A Survey on Static Program Analysis

Xin Liu, Yihong Yan, Hong Zheng
{liux4,zhengho,yih}@onid.oregonstate.edu

Abstract

Static program analysis is an important and effective way to improve the quality of software systems. We conducted a lightweight survey by reviewing 16 papers related to static analysis published in the International Conference on Software Engineering (ICSE) between 2010 to 2015. Through this survey, we would like to get basic knowledges of the development of static analysis techniques in both academia and industry.

1 Introduction

Static program analysis, also called static analysis, is the analysis of computer software that is performed without actually executing programs. In most cases the analysis is performed on some version of the source code, and in the other cases, some form of the object code. Static analysis techniques range from the most mundane to the more complex, semantics-based analysis techniques.

In this survey, we review the papers related to static analysis published in ICSE between 2010 to 2015. The purpose of the survey is to investigate the research directions and the current achievements in the area of static analysis research. Hence, we constructed a Static Analysis publication repository [1] for this survey: we first searched two major online repositories, IEEE XPlore and ACM Portal, collecting 43 papers which have "static analysis", "static program analysis", "symbolic analysis", "program analysis", and "static + analysis" keywords in their title and abstract, and are published in the years between 2010 and

2015. Then, we filter out those unrelated to the static analysis techniques. In the end, we have 16 papers to review. Our discussion in the following sections is based on the analysis of these 16 papers, which is divided into three smaller categories according to the different applications of static analysis techniques: software testing, software maintenance and development, and software security.

2 The Application of Static Analysis

2.1 Using Static Analysis in Software Testing

In this section, we review 7 papers that made different innovations in the application of static analysis techniques in testing software.

In [2], Pradel *et al.* presented an automatic protocol conformance checker, which combines a dynamic miner of multi-object protocols[3][4] and a static checker of API constraints[5]. Compared with the existing approaches for API protocol testing, the new checker is built on two independent analyses. Such a strategy can guarantee the best results returned at different stages. In order to bridge these two analyzers, they designed a translation mechanism to connect two different formalisms used by the two analyzers respectively. Their approach finds bugs without any a priori known specifications and without human refinement or selection of the mined protocols. However, the cost of the automation is the introduction of false positives. To deal with the issue and improve the precision of results, they designed three pruning techniques: pruning by *Number of Confirmation*, by *Protocol Error Rate*, and by *Method Error Rate*[2].

Zhang presented a technique to combine results from a dynamic analysis and a static analysis to improve random test generation[6]. The combination can be summarized as three steps: dynamic inference, static analysis, and guided random test generation. The hybrid approach first takes a sample program as input, infers an call sequence model[7] by dynamic analysis, and enhances it with *direct state transition dependence constraint* and *abstract object profile constraint*. The enhanced call sequence model is used to capture the possible legal method call sequences and argument values. Since the model may miss covering some interesting methods or certain method call invocation orders, a static method dependence analysis is designed to enrich the model, based on the hypothesis: two methods are related if the fields they read or write overlap. Finally, both the dynamically-inferred model and the statically-identified dependence information guide a random test generator to create legal and behaviorally-diverse tests. Compared with Randoop[8] (a pure random approach), Palulu[7] (a dynamic-random approach), RecGen[9] (a static-random approach), the author's hybrid approach achieved higher line coverage on average.

Similarly, Ge *et al.* also proposed a hybrid defect-detection approach[10]. Their approach uses a novel way to combine both static verification and dynamic test generation so as to overcome the false positives and the path space explosion issues. Based on several existing tools, they constructed a tool chain called DyTa to implement the approach. According to their design, in the static phase, *DyTa*, first, applies the static checker of Code Contracts tool[11] to the program, and identifies a set of potential defects and collects their locations and descriptions. In order to make the program under analysis amenable for Dynamic Symbolic Execution (DSE)[12] -based test generation tool *Pex*[13], then *DyTa* performs two types of instrumentations to the program under analysis. In the dynamic phase, *DyTa* applies *Pex* to primarily explore those paths related to the potential defects detected by the static checker. Besides, different from other approaches, such as Check 'n' Crash [14], *DyTa* considers both locations of potential defects and error conditions so that the potential defect could be covered in the first place.

