Static Vulnerability Analysis Using Intermediate Representations: A Literature Review

Adam Spanier and William Mahoney
University of Nebraska at Omaha, Omaha, USA

aspanier@unomaha.edu
wmahoney@unoamah.edu

Abstract: Static Analysis (SA) in Cybersecurity is a practice aimed at detecting vulnerabilities within the source code of a program. Modern SA applications, though highly sophisticated, lack programming language agnostic generalization, instead requiring codebase specific implementations for each programming language. The manner in which SA is implemented today, though functional, requires significant man hours to develop and maintain, higher costs due to custom applications for each language, and creates inconsistencies in implementation from SA-tool to SA-tool. A source of programming language generalization occurs within compilers. During the compilation process, source code is converted into a grammatically consistent Intermediate Representation (IR) (e.g. LLVM-IR) before being converted to an output format. The grammatical consistencies provided by the IR theoretically allow the same program written in different languages to be analyzed using the same mechanism. By using the IRs of compiled programming languages as the codebase of SA practices, multiple programming languages can be encompassed by a single SA tool. To begin understanding the possibilities the combination of SA and IRs may reveal, this research presents the following outcomes: 1) a systematic literature search, 2) a literature review, and 3) the classification of existing work pertaining to SA practices using IRs. The results of the study indicate that generalized Static Analysis using the LLVM IR is already a common practice in all compilers, but that the extended use of the LLVM IR in Cybersecurity SA practices aimed at finding vulnerabilities in source code remains underdeveloped.

Keywords: Vulnerability Detection, LLVM, Compilers, Intermediate Representation Analysis

1. Introduction

Static Analysis (SA) is a software practice focused on detecting patterns and deficiencies contained within the source code of a program, without executing the code. (Thomson, 2021; Bodden, 2018; Schmeelk et al., 2015) Originally used for early development program optimization (Bodden, 2018), SA is now widely utilized as an effective means to detect common vulnerabilities and weaknesses in static code bases (Nachtigal et al., 2022). In practice, SA can take on many forms of varying complexities. Some tools are concise, aiming only to detect formatting issues in code (Nachtigal et al., 2022; Dlint, 2022; JavaScript, 2022). Others offer the ability find minor bugs by observing common patterns occurring in the code (Nachtigal et al., 2022; PMD, 2022; SpotBugs, 2022). More complex SA implementations sometimes claim full vulnerability discovery capabilities (Nachtigal et al., 2022; Find Security Bugs, 2022; SecurityCodeScan, 2022).

As a well understood knowledge domain, current SA implementations have reached a level of sophistication and functionality that early implementations couldn't attain (Bodden, 2018; Johnson et al., 2013). This is due to streamlined pattern detection algorithms and more efficient data structure architectures contained within modern SA tools (Bodden, 2018). Even with high levels of sophistication and complexity, the core benefit of SA lies in the fact that a code-base containing potential malware can be analyzed quickly without risking infection of the host system (Johnson et al., 2013). This low resource, high security functionality means SA is still used extensively in most cybersecurity analysis activities (Johnson et al., 2013).

Even with the sophistication afforded most modern SA tools, several issues arise to the practical implementation of SA in large scale applications. With the blinding pace of technological improvement, the increasing levels of complexity, and the overwhelming sizes of modern application code bases, SA, even with it's dearth of research and sophistication, will always struggle to keep up with rapidly expanding technologies (Bodden, 2018). Due to the direction connection between the vulnerabilities and weaknesses discovered in source code and the SA application designed to find them, each SA tools must be both coded by hand and coded specifically for a single language (Bodden, 2018; Johnson et al., 2013). These laborious development requirements require that large SA tools must be coded to handle new programming languages that may or may not be rapidly evolving as the tool is developed. This can make SA tools lag behind in terms of functionality. Exacerbating the issue is the general lack of usability for most SA tools (Johnson et al., 2013). While the tool developers try to keep apprised of code base changes and new vulnerabilities, the user interface generally suffers from tacked-on additions and difficult-to-understand processes (Johnson et al., 2013). Breaking through these barriers requires a new way of thinking in order to alleviate the code-specific nature of SA tools (Bodden, 2018).

