Role of Program Analysis in Security Vulnerability Detection: Then and Now

Cristina Cifuentes, François Gauthier, Behnaz Hassanshahi,
Padmanabhan Krishnan, Davin McCall
Oracle Labs, Australia
Email:{cristina.cifuentes,francois.gauthier,behnaz.hassanshahi,paddy.krishnan,davin.mccall}@oracle.com

Abstract

Program analysis techniques play an important role in detecting security vulnerabilities. In this paper we describe our experiences in developing such tools that can be used in an industrial setting. The main driving forces for adoption are low false positive rate, ease of integration in the developer's workflow and results that are easy to understand. We also show how program analysis tools had to evolve with the evolving needs of the organisation. We conclude with our vision on how program analysis tools will be melded with DevSecOps.

1 Introduction

Security of applications is a continuously changing domain. However, from an cybersecurity perspective this domain is not well understood. Topics such as infrastructure, perimeter, or network security are better understood than application-level security. Consequently, the role of tools such as firewalls and virus scanners is self-evident. But given that many organisations use large custom software-based applications for their core business, security of these applications are of concern to the software developers. There is no single class of tools that are associated with application-level security.

This problem is further compounded by the fact that in early days, there were very few people who supported security as an integral part of the development cycle. Techniques such as secure coding guidelines, which can be checked using lightweight tools, were not enforced [JR04, EGLPAMF20].

The reactive approach to fixing security problems when they arise can be expensive, challenging, and time consuming. Hence, more recently, the shift-left approach to security has been adopted by software developers. Program analysis tools, although not as well understood as tools such as firewalls or virus scanners, have become key components in the software development life cycle. They are used to detect a variety of defects and give rapid feedback to the developer. There are many commercial tools that have been integrated into the CI/CD pipelines. This ensures that the defects can be fixed by the developer before the software is released. Because coding errors are the reason for a number of security vulnerabilities, over the years program analysis tools have evolved from detecting general software bugs to detecting security-related defects.

As software systems have evolved, the types of security vulnerabilities have also evolved. Thus the tools developed for a specific class of applications cannot always be used for another class of applications. For instance, tools that work on system code are not useful for web-based applications. This imposes additional burden on the tool developers to keep their tools relevant. If these tools are not easy to integrate with the developer's workflow, the developer has to master a number of tools. It

is imperative that the analysis tool developers and the application developers (who use the analysis tools) work together to reduce the overheads of adoption in practice.

One key question is what are security vulnerabilities. Without a clear definition, the usefulness and effectiveness of tools cannot be evaluated. Asking the developer to provide such information is not practical. Because most developers involved in developing large complex applicable are skilled, the defects that are present in their code are because of the complex interactions between different components. Hence simple tools like linters and AST-based scanners are not sufficient. Program analysis tools, that are more advanced than linters and scanners are required. However, they impose a high cognitive load on the developers. The developers need to understand how to use the analysis tools, understand the reports produced by the tools and find a solution to fix the vulnerability without breaking changes. Because most developers are not security experts, they cannot provide security-related information that can guide tools. Any tool that burdens the developer with defining security vulnerabilities for their applications will fail. Hence security issues that are identified by the security community (e.g., OWASP Top 10 [OWA]) need to be used to drive the analysis.

In this paper we summarise more than a decade of our experience with program analysis tools and their evolution, especially in the context of security analysis. This evolution was influenced by the following factors. 1. The changes in the needs of the organisation; 2. The requirements of the developers consuming our tools; 3. The varying landscape in the technology used to develop applications; and 4. The analysis techniques developed by the research community at large.

The paper is structured as follows. Section 2 discusses details of the evolution of our tools that includes the discussion of static analysis 2.1 and dynamic analysis 2.2. Section 3 summarises the lessons we have learned from developing the variety of tools. Section 4 concludes the paper with our vision of how such tools will have an impact on the software development landscape that continues to change rapidly.