Although symbolic analysis has great potential for bug finding, such as better coverage than dynamic testing and higher precision than traditional static analysis, determining symbolic values for program variables especially related to loops and library calls is challenging, as the computation and data related to loops can have statically unknown bounds, and the library sources are typically not available at compile time. Le discussed the issues in details, and also proposed a novel concept of segmented symbolic analysis and its implementation, *Helium*, to deal with the issues in [15]. The core idea is to introduce dynamic analysis to supply information that a pure static symbolic analyzer is slow or unable to produce. When a library call or a loop is encountered, the symbolic analysis suspends the propagation of the current symbolic condition and continues analyzing other paths. Meanwhile, *Helium* performs a structural and def-use analysis of the code and identifies the code segment for constructing a unit test. The dynamic analysis automatically synthesizes the unit test and generates the test inputs. After running the tests, a regression based inference is performed on test inputs and outputs to derive transfer functions for the unknown code segment. Notified with the newly discovered information, the symbolic analysis then updates the blocked symbolic conditions and resumes the analysis of this path. Different from other hybrid approaches, static and dynamic analyses are *concurrently* applied on different parts of the program to ensure that the capabilities of the analyses match the code characteristics, while the interaction between static and dynamic analyses are accomplished via a query based protocol.

Due to the difficulty of obtaining project-specific rules and manually writing checkers for them, commercial static program analysis tools are not fully put into use by software development organizations. To eliminate the obstacles and enhance the defect detection capability of static analysis, Sun *et al.* presented a hybrid approach in their research[16]. They introduced dependence-based pattern mining techniques[17] to capture frequent code patterns, which are in the form of generic System Dependence Graph (SDG)[18][19] subgraphs. Then, a rule extractor analyzes the SDG structure of a pattern and the abstract syntax trees (AST) of its statement instances, and generates the checking rules for the specific project. Finally, the mined code patterns are transformed into custom checkers that a commercial static analysis tool, such as Klocwork, can run against a code base to reveal defects.

[20] and [21] discuss the detection of concurrent-related errors, such as atomicity violations and data races. In [20], Zheng and Zhang proposed a context- and path-sensitive inter-procedural static analysis that detects atomicity violations in web applications regarding external resources in PHP code. The first novelty of the technique is that the authors developed a resource identity analysis, which

encodes the semantics of external operations into bit-vector logic constraints. These constraints, together with those generated from the regular PHP statements, are resolved by a SMT solver to determine if two given operations are accessing the same external resource. The second novelty is to define atomic regions by considering pair-wise atomicity of external operations, and by leveraging the resource identity analysis and program dependences. Marino *et al.* proposed an algorithm in their recent research[21] to extend an existing type-based static analysis so as to enhance the capability of deadlocks detection in programs written in a domain-specific language called *AJ*. The algorithm computes a partial order on atomic sets which is consistent with lock acquisition order. If such an order can be found, a program is deadlock-free. For programs that use recursive data structures, the approach is soundly extended to take into account a programmer-specified ordering between different instances of an atomic set.

2.2 Using Static Analysis in Software Development and Maintenance

Besides the wide application in the software testing phase of the entire software development life cycle, static analysis techniques also play significant role in other phases, such as development and maintenance phases.

Due to the dynamic nature of JavaScript, in IDEs for Javascript, code navigation and completion use heuristics that sometimes fail unexpectedly, while refactoring and analysis is all but unsupported. Different from statically typed programming languages, such as Java or C#, JavaScript is dynamically typed and uses prototype-based inheritance, therefore neither class hierarchy analysis nor its more refined variants that keep track of class instantiations [22] [23] are directly applicable. In [24], Feldthaus *et al.* argued that how to construct call graphs efficiently is the key to the problem. The existing flow analyses, which are either not fast enough for interactive use [25] [26] or only suitable for small framework-based applications [27], do not fit into the practical uses. Therefore, they proposed a fast field-based [28] [29] flow analysis. The approach does not distinguish two functions that are assigned to properties of the same name. Besides, it only tracks function object and does not reason about any non-functional values, and meanwhile, it ignores dynamic property access (i.e., property reads and writes using JavaScript's bracket syntax.) Like any flow analysis, this approach also faces a chicken-and-egg problem: to propagate (abstract) argument and return values between caller and callee, a call graph is required, yet a call graph is the construction target. To tackle this problem, the authors attempted two variants of the flow-based analysis: the standard optimistic analysis [30] (OA) and the pessimistic analysis (PA). OA starts out with an empty call graph, which is gradually extended as new flows are discovered until a fix point is reached, while PA does not reason about interprocedural flow at all and simply give up on call sites whose call target may depend on such flow, except in cases where the callee can be determined purely locally. According to their evaluations, PA may be preferable in many cases.