1.1 Compilers

Compilers are small software packages that translate one programming language into another (Zhong et al., 2009; Aho et al., 1986). The input language is called the source language, while the output is called the target language (Zhong et al., 2009). In general, compilers convert a high level programming language like Fortran, C, C++, or Java into lower level executable machine code; however, compilers are not limited to high/low conversions and can also convert low-level code to high-level languages (Aho et al., 1986).

To carry out this translation process, compilers first analyze the input code and break it into smaller, constituent pieces. These constituent pieces are then passed through a two-phase analysis process wherein the lexical and syntactical characteristics of the input program are parsed to create an *intermediate representation* data structure, often in the form of a *Syntax Tree*. The resulting *Syntax Tree* allows the compiler to generate a program language agnostic, low-level representation of any given program, much like a program meant for an abstract machine. During the analysis portion of the program, should the input code present any number of deficiencies, the compiler generally responds using compiler errors. On the other side of the intermediate representation, the synthesis portion of the compiler is responsible for converting the internal intermediate representation into the desired target program.

1.2 Intermediate Representations

One of the most important elements of intermediate representations is that they must be fast and efficient to both produce and translate. Intermediate representations processes and structures are used in every compiler activity, thus, all intermediate representations must be fast and efficient. Any lack of efficiency will cause the functionality of the compiler to suffer.

It is important to note that the intermediate representation of each compiler exhibits consistent grammatical characteristics regardless of the input or output format. This consistency is necessitates so that a program written in C or Fortran to carry out the same computational steps will be compiled into identical source code outputs. This fundamental consistency within compiler intermediate representations can act as a bridge between languages, describing the functionality of the language in a language-independent grammar.

1.3 Intermediate Representations and Static Analysis

This language-independent, inter-compiler grammatical consistency, when coupled with SA practices can theoretically facilitate the design and creation of a SA tool that analyzes the intermediate representation of the compiler rather than specific high level programming languages. By combining these two elements, the burden of recoding SA tools for new or changing languages can be mitigated in favor of SA tools that specifically analyze the consistent intermediate representation. This research aims to explore foundational literature to gain a better understanding of the state-of-the-topic regarding SA using intermediate representations. This exploration will be achieved by meeting the following objectives: 1) describe the problem, 2) design a systematic literature review, 3) carry out a literature survey, 4) synthesize the findings, and 5) discuss the outcomes.

2. Survey Design

The literature survey conducted in this research follows processes set by (de Sousa Borges et al., 2014). The literature survey follows the following four steps:

1. **Research Questions**: The research questions in this study fall into the following classifications: 1) pertinent literature, and 2) classifications.
2. **Literature Search**: Based on the research questions developed, a replicable process is implemented to gather generally relevant literature.
3. **Literature Selection**: The literature collected is further refined by using a replicable process to select relevant material.
4. **Literature Synthesis**: The literature collected is synthesized and classified to determine patterns or existing themes.

2.1 Goals and Research Questions

The goal of this study aims to both discover and synthesize pertinent literature related to the use of intermediate representations in Static Analysis activities. To achieve this goal, the following research questions are asked:

1. What literature exists relating the use of intermediate representations and compilers to Static Analysis?
2. What common themes and characteristics occur in existing literature?
3. Can these commonalities be used to classify pertinent literature?
4. What themes, information, and knowledge does existing literature present about the use of intermediate representations in Static Analysis practices?

3. Literature Survey

While the application of SA using IRs presents new and novel opportunities, existing literature provides useful insights as to the current state of the practice. The work reviewed in this survey generally falls into three categories: 1) motivating literature, 2) the use of IR in general program analysis, and 3) the use of IRs in bug detection and error discovery.

3.1 Motivating Literature

By its very nature, code is complicated. Adding to this inherent complexity, the progression toward more complex and ubiquitous technological solutions makes understanding modern applications all the more difficult (Thomson, 2021). In such an environment, programmers need as many tools as they possibly can to help manage and understand complex code. Static analysis (SA) is one such tool. SA is defined as "the extraction of facts from another program's source code, without executing the program in question and usually as a distinct stage in the day-to-day software development process" (Thomson, 2021).

