Static Analysis: An Introduction

PATRICK THOMSON

**THE
FUNDAMENTAL
CHALLENGE
OF SOFTWARE
ENGINEERING
IS ONE OF
COMPLEXITY.**

The relationship between source code, computers, and people is complex. Though most code exists to be run on a computer, its purpose is not limited to that. It is also intended to be read and understood. The complexity of modern software development stands in opposition to the effort to understand code: Software that cannot be understood cannot be easily maintained or improved, and many empirical studies indicate a correlation between a given codebase's complexity and its rate of defects.³ It's difficult to understand a codebase of any significant size, and programmers therefore need and deserve every useful tool and technique to help them understand complex codebases.

One such category of tool, *static program analysis*, consists of programs or algorithms designed to extract 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. Software developers who use tools for static program analysis (usually referred to simply as *static analysis*) then have an opportunity to use the facts yielded by the analysis to further understand, evaluate, and modify the associated codebase.

The facts that can be extracted from source code

fall into many different categories. For example, a static analysis designed to discover security vulnerabilities would extract information about the functions and libraries a program used, whereas an analysis designed to beautify or standardize code layout would be concerned with the placement and positions of syntactic constructs. Other common facts extracted from code include dead code detection, which indicates code paths that can never be reached; usage of insecure APIs, such as C's `gets()` function; taint analysis, which identifies and monitors the use of values that could be tainted with malicious data; race condition detection, which identifies code patterns that produce incorrect or unpredictable results arising from interactions between threads of program execution; and bounds checking, which ensures that accesses to arrays or memory locations fall within the bounds that the programmer intended.

Static analyses are distinct from *dynamic* analyses such as `valgrind`, which extract facts from a program as it runs, and *model checking*, which verifies the correctness of a separate external specification of a program. Dynamic and static analyses are often used in tandem across many applications. Static analysis has some advantages over dynamic analysis. One such advantage is that static analysis generally operates over all possible branches of execution in a program, whereas dynamic analysis only has access to the currently executing code path. However, the converse is also true: Dynamic analysis has concrete information regarding the layout and position of data in the memory of the running program, whereas static analysis would have to guess as to how a given language, compiler,

operating system, and computer architecture would represent a given datum. There is also a degree of overlap; both dynamic and static analyses can detect the use of uninitialized variables in C-family languages, a common programming error.

Static analyses are ubiquitous in modern software engineering. Popular examples include security vulnerability scanners such as Coverity and CodeQL; programmer-error detection tools such as `scan-build`, an analysis tool provided by the LLVM project and targeting C, Objective-C, C++, and Swift; code formatters such as Python's `black` or Go's `gofmt`; and user-oriented editor tooling such as Rust's `rust-analyzer`.

The notion of static analysis also lies at the heart of several other fields. Compiler design, for example, is its own area of active research, but static analysis is a fundamental building block within the compiler author's toolkit, and most compilers run many separate static analyses before and after code is generated. Indeed, you can think of a compiler, in the large, as a static-analysis tool under which the facts generated consist of an executable program, as well as any applicable debug information. The term *static analysis*, however, generally refers to external tools that can be used alongside a compiler or build system.

There are some fundamental limits to the practice of static analysis. One useful kind of static analysis, *termination analysis*, is concerned with predicting, given some source code, whether that code eventually finishes running or loops infinitely. Alan Turing, in a groundbreaking 1936 proof,⁷ stated that no algorithm is capable of

determining this property for all possible program inputs. Termination analyses, along with many other program properties that can be proved equivalent to the halting problem, therefore have an upper limit in terms of their capabilities. Such an analysis can approach, but will never reach, perfection for all possible inputs. This limitation means that static analyses are often restricted to yielding approximations of program behavior. An approximation, however, may often be sufficiently useful in practice.

A HISTORICAL PERSPECTIVE

To discuss the history of static analysis, we must draw a distinction between a *pass* and a *tool*. A static-analysis pass consists of a walk-through of a given piece of code that extracts a fact or set of facts from that code, whereas a static-analysis tool is a program, usually separate from the mechanism of program execution (such as an interpreter or compiler), that uses one or more analysis passes to yield a set of facts to the user. Static analysis passes emerged as a part of existing programs before the advent of stand-alone tools.

Adm. Grace Hopper's seminal 1952 paper "The Education of a Computer" specified a set of facts to which a compiler needed access in order to generate code correctly.² The first development of *type inference* algorithms came in 1958; these were procedures capable of both checking the declared types of values in a program, and, conversely, inferring the types of values without explicit declarations. The next decade saw an influx of research on analysis passes within optimizing compilers; a 1965 paper from RCA Laboratories outlined a method of writing a program

that “would examine any other program and perform such simplifications on it as can be detected from the argument-program’s form alone, without having any knowledge of what it will do.”⁶

Today’s notion of static-analysis tools, invoked as a stage separate from a compiler or interpreter, developed in the 1970s during the emergence of modern software development techniques. Among the earliest popular analysis tools was Stephen C. Johnson’s `lint`, written in 1978 and released to the public with 1979’s Version 7 Unix, which checked C programs for errors. The C compilers of that era performed far fewer correctness checks than do modern compilers, and `lint` introduced several analyses of enduring popularity, such as warnings regarding suspicious type conversions, non-portable constructs, and unused or uninitialized variables—although nowadays those warnings are often part of C compilers themselves.

