The Challenges of Shift Left Static Analysis

Quoc-Sang Phan
Meta
USA

Kim-Hao Nguyen
University of Nebraska-Lincoln
USA

ThanhVu Nguyen
George Mason University
USA

***Abstract*—In the software development industry, static analysis is used early in the development process, often as soon as a source file is saved or immediately after a commit is submitted. This approach, known as "shifting left," helps identify and address potential issues early on, before they become more difficult and costly to fix. However, implementing this approach can be challenging, as demonstrated through our experience at Meta. One specific challenge is dealing with build issues and figuring out which files and code to give to the static analyzers. These problems arise because the compilation and inclusion of files depend on various configurations, such as compilation flags and the platforms used to build the files. The purpose of this paper is to highlight these challenges and encourage the research community to find solutions.**

***Index Terms*—static analysis, shift left, configurable software, conditional compilation, variability, build system**

I. Static analysis

In recent years, static analysis has been gaining popularity in the industry [1]–[3]. These tools have been shown to be effective in identifying serious vulnerabilities, such as those listed in the OWASP Top 10 [4] or CWE Top 25 [5]. Additionally, they can scale to large codebases, e.g., Zoncolan [6] is used at Meta to analyze thousands of changes daily across 100 million lines of code. Furthermore, static analysis provides useful information, including source file and line numbers, which can assist developers in understanding and resolving defects. Moreover, static analysis can provide useful trace information, including source file and line numbers, which can assist developers in understanding and debugging defects.

Meta developers consider static analysis to be an essential component of their continuous development process [7]. For example, the developer Alice might use an integrated development environment (IDE) with fast intraprocedural analyzers such as Clang-Tidy [8] and Clang Static Analyzer [9] to quickly report defects when the source file is saved. After Alice commits, her code changes are sent to server running Phabricator [10], which creates a *diff* entry for code review. Phabricator then launches a series of jobs that use cloud machines to clone the repository with Alice's commit, run the diff through tests, and employ more expensive interprocedural analysis tools including Infer [11], Pysa [12], and Zoncolan [6]. If all tests pass, Alice can then request code review, which includes feedback and suggestions from the analyzers, and commit the diff to the main codebase.

At Meta, the use of static analyzers, particularly those that focus on identifying security vulnerabilities (such as static application security testing, or SAST), has been emphasized and shifted to the "left" of the software development life cycle between the coding and review phases:

coding → ***static analysis*** → *code review*→
integration → deployment

Thus, SAST tools are applied early, often as soon as a source file is saved or a commit is submitted for code review. This approach is appealing because it allows developers to discover and fix bugs and security vulnerabilities at earlier stage, making debugging more efficient and cost-effective. In this paper, we use the term "*shift left static analysis*" or *SLSA* to refer to this practice of shifting static analysis to the left.

As with any approach, SLSA has its own set of challenges. Distefano et al. [2] has identified issues with the scalability and accuracy of SAST tools such as Infer and Zoncolan. However, as we will describe below, even the initial step of *applying* these tools to code presents unique difficulties, especially in complex and dynamic development environments such as the one at Meta. The goal of this paper is to outline these challenges and motivate further research to address them.

II. Challenges in shifting left

The *first* law of find bugs in software is that *"you cannot check code that you do not see"* [1]. For a SAST tool to properly analyze a system, it must have access to the program code and know how to build it. This is typically achieved by providing the SAST tool with a build configuration, such as a script that specifies the compilation flags and files to be built. The tool then initiates the build process, captures information about the included files and how they are compiled, and analyzes them. Many major SAST tools support this process, including Coverity and Facebook Infer, which can intercept build system calls to capture compiled files and commands, and Infer and Clang-Tidy, which can consume a compilation database generated by CMake.

However, this seemingly innocent task of producing relevant source code to a SAST tool can become challenging for complex and highly-configurable systems that import or include many external code files or libraries. For example, some systems include third-party libraries that are only included under specific configurations, such as when compiled with certain flags or options. In the past, a security expert would manually select the appropriate configuration when running the SAST tool. However, in the SLSA approach, SAST tools must be automatically configured and run on an engineer's

```
void buffer_overflow(char* tainted_data){
#ifdef FLAG_BAR
  char buf[20] = "Hello World!";
  std::memcpy(buf, tainted_data, sizeof(tainted_data));
#endif // #ifdef FLAG_BAR
}
```

Fig. 1. `baz.cc`

machine or in the cloud with no manual intervention. This introduces several new challenges described below.

A. *False negatives caused by missing files*

When thousands of engineers work together on a cross-platform multi-feature application, the code they submit are often compiled with different sets of compilation flags. This results in a wide range of possible configurations. Without the correct configuration, such as one that with options or flags that include the buggy code, SAST tools will not even have a chance to see and identify bugs in code. This can produce false positives, where the SAST tools conclude unsafe code as safe—a soundness issue.

Consider the scenario where Alice submitted a diff that adds a new buggy source file `baz.cc` shown in Fig. 1. Because `memcpy` does not check if the number of bytes to copy is smaller than the size of the buffer to copy to, this code snippet has a potential buffer overflow vulnerability, allowing a malicious user to execute arbitrary code. While being dangerous, this kind of vulnerability and many others involving unsafe memory usage *can* be detected using the popular taint analysis supported by many SAST tools.

However, defects in the code may be missed by SAST tools if the tool does not have access to the newly added code. For instance, consider the scenario where the file `baz.cc` implements an experimental feature that is not ready for production. To build this feature on-demand, Alice adds a new flag `FLAG_FOO` to `CMakeLists.txt` in the same diff to control the inclusion of `baz.cc`. As a result, `baz.cc` is only compiled when CMake is run with the option `-DFLAG_FOO`.