3.1.1 Static Analysis

Thomson (2021) indicates that SA can manifest in the following ways: 1) discovery of security vulnerabilities, 2) code beautification and standardization, 3) dead code detection, 4) taint analysis, 5) race condition detection, 6) bounds checking.

One of the most common types of SA lies in it's prolific use in compilers. Compilers break source code into pieces, pass these pieces through lexical and syntactic static analysis processes and convert the resulting syntax trees into low level *intermediate representations* (Aho et al., 1986). The lexical and syntactical processes help determine the existence of compiler errors, code formatting issues, syntactical errors, uninitialized variables, or suspicious variable assignments in the code (Thomson, 2021). In this way, the core function of a compiler serves as a means to extract facts about the source code, thus tying compilers intrinsically to the SA process. In general applications, compilers derive facts about the source code and manifest the existence of those facts as machine code.

Modern SA implementations generally manifest through code specific analysis applications generally based in Version Control, Continuous Integration-based, or IDE based environments (Thomson, 2021). Though SA is currently considered a separate step in software development, compilers and SA often end up working synchronously, sometimes to the point of integration into new versions of compiler software. This relation, however, can also cause an arms race between SA and compilers. As code-bases get updated, modified, and become evermore complex, SA must attempt to remain effective within the context of the dynamic code-base. In current practice, this manifests through laborious efforts on the part of the SA tool developers. Such an inefficient process necessitates newer and more efficient analysis as code-bases get bigger and more complex (Thomson, 2021).

3.2 IR in General Program Analysis

Most general SA activities in compilers, should they occur, happen at the intermediate representation level (IR). Due to the consistency in program logic, structure, and control, IRs can provide a language agnostic access point for programs of varying source code languages.

The use of IR in general program SA remains somewhat novel in application. A number of limited feasibility studies and experiments relating directly to general program analyses using IR are described below (Kataev et al., 2018; Belyaev et al., 2013; Ghime et al., 2022; Khaldi et al., 2016).

3.2.1 SAPFOR: Automatic Parallelization using IR

Kataev et al. (2018) provide insight into the state of the numerous and complex computational environments that exist today and the challenges that are posed by the automatic parallelization processes required to fit into these environments. With so much processing power now available, effective automatic parallelization of compiled programs via the use of compilers remains somewhat elusive.

As stated by Kataev, *System FOR Automated Parallelization* (SAPFOR) is a tool that can simplify the development of automatically parallelized programs. SAPFOR functions by carrying out source-to-source transformations such that a single, linear progressing program can be divided into portions that can be allotted to parallelized computing systems. This process generates a need to better represent and analyze programs originating from multiple high-level programming languages; notably, C and FORTRAN.

To meet this need, the research provided by presents a novel auto-parallelization application using SAPFOR, LLVM, and the LLVM-IR. The program outlined by first examines compiled programs via the LLVM-IR, then determines memory locations, and ensures that no two pointers generate homogeneous memory locations. This process is carried out by analyzing IR-level memory locations and mapping them to higher level source code and creating what the researchers call a source-level alias tree.

The outcomes of the research carried out by indicate that the novel source-level alias tree created in this work provides usefulness in automation of parallelization in C and FORTRAN-based programs using the LLVM architecture. The primary contribution of this work is the novel application of LLVM-IR analysis for more efficient, automatic parallelization operations.

3.2.2 Type and Effect Systems in SA using IR

Belyaev et al. (2013) outline a novel system that implements type and effect systems to analyze programs using their derivative LLVM-IRs. As a feasibility study, this paper serves as means to determine whether such a tool is possible via the exploration of general concepts and a novel design.

Belyaev et al. chose the LLVM-IR approach due to the need for a more consistent, language agnostic approach to static analysis. C and C++ programs can prove complex when analyzed in their source format. An IR accurately represents the program regardless of source language, while providing less complex representations of the structures involved. The researchers in this case used the LLVM default Static Single Assignment (SSA) representation model for their IR. The LLVM SSA IR is called bitcode, and forms the input of the novel system proposed by the researchers.

The outcomes of the research carried out by Belyaev et al. saw the creation of both a bitcode parser and a basic analysis library and the development of a proof of concept type and effect system that can detect undefined variables.