In [31], Balachandran proposed a tool, Review Bot, to integrate the static analysis into the code review phase of software development. The tool attempts to solve two issues: the automations of both checking coding standard violations and comment defect patterns and generating reviewer recommendations. To answer the first question, Review Bot is built as an extension to Review Board, an open-source code review tool. Review Bot uses three static analysis tools, which are Checkstyle, PMD, and FindBugs, to automate the checks for coding standard violations and common defect patterns, and publish code reviews using the output from these tools. To solve the second problem, the authors designed a reviewer recommendation algorithm, which is based on line change history of source code. The line change history of a line in a filediff contained in a review request is the list of review requests which affected that line in the past.

[32], [33], and [34] discuss the use of static analysis in detecting potential software system configuration errors. Usually, a configuration error is silent, sometimes non-crashing, which does not exhibit a crashing point, dump a stack trace, output an error message, or indicate suspicious program variables that may have incorrect values. Lacking such information makes many existing techniques such

as dynamic slicing [35], dynamic information flow tracking [36], and failure trace analysis inapplicable.

In [32] and [33], Zhang and Ernst discussed a case of configuration errors, in which the bug reporter had already minimized the bug report: if any part of the configuration or input is removed, SUT either crashes or no longer exhibits this error. Although the root cause of the bug is that the user failed to set one configuration option, no previous configuration error diagnosis technique [37] [36] [38] [39] [40] [41] [42] can be applied directly in this case. For that, they designed a tool called *ConfDiagnoser*, which is used by system administrators and end-users when they encounter an error that they do not know how to fix. The process of linking the undesired behavior to specific root cause can be summarized as three steps: (1) *ConfDiagnoser* uses thin slicing [20] to statically identify the predicates each configuration option affects in the source code; (2) *ConfDiagnoser* selectively instruments the program-to-diagnose so that it records the run-time behaviors of affected predicates in an execution profile; (3) *ConfDiagnoser* identifies the predicates whose dynamic behaviors deviate the most between correct and undesired executions, and then identifies its affecting configuration options as the likely root causes. Finally, it outputs a ranked list of suspicious configuration options and explanations. Compared to previous approaches [36] [38] [39] [40] [43] [35], *ConfDiagnoser* is fully automated, can diagnose both non-crashing and crashing errors, and does not require OS-level support.

Rabkin and Katz also discussed the issues happened in the software system configuration management in their research [34]. They observed that the key-value style of configuration is widely adopted in software systems, as it is convenient for developers or users to add new options incrementally. However, such a convenience also introduces some kinds of errors. For example, user-written configuration files can assign values for options that a program never reads; documentation falls out of step with an evolving program, and etc.. Therefore, Rabkin and Katz proposed an approach based on static analysis. They classifies the program configuration options into 17 types. Then, [34] Kim *et al.* discussed semantic clone detections in their research [44]. According to their investigation, most clone detectors [45] [46] [47] [48] [49] are based on textual similarity, which are not effective to detect semantic clones that are functionally similar but syntactically different, while existing approaches to detect semantic clones have limitations. For example, program dependence graphs-based approaches can be affected by syntactic changes so as to miss some semantic clones; the clone detectability of random testing-based approaches may depend on the limited test coverage. Hence, Kim *et al.* introduced semantic-based static analysis into the clone detection. They built a semnatic-based static analyzer on top of commercialized analyzer SPARROW, which can summarize each procedure after analyzing the procedure based on the abstract interpretation framework [50], and these procedural summaries have been carefully tuned to capture all memory-related behaviors in real-world C programs [51]. In order to support path-sensitive analysis, they further extended SPARROW to be path-sensitive like [52] by adding guards and guarded values to the abstract domain. By comparing programs' abstract memory states, they achieve the goal of the semantic clone detection.