2 Evolution of Application Domain and Techniques

Up until the early 2000s, most systems were developed using C. The most common errors and vulnerabilities were related to memory management including buffer overflows, use after free and illegal pointer references. Bug detection approaches to support the developer therefore focused on modelling memory including arrays and numeric expressions that were used to access the memory [MRS10]. The effectiveness of static analysis tools led to commercial bug-finding tools [BBC+10] where the focus was not on soundness but finding bugs that matter on large codebases. Given the size of the codebases, techniques that helped the developers understand the code were also needed.

In the early 2000s, however, Java[1] rose to prominence as web applications became ubiquitous and dominated the programming language landscape for roughly 15 years. Being a memory-safe language, the analyses that were relevant to C did not apply to Java and much of the focus shifted to detecting null pointer exceptions and injection vulnerabilities. While modelling memory was necessary for detecting null pointer exceptions, the specifics were different than those used for C programs. Here one needed a model for the heap where objects were stored and what variables and named-fields "pointed-to" each object. For injection vulnerabilities, taint analysis which was not directly relevant for C programs had to be developed. Because of various security vulnerabilities reported, the techniques had to address both the JDK [2] (the Java Platform) and Java-based applications.

[1] Java is a registered trademark of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

[2] JDK is a registered trademark of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

In recent years, languages like JavaScript and Python have gained in popularity, calling for a new generation of analysis tools capable of producing precise reports despite their highly dynamic nature. A specific problem is that of call graph generation as it underpins all analyses. Figure 1 presents the overview of our experience in the evolution of the requirements and hence the development of specific tools and techniques. These tools and techniques will be expanded in the following sections.

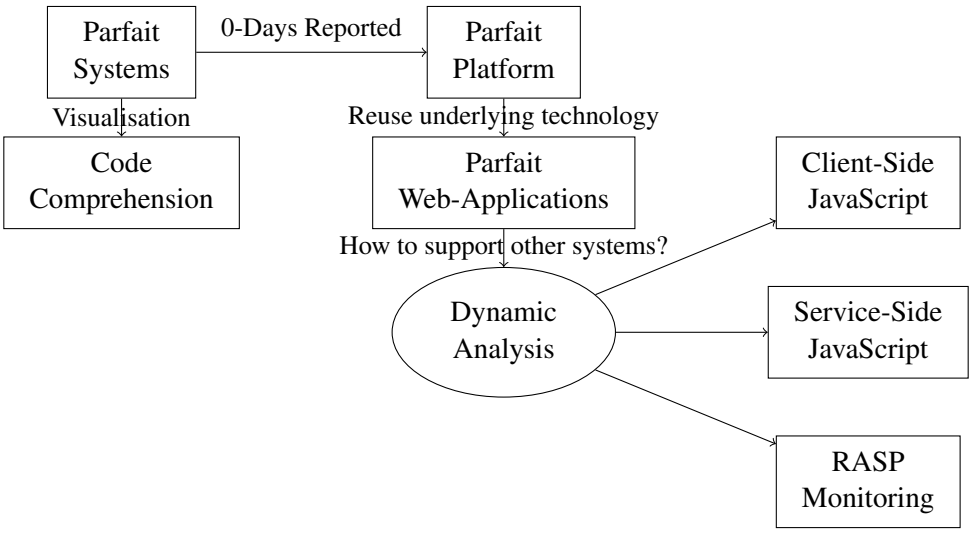

Figure 1: Summary of Evolution

2.1 Static Analysis: Parfait

Systems Level Code From the perspective of the user, i.e., the developer, the four main properties desired from static analysis tools are precision, recall, scalability, and usability. Circa 2007, commercial static analysis tools were suffering from the following three major flaws. 1. They produced large amounts (e.g. 30%-50%) of false positives; 2. They took days or weeks to analyse large (e.g. MLOC) code bases; and 3. They did not integrate well with existing build processes.

From a developer's view, the above properties do not have the same priority. An imprecise analysis will waste the developer's time, cause frustration and be abandoned quickly. A slow analysis hinders usability by forcing developers to context-switch back to potentially outdated prior work and account for changes that happened since the analysis ran. A tool that cannot be integrated smoothly with the developer's workflow is unlikely to be used. An analysis that produces correct, but hard-to-understand reports will be mis-represented as producing false positives. Optimising for any one of the desired features invariably affects the other features adversely. From a practical perspective, a fast and precise analysis that misses some defects but measurably improves security by eliminating defects is thus far better than a high recall analysis that is not used at all.

