchers apply this theory to

program verification problem, [15] address verifying pointer safety and use a rule-based approach in separation logic [3]. In this paper, we mainly introduce the work from Stanford University. It has significant practical use.

Bastani et al. [1] focus on the specification problem about taint analysis in Android applications. The huge Android library stack which contains a mix of Java and C++ code make it difficult to perform sound and precise taint analysis to find bugs. They use abductive inference theory to help perform a sound and precise CFL reachability analysis along with missing specifications. The purpose is to inference potential aliases and taint flows in the library. Their work can be simply divided into three steps:

(1) They make worst-case assumptions about the missing specification, and find potential flows.

(2) Based on the assumptions: potential flows that require fewer assumptions are more likely to be real flows that require more assumptions [1]. For each potential flow, their static analysis tool records the sufficient assumptions (minimal set of sufficient assumptions) that prove the flow is true.

(3) These sufficient assumptions is the inferred specifications.

The key of their method's success is the reasonability of assumption. This way does not need to know the concrete implementations of libraries. We look forward to subsequent achievements.

Conclusion

In this paper, we have surveyed some of the state-of-the-art research about specification inference. It is a pleasure to see the specification inference technique is using in more and more real-world applications. By summing up, we can find that the real difficulty of inference is gathering enough information about the behavior of program. The three methods differ depend on the way they extract information. Some need the assistant of human auditors while others rely on the assumptions. Meanwhile, we believe that there are still lots of meaningful works that can be done to further improve the precision. Nearly all assumption made in the technique is from the researchers' observing. They summarize the patterns and apply to the inference technique. Another interesting idea is that tech convergence may improve the effect of inference.

We strongly believe that the scientists grow more modest about what we are close to knowing and achieving as the research moves along.

References

[1] O Bastani, S Anand, A Aiken. Specification inference using context-free language reachability[C]//ACM SIGPLAN Notices. ACM, 2015, 50(1): 553-566.

[2] K Ali, O Lhoták. Averroes: Whole-program analysis without the whole program[M]//ECOOP 2013–Object-Oriented Programming. Springer Berlin Heidelberg, 2013: 378-400.

[3] H Zhu, T Dillig, I Dillig. Automated inference of library specifications for source-sink property verification[M]//Programming Languages and Systems. Springer International Publishing, 2013: 290-306.

[4] M Sridharan, S Artzi, M Pistoia, et al. F4F: taint analysis of framework-based web applications[J]. ACM SIGPLAN Notices, 2011, 46(10): 1053-1068.

[5] Typestate-oriented programming:The case for analysis preserving language transformation

[6] C Jaspan, J Aldrich. Checking framework interactions with relationships[M]. Springer Berlin Heidelberg, 2009.

[7] X Zhang, L Koved, M Pistoia, et al. The case for analysis preserving language transformation[C]//Proceedings of the 2006 international symposium on Software testing and analysis. ACM: 191-202(2006).

[8] B Livshits, A Nori V, S K Rajamani, et al. Merlin: specification inference for explicit information flow problems[C]//ACM Sigplan Notices. ACM, 44(6): 75-86(2009).

[9] N E Beckman, A V Nori. Probabilistic, modular and scalable inference of typestate specifications[C]//ACM SIGPLAN Notices. ACM, 46(6): 211-221(2011).

[10] T Kremenek, P Twohey, G Back, et al. From uncertainty to belief: Inferring the specification within[C]//Proceedings of the 7th symposium on Operating systems design and implementation. USENIX Association: 161-176(2006).

[11] S Shoham, E Yahav, S J Fink, et al. Static specification mining using automata-based abstractions[J]. Software Engineering, IEEE Transactions on, 34(5): 651-666(2008).

[12] J W Nimmer, M D Ernst. Automatic generation of program specifications[J]. ACM SIGSOFT Software Engineering Notes, 27(4): 229-239(2002).

[13] J S Yedidia, W T Freeman, Y Weiss. Understanding belief propagation and its generalizations[J]. Exploring artificial intelligence in the new millennium, 8: 236-239(2003).

[14] Information on https://en.wikipedia.org/wiki/Abductive_reasoning

[15] C Luo, F Craciun, S Qin, et al. Verifying pointer safety for programs with unknown calls[J]. Journal of Symbolic Computation, 45(11): 1163-1183(2010).