Static program slicing is an attractive option for performing routine change impact analysis of newly committed changesets. However, static program slicing suffers from performance issues when analyzing large systems. Besides, program slicing can have accuracy issues as well [53]. In their experiment [54], Acharya and Robinson identified and outlined the unique problems encountered in using static program slicing directly for change impact analysis of large industrial software: (1) the build time and the total slice time are excessive with HIGH setting being the most precise. (2) the safety with HIGH setting might come at the expense of the impact set being over conservative. (3) it's not clear how much of the impact set's accuracy is sacrificed for faster builds with the LOW setting. (4) The impact set can be very large so that it cannot be easily explored by the developers for routinely inspecting the impact of the committed changesets. The terms of HIGH and LOW denote two sets of CodeSurfer, which is a commercial static analyzer. In order to reduce the time and improve the accuracy, Acharya and Robinson proposed a framework based on static program slicing, *Imp*. The framework is implemented on top of CodeSurfer. The core ideas behind *Imp* are (1) perform HIGH setting analysis infrequently; (2) obtain clues in the program for possible over conservativeness by computing the common global data structures among large impact sets and the high impact functions; (3) perform LOW setting analysis frequently; (4) use the clues obtained from the infrequent HIGH setting builds to guide the LOW setting builds. According to their evaluations, *Imp* is capable of perform change impact analysis on systems with over a million lines of code.

2.3 Using Static Analysis in Software Security

Security of software systems touches on a vast and complex array of issues, making it difficult and expensive to implement a comprehensive security solution. In this section, we discuss three novel approaches to detect vulnerabilities.

Remote code execution (RCE) attacks are one of the most prominent security threats for web applications. However, the existing detection techniques failed or have a limited capability to detect and confirm RCE attacks. Most of the existing static analysis techniques [55] [56] [57] [58] [59] [20] [60] cannot cohesively reason about the string and non-string parts of an application and many lack path sensitivity, whereas RCE attacks require satisfying intriguing path conditions, involving both strings and non-strings. Even though some researchers model both strings and non-strings in dynamic symbolic execution of web apps [61], these new techniques simply focus on modeling the executed path, whereas RCE attack detection requires modeling all program paths. To solve the issue, Zheng and Zhang proposed a path- and context-sensitive inter-procedural static analysis in their research [62], which can detect RCE vulnerabilities in PHP code. Their system makes use of LLVM, the PHP compiler (*phc*) [63], the STP solver and the HAMPI string solver [55]. At the abstraction phase, the analysis first creates two abstractions of each PHP script: one for the string related behavior and the other for the non-string related behavior. The non-string abstraction includes additional taint semantics to reason about the input correlation for each variable. At the encoding and solving phase, two abstractions are encoded separately. An algorithm is designed for querying the STP solver and the HAMPI string solver [55] iteratively and alternatively to derive a consistent path-sensitive solution for both sets of constraints. For a potentially vulnerable file write, the system queries the string constraints if the file name ends with the PHP extension, and queries the non-string constraints to determine if the written content is tainted and the file write is reachable. The solution has to consistently satisfy all these queries.

In the previous work [64] [65], the authors mined static code patterns that implement input validation and input sanitization methods to build vulnerability predictors based on supervised learning. However, the earlier work suffers from two major drawbacks: (1) the predictive capability of proposed static attributes is dependent on the precision of the classification of the input validation and sanitization code patterns; (2) the effectiveness of supervised learning-based approach is dependent on the availability of sufficient training data labeled with manually checked security vulnerabilities. In order to address the limitations, the authors proposed attributes [66], based on hybrid static and dynamic code analysis, which characterize input validation and sanitization code patterns for predicting SQL injection and XSS vulnerabilities. static analysis in their research is used to classify the nodes on the data dependence graph (DDG) of a sink that represents a program statement that interacts with database or web client. Then, their approach captures the classifications in a set of attributes on which vulnerability predictors are to be built.