These expectations from the developers and the limitations of commercials tools on our internal codebases were recognised, which led to the development of the Parfait static analyser for C programs [CKL+12]. From its early days, the main focus of Parfait was to deliver a highly scalabile and precise analyses. Specifically, the aim was for Parfait to take less than 10 min per MLOC on a standard desktop machine, and a false positive rate of less than 10%.

Achieving high scalability and precision is only the first step to adoption, however. Even the best analysis tool will be shelved unless non-expert developers can also run it, integrate it in their development lifecycle, understand its reports, and take action. This need was also recognised early on in the development of Parfait and drop-in replacements for commonly used compilers at the time like `gcc`, `icc` and Oracle Solaris Studio were made available. A web application was also developed to facilitate report visualisation and classification. Over time, it was integrated with bug tracking systems to allow one-click creation of bug reports.

Achieving high precision and scalability on large code bases required careful thought and design. First, we realised that common assumptions made by static analysers at the time did not hold for large code bases. When dealing with industrial code bases, one cannot assume that: 1. The entire code base can be loaded in memory; 2. All dependencies can be compiled from source; and 3. All dependencies are available to the analysis (i.e., the closed-world assumption).

Parfait's analysis, thus, had to be *modular* and *summary-based* to analyse large programs in chunks, produce composable summaries, and ultimately report defects about the whole program. Second, because some dependencies might be unavailable, the analysis has to operate based on the assumption that certain program execution paths will be truncated (e.g., if the path goes through an unavailable dependency).

To address these challenges, Parfait's analyses are *bottom-up* and *on-demand* [SGSB05, SB06] starting from points to interest (POI) and working their way backward along all possible execution paths. This strategy allows Parfait to report all (partial) program paths reaching a POI, as opposed to a top-down analysis that might be unable to even reach a POI due to missing dependencies. On-demand analysis also gives developers the flexibility to choose the POI that are relevant in their context and avoid paying the cost of unwanted analyses. On-demand analysis, when combined with persistent summaries, also enables incremental analysis [KOAL19]. A developer can indeed limit the analysis to POI in her commit, knowing that Parfait will incrementally re-compute the summaries that need to be updated.

Figure 2 captures several of the latest features of Parfait. In it, `C.g` is the POI and the dashed arrows highlight the propagation of summaries in a bottom-up fashion. Yellow boxes represent modules, which can be analysed in an incremental manner. Finally, block arrows highlight the querying and updating of persistent summaries for a given system.

Since the inception of Parfait in 2007, the software development lifecycle has sped up quite significantly, integrated development environments (IDEs) have become more powerful, and developers are now expecting near instantaneous feedback. Local online analysis that highlights defects as one types is indeed becoming the norm, leaving nightly scans for global analyses that require more time and resources. To address these new expectations Parfait now leverages the Language Server Protocol (LSP) to perform advanced local static analysis in the background and in a variety of IDEs, to report defects as the code is being edited.

Parfait takeaways

- Parfait's analyses have been *designed* from the start for precision and scalability.
- While precision and scalability are essential, ease-of-use and timely reporting will ultimately drive the adoption of a static analysis tool by a wider audience.

While we have presented our experience starting around 2007 here, others have confirmed the above observations in their organisation [SvGJ+15].