While `lint` filled an important niche and saw wide use, it was prone to emitting false-positive results, which required programmers to annotate their programs with auxiliary information intended to suppress warnings. `Lint` proved so influential that it bequeathed its name to a whole class of tools, *linters*, across many programming languages.

Concurrently, static analysis received attention from theory-oriented branches of computer science. Patrick and Radhia Cousot introduced the theory behind *abstract interpretation* in a 1977 paper at the ACM SIGPLAN conference, providing a formal mathematical approach to static analysis.¹ Abstract interpretation formalized the notion of an approximation of program behavior: The

halting problem, along with the many other problems that can be reduced to the halting problem, is uncomputable, meaning that approximation of behavior is the only viable path forward.

The abstract interpretation approach allows an analysis to consider all possible branches of control flow within a problem, and details how the approximate behavior of each part of the program can be composed into an approximate behavior of the entire program. Abstract interpretation proved an extremely useful approach in practice; many subsequent analysis tools were built atop abstract interpretation, and it remains an active area of research.

Over time, static analysis became a fundamental part of many software development approaches. Among the most prominent users of static-analysis tools was NASA [National Aeronautics and Space Administration]. Its *Software Engineering and Assurance Handbook*, a document establishing and sometimes mandating aspects of the software development process, established in its earliest versions that NASA software must undergo static analysis by available analysis tools.⁴ This handbook established static analysis as a fundamental responsibility of the project manager and associated engineering groups. Companies developing safety-critical software, such as avionics software or firmware for medical devices, have adopted static analysis enthusiastically; coding standards such as MISRA C, a popular set of guidelines for developing safety-critical embedded systems in C developed by the Motor Industry Software Reliability Association, mandate the use of such tools.

IMPLEMENTATION AND DEVELOPMENT

Just as `lint` was targeted toward C, most real-world static-analysis tools target a single language or language family because of the wide variations in syntax and semantics among languages. The capabilities of a language have an immense impact on the behavior and requirements of any associated analysis tool; for example, tools targeting languages that support runtime code evaluation (`eval()`) of strings, such as JavaScript and Python, must take those capabilities into account. A comprehensive vulnerability scanner for Python would warn the programmer if it encountered a code path that called `eval()` on untrusted user input. In a language without `eval()` the programmer would not need to take this into account. Conversely, analyses targeting high-level languages do not need to account for the perils of low-level memory access.

There are many ways to introduce static analysis into a codebase. Among the most popular are those that integrate with VCS (version-control software) such as Git; many VCS services, free or commercial, provide a platform for integrating static-analysis tools so that analyses are performed on demand when new code is pushed to a repository. Static-analysis tools also integrate with CI (continuous integration) services, which manage the process of building and packing software as code is added or removed. Most CI services allow their users to specify that their build process should fail if an analysis tool reports unexpected results. Users who want to write their own static analyses often write extensions for an existing analysis framework; for example, `scan-build` provides an API enabling end users to hook into LLVM's internal

processes and take advantage of LLVM's rich facilities for traversing and analyzing a program's syntax tree.

It's worth noting that the interchange between static analyses and their associated languages is bidirectional; as analyses evolve, they drive the development of compilers and associated tooling. A modern example is the development of the Swift programming language. Swift's predecessor, Objective-C, originally used a style of memory management, manual reference counting, that required the programmer to specify when a given object's memory should persist (a "retain") and when it should be relinquished (a "release"). This process was error-prone; consequently, the static-analysis project associated with the Clang compiler introduced the ability to check that the programmer inserted retains and releases correctly.

The fact that this management could be checked for correctness prompted a realization that Objective-C memory management in its entirety could be automated by the compiler. This insight led to the introduction of ARC (automated reference counting), which paved the way for a language with a formal memory management policy—namely, Swift itself—in which memory management is usually entirely automated. Similarly, the existence of the literature describing type-inference analyses inspired the development of the ML language family and helped introduce type inference into languages that lacked it, such as C++.

MODERN PRACTICE

Static analysis manifests itself in the practice of programming in several ways. The most immediate

As codebases grow larger, programmers need more sophisticated and efficient analyses.

methods of analysis involve end users running the analysis on their local machines. Many popular text editors and IDEs (integrated development environments) integrate static-analysis tools automatically, providing analysis feedback directly to programmers as they develop their software.

As mentioned previously, CI and VCS services often provide hooks to integrate static analysis into the development and build process. Yet a great many static analyses are invisible. A compiler, after all, is itself a static analysis, built out of dozens of individual analysis passes, that yields an artifact executable by a computer. Even text editors perform their own analysis. Syntax highlighting is common to almost all editors and is a static analysis that yields information about the semantic role of the identifiers and keywords used in a program. Additionally, a significant proportion of software used during development has been statically analyzed, down to the operating system and even perhaps the CPU's microcode.