Moller and Schwarz addressed their observation on web applications and argued that vulnerability involving client-state manipulation is strongly correlated to information flow from hidden fields or other kinds of client state to operations involving the shared application state on the server [67]. Based on the observation, they prototyped a static analysis tool, called *WARlord*, for mainstream web application frameworks to detect such vulnerabilities. The analysis has three major components: the first component infers the dataflow between the individual servlets and pages that constitute the application, in order to identify the client-state parameters; the second component finds out which objects represent shared application state by analyzing the program codes; the third component performs an information flow analysis to identify the possible flow of user controllable input from client-state parameters to shared application state objects. Compared to the manual inspection in their evaluation, this approach has great precision, and it can analyze between 10 and 200 pages per minutes.

3 Conclusion and Future Work

In this paper, we reviewed 16 conference papers related to static program analysis based on three application categories: software testing, software maintenance and development, and software security. In the software testing community, the general research trend is to combine other assistant methods with pure static analysis, because pure static analysis inevitably produces false positives. Besides, the static analysis techniques are also adopted by researchers to detect resource contention problems in concurrent computing. In the software maintenance and development community, researchers prototyped different toolchain or extended existing automation tools to maximize the automation in coding/reviewing, managing configuration options, and monitoring and analyzing code changes. In the software security community, static analysis is used to obtain the overall path information of program under test.

Due to the time limitation, our survey simply covers 16 papers from a single international conference in only recent 5 years. Therefore, the help of the survey is very limited for a future research. However, it is a great start. We plan to extend our current survey to cover more research works in a wider timeline so that the survey can help pinpoint those valuable research points.

References

[1] X. Liu, H. Zheng, and H. Yi, "Static analysis paper repository," *https://github.com/apepkuss/StaticAnalysisPaperRepository.*

[2] M. Pradel, C. Jaspan, J. Aldrich, and T. Gross, "Statically checking api protocol conformance with mined multi-object specifications," in *Software*

Engineering (ICSE), 2012 34th International Conference on, pp. 925–935, June 2012.

[3] M. Pradel, P. Bichsel, and T. Gross, "A framework for the evaluation of specification miners based on finite state machines," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pp. 1–10, Sept 2010.

[4] M. Pradel and T. Gross, "Automatic generation of object usage specifications from large method traces," in *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, pp. 371–382, Nov 2009.

[5] C. Jaspan, "Checking framework interactions with relationships," in *Companion to the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA Companion '08, (New York, NY, USA), pp. 901–902, ACM, 2008.

[6] S. Zhang, "Palus: a hybrid automated test generation tool for java," in *Software Engineering (ICSE), 2011 33rd International Conference on*, pp. 1182–1184, May 2011.

[7] S. A. M. D. E. Adam and K. C. P. J. H. Perkins, "Finding the needles in the haystack: Generating legal test inputs for object-oriented programs," *and Object-Oriented Systems*, p. 27, 2006.

[8] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball, "Feedback-directed random test generation," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pp. 75–84, May 2007.

[9] W. Zheng, Q. Zhang, M. Lyu, and T. Xie, "Random unit-test generation with mut-aware sequence recommendation," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, (New York, NY, USA), pp. 293–296, ACM, 2010.

[10] X. Ge, K. Taneja, T. Xie, and N. Tillmann, "Dyta: dynamic symbolic execution guided with static verification results," in *Software Engineering (ICSE), 2011 33rd International Conference on*, pp. 992–994, May 2011.

[11] F. Logozzo, "Practical verification for the working programmer with codecontracts and abstract interpretation," in *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, (Berlin, Heidelberg), pp. 19–22, Springer-Verlag, 2011.

[12] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," *SIGPLAN Not.*, vol. 40, pp. 213–223, June 2005.

[13] N. Tillmann and J. De Halleux, "Pex: White box test generation for .net," in *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, (Berlin, Heidelberg), pp. 134–153, Springer-Verlag, 2008.

[14] C. Csallner and Y. Smaragdakis, "Check 'n' crash: Combining static checking and testing," in *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, (New York, NY, USA), pp. 422–431, ACM, 2005.

[15] W. Le, "Segmented symbolic analysis," in *Software Engineering (ICSE), 2013 35th International Conference on*, pp. 212–221, May 2013.

[16] B. Sun, G. Shu, A. Podgurski, and B. Robinson, "Extending static analysis by mining project-specific rules," in *Software Engineering (ICSE), 2012 34th International Conference on*, pp. 1054–1063, June 2012.