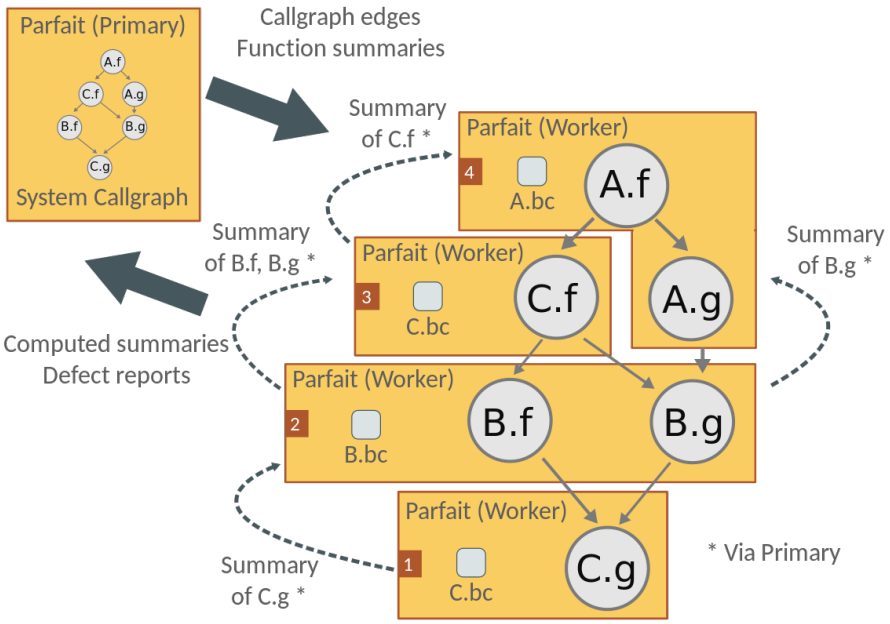

Figure 2: On-demand, bottom-up, modular, incremental and summary-based analysis in Parfait

Frappé As Parfait gained popularity and developers started consuming its defect reports, it became clear that flow-like traces did not always provide enough contextual information for triage and remediation. Complex defect traces often required investigation of upstream calls and variable usage to determine if they were true or false positive. Alternatively, some defects require fixes that can affect other downstream code. Figure 3 shows an example null pointer dereference report in OpenJDK where a dictionary might return `NULL` when a value is retrieved.

A developer reviewing this report and attempting a fix might want to investigate other uses of the dictionary, for example. Predicting which defect will require further investigation and what questions a developer will ask is, however, extremely difficult. It was decided at the time that this kind of code comprehension tasks were best left to an interactive tool that could be driven by developers themselves. The Frappé tool was born [HBC15].

Because code comprehension tasks often include elements of static analysis (e.g. What are the callers of this method? Where is this variable defined?), Frappé re-uses the same analysis infrastructure as Parfait to pre-compute various analyses (e.g. call graphs, def-use chains, and basic pointer and macro resolution), and adds graph-based querying and visualisation capabilities. To accommodate different types of tasks and developers, Frappé provides a web-based interface for interactive code exploration as well as Emacs and Vim integrations that provide a similar user experience but more advanced querying capabilities than commonly used tools like `ctags` and `cscope`. Over time, Frappé's capabilities evolved from single commit visualisation and querying to multi-commit navigation, allowing to diff and pinpoint faulty commits in a history. Figure 4 shows how the impact of commits on multiple files can be visualised. To support advanced queries, Frappé was also extended with a graph-based representation of code that can be interactively queried using PGQL [vRHK+16]. This