3.2.3 IR Mapping in Incremental Analysis

Ghime et al. (2022) posit that the use of intermediate representations (IR) in SA can provide useful insights as to the functionality of a given program. They indicate that, while the use of IR objects in one-pass SA is very useful, each resulting IR of a given program as it changes over time generates different IDs for each program entity. For example, a variable contained within the program could receive a different unique identifier each time the IR is modelled. This creates significant boundaries to tracking program entity SA information through time.

Incremental analysis is the study of programs as they evolve over time. Incremental analysis is generally carried out on the source code of programs as it changes during the System Development Lifecycle. While this more traditional approach functions well in a historic sense, it can be laborious, inefficient, and language specific. The use of IRs in incremental analysis can allow for a more streamlined process as a program is analyzed over time.

The research carried out by Ghime et al. presents an accurate approach by which IR objects can be mapped to each other through time, such that SA information can be carried forward throughout the development of a software package.

3.2.4 Automatic HBM Allocation using IR

Khaldi et al. (2016) present a novel means by which High Bandwidth memory (HBM) can be automatically allocated during the execution of a program. Much like the research carried out by Kataev et al. (2018), Khaldi

et al. (2016) seek to implement the use of IR SAs so that the optimizations that occur within a compiler can be translated into more efficient computation. Where Kataev et al. (2018) seek to balance processing loads between multiple environments, Khaldi et al. (2016) seek to optimize the usage of memory allocations.

According to, HBM systems utilize a new three-dimensional memory stacking process by which memory chips are stacked vertically using processor dies. This stacking process allows the memory chips to work together and act like one single device. By changing the architecture, HBM allows for much higher bandwidth performance. While HBM achieves high bandwidth, traditional DDR memory still operates at a much lower latency, therefore making IO much faster to and from the DDR modules. These balanced benefits necessitate the dynamic combination of both HBM and DDR memory-types during execution to achieve the best computational output.

To meet the highly dynamic requirements needed to automatically allocate program entities to either HBM or DDR, chose to utilize a compiler-based optimization function that analyzes the IR of a given program and chooses which memory allocations using the *malloc* function should be given to HBM and which would be better served in DDR.

The outcomes of the work carried out by saw the implementation of the novel IR-based BCDA process in the LLVM compiler, and the subsequent dynamic allocation of HBW/DDR memory during execution. The resulting efficiency of the system increased the speed of the executing program by up to 2.29 times that of the normally executed binary (Khaldi et al., 2016).

3.3 IR in Static Vulnerability Analysis

Though the use of IRs in program analysis relates directly to the questions asked in this research, the scope yet remains beyond the desired aim of this work. To hone in on IR and Static Vulnerability Analysis pairings, studies relating directly to bug detection using IRs are required.

Much like general program analysis using IRs, vulnerability analysis experiments using IRs are limited to a small number of application and feasibility studies (Liang et al., 2016; Cassez et al., 2017; Fornaia et al., 2019; Schilling et al., 2022). While each study discovered relates directly to the problem in question, all outcomes thus far are limited to the detection of, at most, three types of software bugs.

3.3.1 MLSA: Static Bug Detection using IR

Liang et al. (2016) state that static analysis falls into five (5) categories: 1) control flow analysis, 2) data flow analysis, 3) model checking, 4) taint analysis, and 5) symbolic execution. Control flow analysis deals with operators contained in instructions. Data flow analysis concerns the operands that contain data in a program. Model checking constructs an automaton based on bug behavior patterns and checks for the existence of the automaton within the program. Symbolic execution sees an analysis of execution paths based on symbolic variable values.

Liang et al. present a static analysis tool called MLSA that uses the LLVM-IR to support limited bug discovery in multiple C-based programming languages. Currently MLSA can detect three bugs: 1) a division by zero error, 2) pointer overflows, and 3) dead code.

The system proposed by Liang et al. (2016) operates by using both Clang and Dragon Egg compilers to generate an LLVM-IR file from a C-based source code file. The LLVM-IR files generated in this research contain three (3) main sections: modules, functions, and basic blocks. The proposed program then processes the instructions in the module according to the Control Flow Graph (CFG) generated by the IR, and the relationships between the basic blocks contained in the IR. A Z3 SMT solver is then used to determine if bugs exist within the IR model. When a bug is found, MLSA marks the location of the bug so it can be found quickly. These locations are then written into a log file.