[17] R.-Y. Chang, A. Podgurski, and J. Yang, "Discovering neglected conditions in software by mining dependence graphs," *Software Engineering, IEEE Transactions on*, vol. 34, pp. 579–596, Sept 2008.

[18] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, pp. 319–349, July 1987.

[19] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *SIGPLAN Not.*, vol. 39, pp. 229–243, Apr. 2004.

[20] Y. Zheng and X. Zhang, "Static detection of resource contention problems in server-side scripts," in *Software Engineering (ICSE), 2012 34th International Conference on*, pp. 584–594, June 2012.

[21] D. Marino, C. Hammer, J. Dolby, M. Vaziri, F. Tip, and J. Vitek, "Detecting deadlock in programs with data-centric synchronization," in *Software Engineering (ICSE), 2013 35th International Conference on*, pp. 322–331, May 2013.

[22] D. F. Bacon and P. F. Sweeney, "Fast static analysis of c++ virtual function calls," in *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, (New York, NY, USA), pp. 324–341, ACM, 1996.

[23] F. Tip and J. Palsberg, "Scalable propagation-based call graph construction algorithms," in *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, (New York, NY, USA), pp. 281–293, ACM, 2000.

[24] A. Feldthaus, M. Schafer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for javascript ide services," in *Software Engineering (ICSE), 2013 35th International Conference on*, pp. 752–761, May 2013.

[25] L. O. Andersen, *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[26] O. Shivers, "Control flow analysis in scheme," *SIGPLAN Not.*, vol. 23, pp. 164–174, June 1988.

[27] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip, "Correlation tracking for points-to analysis of javascript," in *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, (Berlin, Heidelberg), pp. 435–458, Springer-Verlag, 2012.

[28] N. Heintze and O. Tardieu, "Ultra-fast aliasing analysis using cla: A million lines of c code in a second," *SIGPLAN Not.*, vol. 36, pp. 254–263, May 2001.

[29] O. Lhoták and L. Hendren, "Scaling java points-to analysis using spark," in *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, (Berlin, Heidelberg), pp. 153–169, Springer-Verlag, 2003.

[30] D. Grove and C. Chambers, "A framework for call graph construction algorithms," *ACM Trans. Program. Lang. Syst.*, vol. 23, pp. 685–746, Nov. 2001.

[31] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *Software Engineering (ICSE), 2013 35th International Conference on*, pp. 931–940, May 2013.

[32] S. Zhang and M. Ernst, "Automated diagnosis of software configuration errors," in *Software Engineering (ICSE), 2013 35th International Conference on*, pp. 312–321, May 2013.

[33] S. Zhang, "Confdiagnoser: An automated configuration error diagnosis tool for java software," in *Software Engineering (ICSE), 2013 35th International Conference on*, pp. 1438–1440, May 2013.

[34] A. Rabkin and R. Katz, "Static extraction of program configuration options," in *Software Engineering (ICSE), 2011 33rd International Conference on*, pp. 131–140, May 2011.

[35] X. Zhang, R. Gupta, and Y. Zhang, "Precise dynamic slicing algorithms," in *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, (Washington, DC, USA), pp. 319–329, IEEE Computer Society, 2003.

[36] M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, (Berkeley, CA, USA), pp. 1–11, USENIX Association, 2010.

[37] M. Attariyan and J. Flinn, "Using causality to diagnose configuration bugs," in *USENIX 2008 Annual Technical Conference*, ATC'08, (Berkeley, CA, USA), pp. 281–286, USENIX Association, 2008.

[38] A. Rabkin and R. Katz, "Precomputing possible configuration error diagnoses," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, (Washington, DC, USA), pp. 193–202, IEEE Computer Society, 2011.

[39] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang, "Automatic misconfiguration troubleshooting with peerpressure," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, (Berkeley, CA, USA), pp. 17–17, USENIX Association, 2004.

[40] A. Whitaker, R. S. Cox, and S. D. Gribble, "Configuration debugging as search: Finding the needle in the haystack," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, (Berkeley, CA, USA), pp. 6–6, USENIX Association, 2004.

[41] S. Zhang, Y. Lin, Z. Gu, and J. Zhao, "Effective identification of failure-inducing changes: A hybrid approach," in *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '08, (New York, NY, USA), pp. 77–83, ACM, 2008.