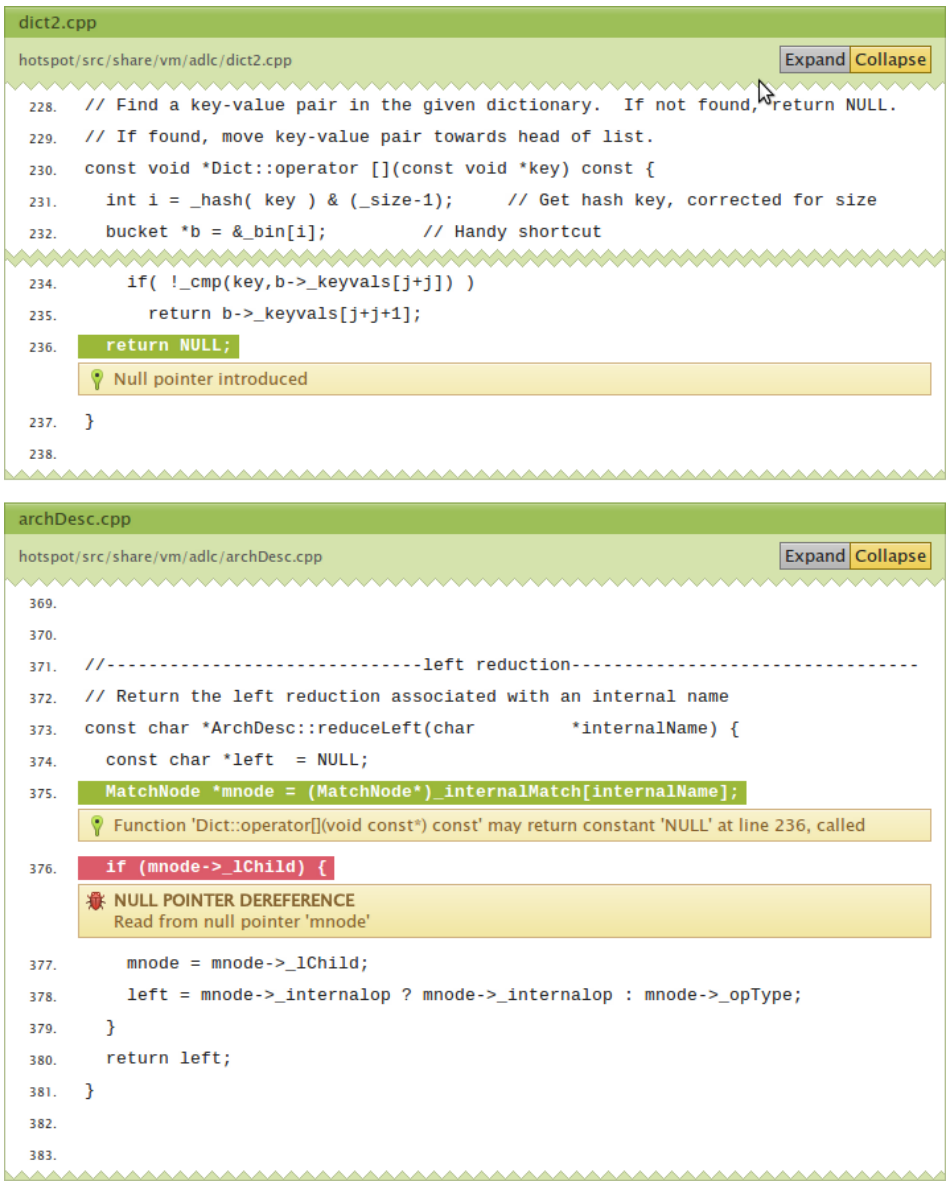

Figure 3: A source view of an OpenJDK defect report

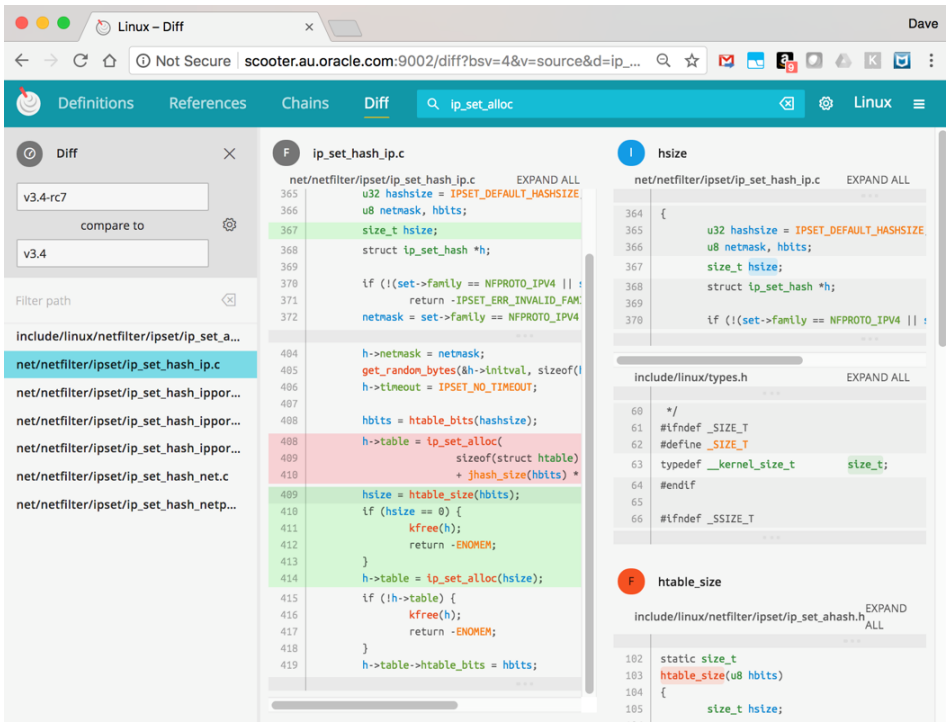

Figure 4: Frappé Example

is a generalisation of CodeQL [3] which focuses only on queries related to vulnerabilities.

A natural but unplanned use of Frappé was for change impact analysis and test case selection. In an industrial setting, it is often impractical, or even impossible to run an entire test suite after every code change, and selecting a subset of test cases that are relevant becomes crucial. Using the code querying features of Frappé, developers are now able to determine the functions that are potentially impacted by their changes, rank them, and select the test cases that should be run first.

Frappé takeaways

- Understanding and fixing complex bugs often involve understanding code beyond the bug trace.
- Code comprehension tools not only help understanding bug traces but also selecting the test cases to run following a fix.

Java Platform Around 2012-2013, several 0-day vulnerabilities were reported against the JDK, which allowed attackers to bypass the Java sandbox and completely take over the host machine. These defects, dubbed as caller-sensitive method vulnerabilities [CGK15], were highly specific to the JDK, with literally no tools available to automatically detect them. Following the initial attacks, there was a call to action at Oracle that sparked an effort to extend Parfait to also support the Java language and detection of JDK-specific vulnerabilities.

Given that Parfait operates on the LLVM bitcode intermediate representation, the first step was to translate stack-based Java bytecode to register-based LLVM bitcode. We perhaps naively assumed

[3] https://codeql.github.com/