3.3.2 Skink: Static Program Analysis using IRs

The research carried out by Cassez et al. (2017) aims to implement LLVM-IRs to determine the existence of error traces in the logical architecture of a given C-based program. To do this Cassez et al. generate an LLVM-IR using the Clang compiler. The resulting IR is then mapped onto a CFG. This graph is a Finite Labelled Automaton (FLA) that implements two specific labels: basic blocks and choices. The FLA accepts a regular language comprised of a set of traces that lead to a given error block. These traces, called abstract error traces, are not guaranteed to be feasible within the actual program. To check the correctness of a given program, Cassez et al. simply need to

determine if an abstract error trace is feasible in the IR CFG. Cassez et al. check feasibility by encoding a given abstract error trace into a logical statement and checking if the resultant output is satisfiable using an SMT. If satisfiable, an error trace has been discovered and the program can be called incorrect.

The outcomes of the research carried out by Cassez et al. is comprised of a software package called Skink. The package includes the full LLVM-IR suite, a front end, middle-end, and a back-end architecture. Skink has been shown to work on any programming language that can be compiled into LLVM-IR. The major benefits, as stated by Cassez et al., lie in the program's ability to discover loop invariants and the ability to establish program correctness.

3.3.3 JSCAN: IR-Based Static Analysis Framework

Fornaia et al. (2019) propose a cross-language framework that utilizes LLVM-IR for the analysis of Object-Oriented code. By using the language-agnostic nature of the LLVM-IRs generated by LLVM compatible languages, the tool developed, called JSCAN, provides a readily readable representation of IR contents. SA tools like BCEL can then be built on top of the generalized IR framework.

As stated by Fornaia et al., the main contribution JSCAN provides lies in it's ability to augment the LLVM-IR with a more nuanced and structured information support system. The two main characteristics of the JSCAN framework are: 1) it is easy to understand and work with, and 2) it is uniform to abstraction of a tool called BCEL.

The aim of the research carried out by Fornaia et al. lies in a desire to see other developers take the frameworks and build more comprehensive and diverse tools on top of the code developed by this research.

3.3.4 VANDALIR: IR-Based Vulnerability Analyses

According to Schilling et al. (2022), custom built static vulnerability analysis tools are expensive, time-consuming, and inefficient. Many man-hours are required to develop, code, and maintain functionality for very limited, very specific applications. Though SA tools can find hidden vulnerabilities and complex patterns in program structures, the implementation, maintenance, and upkeep can be both logistically inefficient and quite expensive. These limitations drive the need for newer and more dynamic methods by which static vulnerability analysis can be carried out on wider code bases in more simple contexts.

Schilling et al. present a novel approach by which Datalog programming is modelled as an LLVM-IR and subsequent static vulnerability activities are carried out on the new, intermediate representation. As an added bonus, the researchers also generate a new static analysis detection rule that helps detect stack based memory corruption in 90% of test cases in LLVM-IR programs.

Table 1: Literature Review Findings

Authors	Origin	Purpose	Type of Source	Themes	Classification
Thomson (2021)	USA	To explore difficulties and potential improvements that can be made to static analysis in cybersecurity	Discussion	Generalized Static Analysis, Static Analysis using Compilers	Motivating
Kataev (2018)	Armenia	An exploration of IR analysis for automatic parallelization during compilation	Research	IR Analysis, Parallelization	General Program Analysis
Belyaev et al. (2013)	Russia	Research on IR parsing and the creation of a basic IR analysis library for type and effect system variable analysis	Research	IR Parsing, IR Analysis, Type and Effect Systems	General Program Analysis
Ghime et al. (2022)	India	To explore the application of consistent labels to IRs such that incremental analysis can be undertaken.	Research	IR Parsing, IR Analysis, IR Labeling, IR Objects, Incremental Analysis	General Program Analysis
Khaldi et al. (2016)	USA	Explore novel means to distribute memory allocations between High Bandwidth Memory and DDR Memory using IR analysis to modify system calls during compilation	Research	IR Parsing, IR Analysis, Compilation Modification, High Bandwidth Memory	General Program Analysis
Liang et al. (2016)	China	To explore the discovery of bugs in source code via IR parsing and	Research	IR Parsing, IR Analysis, Vulnerability Detection	Static Vulnerability