Because it is not possible to predict which flags will be added to the program, the automated Phabricator process will trigger the SAST job with only one hardcoded configuration that does not include additional flags such as `FLAG_FOO`. Thus, SAST tools may not have access to the code that includes the new feature and the bugs in it may not be detected.

B. *False negatives caused by missing code*

This problem is related to missing files and code, as discussed in Section II-A, but is caused by the use of the preprocessor directive `#ifdef` in C and C++. For example, if the vulnerable code snippet in Fig. 1 is only included when the compilation flag `DFLAG_BAR` is set, then even if `baz.cc` is checked by the SAST tool, the vulnerable code would still not be analyzed, unless `baz.cc` is built with the flag `-DFLAG_BAR`.

While false negative caused by conditional compilation has been studied (e.g., [13]), existing research has been focusing on *finished products*, and do not address this problem at diff time. For example, the VAMPYR tool in [14] can identify a set of configurations that cover all `#ifdef` blocks of changed files, such as `baz.cc` in our example. However, this does not solve the problem of diff-time analysis as we only need to find the `-DFLAG_BAR` that covers the changed blocks, not all blocks in the file.

There are existing static and dynamic (sampling) techniques in variability-aware analyses. However, they generally do not scale well with an exponentially large number of states. For example, the WhatsApp VoIP module alone has nearly two millions LoC and more than 10,000 `#ifdef` directives, resulting in over a billion possible states. The work in [15] suggests that these techniques are not sustainable and have not been widely adopted by the industry or that they are only effective for scenarios with a small configuration space (e.g., Cppcheck by default analyzes all configurations of the program [16]).

C. *Optimization for intraprocedural analysis*

Suppose we develop a safer function, `safe_memcpy`, that performs bound checking before memory copying. However, this function can only be used when the destination is a static array (as in the example in Fig. 1). Thus, we want the SAST tool to automatically replace all instances `memcpy` where the destination is a static array with the safer `safe_memcpy` function. In this case, we can search for this pattern on the abstract syntax tree of `baz.cc` and ignore all files that are not affected by the diff.

Currently, there is no efficient method available to analyze and extract the compile command for `baz.cc` in large build projects. This means that the SAST tool must initiate the entire build process, which can take several hours, even though the actual analysis only takes seconds. This creates a significant inconvenience for Meta engineers as they have to build the entire project to obtain the necessary information for static analysis to run. This disrupts the workflow of users and hinders the adoption of SLSA. Even at diff time, adding several hours to the Phabricator job of hundreds or thousands of diffs requires huge cloud resources, which may lead to pushback from users.

D. *Conclusion*

We list several challenges to the SLSA approach, which encourages using SAST tools in early development cycles. While there are many limitations in SAST tools themselves [2], we argue that even gathering sufficient information, such as files and compilation flags, to give to an SAST tool is a tedious and manual task. Moreover, build languages do not have well-defined syntax and semantics, making the problem even more difficult. For example, CMake is Turing complete and supports arbitrary command invocations [17]). Being able to effectively overcome these challenges will further promote the use of SAST tools and in particular make the SLSA approach used at Meta and other companies realizable.

References

[1] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, p. 66–75, Feb. 2010. [Online]. Available: https://doi.org/10.1145/1646353.1646374

[2] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn, "Scaling static analyses at facebook," *Commun. ACM*, vol. 62, no. 8, p. 62–70, Jul. 2019. [Online]. Available: https://doi.org/10.1145/3338112

[3] F. Raimondi and B.-Y. E. Chang, "How automated reasoning improves the Prime Video experience," https://www.amazon.science/blog/how-automated-reasoning-improves-the-prime-video-experience.

[4] "OWASP Top Ten," https://owasp.org/www-project-top-ten/.

[5] "CWE Top 25 in 2022," https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html.

[6] "Zoncolan: How Facebook uses static analysis to detect and prevent security issues," https://engineering.fb.com/2019/08/15/security/zoncolan/.

[7] D. Feitelson, E. Frachtenberg, and K. Beck, "Development and deployment at facebook," *IEEE Internet Computing*, vol. 17, no. 4, p. 8–17, Jul. 2013. [Online]. Available: https://doi.org/10.1109/MIC.2013.25

[8] "Clang-tidy," https://clang.llvm.org/extra/clang-tidy/.

[9] "Clang Static Analyzer," https://clang.llvm.org/docs/ClangStaticAnalyzer.html.

[10] "Phabricator," https://en.wikipedia.org/wiki/Phabricator.

[11] "Facebook Infer," http://fbinfer.com/.

[12] "Pysa: An open source static analysis tool to detect and prevent security issues in Python code," https://engineering.fb.com/2020/08/07/security/pysa/.

[13] P. Gazzillo and S. Wei, "Conditional compilation is dead, long live conditional compilation!" in *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER '19. IEEE Press, 2019, p. 105–108. [Online]. Available: https://doi.org/10.1109/ICSE-NIER.2019.00035

[14] R. Tartler, C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann, "Static analysis of variability in system software: The 90,000 #ifdefs issue," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 421–432. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/tartler

[15] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 643–654. [Online]. Available: https://doi.org/10.1145/2884781.2884793

[16] "Cppcheck," https://cppcheck.sourceforge.io/.

[17] K. Nguyen, T. Nguyen, and Q.-S. Phan, "Analyzing the cmake build system," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2022, pp. 27–28.