[42] S. Zhang, C. Zhang, and M. D. Ernst, "Automated documentation inference to explain failed tests," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, (Washington, DC, USA), pp. 63–72, IEEE Computer Society, 2011.

[43] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, (New York, NY, USA), pp. 1–10, ACM, 2002.

[44] H. Kim, Y. Jung, S. Kim, and K. Yi, "Mecc: memory comparison-based clone detector," in *Software Engineering (ICSE), 2011 33rd International Conference on*, pp. 301–310, May 2011.

[45] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, (Washington, DC, USA), pp. 96–105, IEEE Computer Society, 2007.

[46] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, pp. 654–670, July 2002.

[47] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Trans. Softw. Eng.*, vol. 32, pp. 176–192, Mar. 2006.

[48] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, (New York, NY, USA), pp. 321–330, ACM, 2008.

[49] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, (London, UK, UK), pp. 40–56, Springer-Verlag, 2001.

[50] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fix-points," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, (New York, NY, USA), pp. 238–252, ACM, 1977.

[51] Y. Jung and K. Yi, "Practical memory leak detector based on parameterized procedural summaries," in *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, (New York, NY, USA), pp. 131–140, ACM, 2008.

[52] Y. Xie and A. Aiken, "Context- and path-sensitive memory leak detection," *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 115–125, Sept. 2005.

[53] D. Binkley, N. Gold, and M. Harman, "An empirical study of static program slice size," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, Apr. 2007.

[54] M. Acharya and B. Robinson, "Practical change impact analysis based on static program slicing for industrial software systems," in *Software Engineering (ICSE), 2011 33rd International Conference on*, pp. 746–755, May 2011.

[55] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "Hampi: A solver for string constraints," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, (New York, NY, USA), pp. 105–116, ACM, 2009.

[56] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," *SIGPLAN Not.*, vol. 42, pp. 32–41, June 2007.

[57] F. Sun, L. Xu, and Z. Su, "Static detection of access control vulnerabilities in web applications," in *Proceedings of the 20th USENIX Conference on Security*, SEC'11, (Berkeley, CA, USA), pp. 11–11, USENIX Association, 2011.

[58] F. Yu, M. Alkhalaf, and T. Bultan, "Patching vulnerabilities with sanitization synthesis," in *Software Engineering (ICSE), 2011 33rd International Conference on*, pp. 251–260, May 2011.

[59] A. Groce, M. Musuvathi, F. Yu, T. Bultan, and B. Hardekopf, *String Abstractions for String Verification, Lecture Notes in Computer Science*, vol. 6823, pp. 20–37. Springer Berlin Heidelberg, 2011.

[60] W. G. Halfond, S. Anand, and A. Orso, "Precise interface identification to improve testing and analysis of web applications," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, (New York, NY, USA), pp. 285–296, ACM, 2009.

[61] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *Security and Privacy (SP), 2010 IEEE Symposium on*, pp. 513–528, May 2010.

[62] Y. Zheng and X. Zhang, "Path sensitive static analysis of web applications for remote code execution vulnerability detection," in *Software Engineering (ICSE), 2013 35th International Conference on*, pp. 652–661, May 2013.

[63] "Phc: open source php compiler @MISC."

[64] L. K. Shar and H. B. K. Tan, "Mining input sanitization patterns for predicting sql injection and cross site scripting vulnerabilities," in *Proceedings*

of the 34th International Conference on Software Engineering, ICSE '12, (Piscataway, NJ, USA), pp. 1293–1296, IEEE Press, 2012.

[65] L. K. Shar and H. B. K. Tan, "Predicting common web application vulnerabilities from input validation and sanitization code patterns," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, (New York, NY, USA), pp. 310–313, ACM, 2012.

[66] L. K. Shar, H. B. K. Tan, and L. Briand, "Mining sql injection and cross site scripting vulnerabilities using hybrid program analysis," in *Software Engineering (ICSE), 2013 35th International Conference on*, pp. 642–651, May 2013.

[67] A. Moller and M. Schwarz, "Automated detection of client-state manipulation vulnerabilities," in *Software Engineering (ICSE), 2012 34th International Conference on*, pp. 749–759, June 2012.