Authors	Origin	Purpose	Type of Source	Themes	Classification
		analysis.			Analysis
Cassez et al. (2017)	USA	To investigate the use of IRs in determining source code error traces, loop invariants, and program correctness	Research	IR Parsing, IR Analysis, Error Traces, IR Objects, Program Correctness	Static Vulnerability Analysis
Fornaia et al. (2019)	Italy	An exploration of a cross-language framework for the creation of IR objects that can be used with cybersecurity static analysis tools.	Research	IR Parsing, IR Analysis, IR Objects, Analysis Framework, SA Tool Integration	Static Vulnerability Analysis
Schilling et al. (2022)	Italy	To explore a novel IR analysis method based on Datalog programming for vulnerability and error detection	Research	IR Modelling, IR Parsing, IR Analysis, Vulnerability Detection	Static Vulnerability Analysis

4. Themes and Classifications

As an answer to RQ1, the literature outlined in the preceding sections provides a general understanding of how IRs are being used in Static Analysis activities. To answer RQ2, the common themes and characteristics discovered during this review fall into three areas: 1) literature that provides impetus and motivation for IR-based static analysis, 2) literature regarding general LLVM IR-based static analysis, and 3) literature carried out in exploration of static vulnerability analysis using LLVM IRs.

As an answer to RQ3, the relevant works selected during the course of this study fall into three broad categories as defined by the themes stated above: 1) motivation, 2) general program analysis using IRs, and 3) vulnerability detection using IRs. These categories also represent the most common themes discovered in the analysis of the literature surveyed.

To answer RQ4, the following information provides useful insights as to the use of IRs in Static Analysis. Motivating works serves not only as the motive to carry out the work in question, but also as a means to connect the proverbial dots between SA and the use of IRs. The foundational premises outlined by Thomson et al. (Thomson, 2021) drive the need to experiment with more efficient and diverse SA tools, while simultaneously outlining the intrinsic and inherent connection between compilers and static analysis. In this work, one piece of literature fell within the motivating category.

Many of the works selected for inclusion relate directly to the use of IRs in general program analysis. While the intent of this work is to determine the state of research based on static vulnerability analysis using IRs, the use of IRs in novel ways in program analysis practices can provide useful insights for novel static vulnerability techniques. Works included in this classification indicate the use of an IR via static program analysis. This work analyzes four general program analysis applications that incorporate IRs in the analysis structure.

The remaining works included in this research fall into the final and most pertinent category, vulnerability detection using IR. The works included in this classification demonstrate, in some capacity, the use of an IR in static vulnerability analysis. This classification contains the remaining four works analyzed in this study.

5. Conclusion

The use of intermediate representations in static analysis as a means to detect and discover vulnerabilities in software remains a relatively under-developed research area. While several applications analyzed in this study delved into the vulnerability discovery via IRs, the overall state of research relating to vulnerability detection in IRs is generally limited. At the same time, the use of IRs in general static analysis through internal compiler functionality has been discovered to be a standard practice. All compilers parse the logical structure of compiled programs to determine the existence of compiler errors. This connection is what drives the need to explore the relationships between static analysis practices, compilers, and IRs. The extraction of static analysis data from IRs outside the compiler structure, however, is still relatively unexplored.

Through the limited observations put forward by this research, more effort should be placed into a deeper understanding of how IRs can better be implemented in static vulnerability analysis. Should the state of practice of IR-based SA find a greater level of sophistication, the potential for full vulnerability detection integration within many compilers could potentially occur, adding significant security robustness to modern compiler

architectures. By streamlining and integrating such a process, most compiled programs could account better for basic vulnerabilities, therefore increasing general security overall.