that analyses for C code could then be ported with minor adaptations to support features in Java. But we had not anticipated the challenges ahead of us. Java features such as classes and virtual calls do not have straightforward mapping to C constructs and required significant extensions to Parfait. Furthermore, where efficient reasoning about arithmetic operations was key to achieve high precision in C programs (e.g., to reason about buffer overflows), precise resolution of virtual calls turned out to be key in Java programs.

Existing solutions such as Class Hierarchy Analysis (CHA) that conservatively resolve virtual calls based on the declared type of the caller turned out to be much too imprecise. Conversely, more precise techniques using points-to analysis were prohibitively expensive or even impossible to run on code bases like the JDK [SBL11]. Drawing from our experience scaling Parfait, our first step was to convert an existing whole-program points-to analysis (DOOP) [SB15] into a demand-driven one through slicing [ASK15]. Starting from a POI, our approach first produced a backward slice before launching the "whole-program" analysis on the slice only. Analyzing the JDK, which is a library, also presented unique challenges from a program analysis perspective. As we mentioned earlier, most static analysers assume that the entire program is available at analysis time (i.e., close-world assumption). While this is rarely the case for large code bases, this assumption never holds for libraries, which are meant to be called from arbitrary programs and operate on arbitrary objects (i.e. open-world assumption). To enable the evaluation of existing whole-program points-to analysis frameworks on library code, we designed an abstraction, called the most-general application (MGA), that acted as a stub for all possible programs calling into the library [AKS15]. This effectively over-approximates the call-graph within the JDK.

Java platform analysis takeaways

- While analysis frameworks can be made language-agnostic, analyses are often language-specific.
- Most static analysis implicitly operate on a closed-world assumption, which is unsuitable for the analysis of libraries or partial programs.