Not all analyses are feasible in practice. The larger a codebase becomes, the longer it takes to parse and traverse; in addition, many static analyses are computationally expensive—often quadratic, sometimes even cubic—in terms of space or time needed to perform them. Consequently, a sort of arms race exists between static analyses and the codebases being analyzed. As codebases grow larger, programmers need more sophisticated and efficient analyses.

One barrier—perhaps the most important barrier—to programmer adoption of static-analysis tools is the requirement that humans change their behavior to

account for the issues discovered and the warnings that arise. Since the days of `lint`, programmers have struggled to silence warnings associated with false-positive results from a given analysis; the remedy often entails inserting “magic comments” into code, relying on the given analysis tool to scan and appropriately disable the corresponding warning. It can be tedious to insert these directives by hand, however, and programmers will often elect not to use a static-analysis tool that yields too many false positives.

Similarly, false negatives, such as security bugs that go undetected, can give programmers an unfounded confidence in the correctness of their code. Programmers can work around false positives with careful configuration of a given tool; false negatives are more difficult to discover, though the risk can be reduced by using multiple static-analysis tools in tandem. Additionally, an analysis may detect issues that are valid in theory but benign in practice, such as violations of the numerous and fussy restrictions on named identifiers in C.

TOWARD THE FUTURE

Modern static-analysis tools provide powerful and specific insights into codebases. The Linux kernel team, for example, developed Coccinelle, a powerful tool for searching, analyzing, and rewriting C source code; because the Linux kernel contains more than 27 million lines of code, a static-analysis tool is essential both for finding bugs and for making automated changes across its many libraries and modules. Another tool targeted at the C family of languages is Clang `scan-build`, which

comes with many useful analyses and provides an API for programmers to write their own analyses.

Cloud-based tools such as LGTM.com integrate with existing build and release processes and work across a variety of programming languages. High-level protocols relating to static analysis have also emerged. The Language Server Protocol, a set of common definitions standardizing the ways in which analysis tools interface with text editors such as Emacs and VS Code, ensures that analysis tools can integrate with programmers' workflows regardless of their choice of tooling; similarly, SARIF (Static Analysis Results Interchange Format) provides a standard for the output produced by static-analysis tools.

The subfield of static analyses targeted toward detection of security vulnerabilities continues to attract industrial and research attention. As the consequences of security vulnerabilities become more dire, the utility of static analysis increases. Bugs such as Spectre (CVE-2017-5753), which exposed security flaws associated with speculative execution within a computer's CPU, prompted the development of analysis tools geared specifically toward the detection of that category of exploits. Many analysis tools draw upon the CVE (Common Vulnerabilities and Exposures) and CWE (Common Weakness Enumeration) databases of specific vulnerabilities and antipatterns.

The fundamental challenge of software engineering is one of complexity. Large software products are among the most complicated human endeavors ever attempted. Further adding to this burden is its short history—humans have been building software for barely 50 years, in

contrast with the millennia of history behind other fields such as architecture or medicine.

Atop all that, the mechanics of software production grow more complicated every year, as new platforms, operating systems, and programming languages emerge. The fact that the complexity of industrial software generally grows faster than its authors' ability to manage that complexity is one of the central challenges facing software engineering and computer science as disciplines. We have precious few weapons in this fight, but static analysis is among the most effective, as demonstrated by numerous meta-analyses of development effort and defect rate.⁵

Like so many things in computer science, the utility of static analysis is self-referential: To write reliable programs, we must also write programs for our programs. But this is no paradox. Static-analysis tools, complex though their theory and practice may be, are what will enable us, and engineers of the future, to overcome this challenge and yield the knowledge and insights that we practitioners deserve.

References

1. Cousot, P., Cousot, R. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Los Angeles, California, 238-252. New York, NY: ACM Press; <https://dl.acm.org/doi/10.1145/512950.512973>.

2. Hopper, Adm. G. M. 1988. The education of a computer. *Annals of the History of Computing* 9(3-4), 271-281; <https://dl.acm.org/doi/10.1109/MAHC.1987.10032>.
3. Kemerer, C. F. 1995. Software complexity and software maintenance: a survey of empirical research. *Annals of Software Engineering* 1(1), 1-22.
4. NASA Office of the Chief Engineer. 2011. *Software Engineering and Assurance Handbook*. National Aeronautics and Space Administration. SWE-135. <https://swehb.nasa.gov/display/SWEHBVC>.
5. Nichols, W. R., Jr. 2020. The cost and benefits of static analysis during development. arXiv:2003.03001; <https://ui.adsabs.harvard.edu/abs/2020arXiv200303001N>.
6. Nievergelt, J. 1965. On the automatic simplification of computer programs. *Communications of the ACM* 8(6), 366-370; <https://doi.org/10.1145/364955.364963>.
7. Turing, A. M. 1937. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society* 2-42(1), 230-265; <https://doi.org/10.1112/plms/s2-42.1.230>.

Patrick Thomson is a senior engineer at GitHub Inc., working on static analysis of the world's largest corpus of code. He lives in New York City.

Copyright © 2021 held by owner/author. Publication rights licensed to ACM.