Through the limited observations put forward by this research, more effort should be placed into a deeper understanding of how IRs can better be implemented in static vulnerability analysis. Should the state of practice of IR-based SA find a greater level of sophistication, the potential for full vulnerability detection integration within many compilers could potentially occur, adding significant security robustness to modern compiler architectures. By streamlining and integrating such a process, most compiled programs could increasingly search for basic cybersecurity vulnerabilities, therefore increasing general security overall. Our research team is currently exploring ways in which the LLVM-IR can be utilized in this way; we hope to present research papers with our results at a future ECCWS or ICCWS venue.

References

Aho, A., Sethi, R., and Ullman, J. (1986). Compilers: Principles, Techniques, and Tools. Pearson Education Inc.

Belyaev, M. and Tsesko, V. (2013). LLVM-Based Static Analysis Tool Using Type And Effect Systems. Automatic Control and Computer Sciences. Volume 46.

Bodden, E. (2018). Self-Adaptive Static Analysis. Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results. ICSE-NIER ’18. New York, NY, USA: Association for Computing Machinery. 45–48.

Cassez, F., Sloane, A.M., Roberts, M., Pigram, M., Suvanpong, P., and de Aledo, P.G. (2017). Skink: Static Analysis of Programs in LLVM Intermediate Representation. International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer. 380–384.

de Sousa Borges, S., Durelli, V.H., Reis, H.M., and Isotani, S. (2014). A Systematic Mapping on Gamification Applied to Education. Proceedings of the 29th annual ACM symposium on applied computing. 216–222.

Dlint. (2022). [Online]. Available: https://github.com/dlint-py/dlint

Find Security Bugs. (2022). [Online]. Available: https://find-sec-bugs.github.io/

Fornaia, A., Scafiti, S., and Tramontana, E. (2019). JSCAN: Designing An Easy to Use LLVM-Based Static Analysis Framework. 2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE). 237–242.

Ghime, V., Khadsare, A., Jana, A., and Chimdyalwar, B. (2022). IR Mapping: Intermediate Representation (IR) Based Mapping to Facilitate Incremental Static Analysis. 15th Innovations in Software Engineering Conference. 1–5.

JavaScript standard style. (2022). [Online]. Available: https://standardjs.com/

Johnson, B., Song, Y., Murphy-Hill, E., and Bowdidge, R. (2013). Why Don’t Software Developers Use Static Analysis Tools to Find Bugs? Proceedings of the 2013 International Conference on Software Engineering. ICSE ’13. IEEE Press. 672–681.

Kataev, N. (2018). Llvm-Based Approach to Static Analysis Of C Programs In SAPFOR. 19–23.

Khaldi, D. and Chapman, B. (2016). Towards Automatic HBM Allocation Using LLVM: A Case Study With Knights Landing. 2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC). 12–20.

Liang, H., Wang, L,. Wu, D., and Xu, J. (2016). Mlsa: A Static Bugs Analysis Tool Based on LLVM IR. 2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD). 407–412.

Nachtigall, M., Schlichtig, M., and Bodden, E. (2022) A Large-Scale Study of Usability Criteria Addressed by Static Analysis Tools. Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA ‘22. New York, NY, USA: Association for Computing Machinery. 532–543.

PMD. (2022). [Online]. Available: https://pmd.github.io/

Schilling, J. and Muller, T. (2022). Vandalir: Vulnerability Analyses Based on Datalog And LLVMiIR. Detection of Intrusions and Malware, and Vulnerability Assessment: 19th International Conference, DIMVA 2022, Cagliari, Italy. 96-115.

Schmeelk, S., Yang, J., and Aho, A. (2015) Android Malware Static Analysis Techniques. Proceedings of the 10th Annual Cyber and Information Security Research Conference. CISR ‘15. New York, NY, USA: Association for Computing Machinery.

SecurityCodeScan. (2022). [Online]. Available: https://security-code-scan.github.io/

SpotBugs. (2022). [Online]. Available: https://spotbugs.github.io/

Thomson, P. (2021). Static Analysis: An Introduction. The Fundamental Challenge of Software Engineering is One of Complexity. Volume 19, Number 4, 29–41.

Zhong, S., Shen, Y., and Hao, F. (2009). Tuning Compiler Optimization Options Via Simulated Annealing. 2009 Second International Conference on Future Information Technology and Management Engineering. 305–308.