Java-based Web Applications Once basic support for Java analysis was in place, the logical next step was to extend Parfait to support security analysis of Java EE (now Jakarta EE) web applications, which account for a large proportion of enterprise applications. Compared to the JDK, Java EE adds several layers of abstractions over the core Java language that directly impact the control and data flows of applications and need to be accounted for by the analysis. An example is Java servlets, which are classes designed to handle requests (e.g. HTTP or others) and return a response. The mapping from a request to a servlet is typically handled by a servlet container that is also responsible for managing the life cycle of servlets, providing concurrent request processing, etc. From a program analysis perspective, analysing Java EE applications thus requires analysing the servlet container, which itself runs on top of a web server. Precisely analysing the complete application stack is not only prohibitively expensive, it is also beyond the capabilities of all state-of-the-art static analysers. This is a well known, but rarely acknowledged fact of static analysis: every popular programming language has constructs that cannot be precisely analysed [LSS+15]. It is also important to remember that developers program against APIs, not implementations. From that perspective, analysing the complete application stack is not only a waste of time, it is also detrimental to the understanding of static analysis reports. A good report should not conflict with developers' mental models and reports that span the whole application stack break that contract. For these reasons, it is common practice

to abstract the application stack into a model that allows for fast and precise partial program static analysis while being transparent to the analysis' consumers [SAP+11, ARF+14].

Because injection vulnerabilities are among the most common flaws in web applications [OWA], our main objective was to develop a taint analysis that could flag unsanitised attacker-controlled inputs flowing to security-sensitive operations. Our model thus needed to capture 1. entry points into the application; 2. sources of attacker-controlled inputs (i.e., taint sources); and 3. dynamically induced dependencies (e.g., dependency injections).

While our models were manually designed, automatic inference of models is an active area of research [AB16, HSC15].

With a solid model of the application container in place, we could tackle our next challenge: designing a static taint analysis for Java EE web applications. Drawing from our experience analysing the JDK, our requirements were pretty clear by that time: the analysis had to be bottom-up, and demand-driven. Unfortunately, the custom strategy we had deployed for the JDK did not meet our precision and scalability criteria for large-scale deployment to a wider audience, which led to the development of a "pure" bottom-up and on-demand analysis that was better suited for partial program analysis. Inspired by the Boomerang approach [SNQDAB16] that uses access paths to resolve points-to relations instead of whole-program heap modelling, we implemented a backwards-only access path-based taint analysis on top of the IFDS framework [RHS95]. Trading off the soundness of Boomerang that alternatively performs backward and forward analyses to resolve field accesses for an unsound backwards-only analysis was necessary for Parfait to meet its scalability and precision requirements.

Java Web analysis takeaways

- Web frameworks use hard-to-analyse and highly dynamic constructs, and require modelling to enable analysis.
- Access paths aliases, contrary to whole-program heap modelling, enable scalable bottom-up and demand-driven analysis of Java EE web applications.

Python Support Parfait gained traction within Oracle, and eventually demand grew for it to support analysis of additional programming languages. A popular choice for web applications, besides Java, was Python. We therefore set about implementing support for analysis of Python. As with Java, the first step was translation to LLVM bitcode. Also as with Java, the resolution of dynamic calls proved critical to analysis, but in this case even more so, since in Python *all* function calls are effectively dynamic.

Python, as with other dynamic languages, raises significant challenges for static analysis [Mad15]. Precise type inference, necessary for dynamic call resolution, becomes significantly more important as compared to when analysing Java. Again we applied the principles we had used earlier: abstract away unnecessary detail to achieve reasonable precision while remaining scalable. Python classes, as in various other dynamic languages that have the notion of class, are malleable and their methods can be injected or modified at run time; however, in practice, this is done only in specific cases and Python programs can to a great degree be viewed as if they were written in a less dynamic subset.

By recognising class and method definition instructions within the intermediate representation (after translation from Python bytecode), Parfait is able to determine a class hierarchy along with the named methods belonging to each class, and by assuming that import statements create static aliases to other classes or modules, it can resolve enough references to bootstrap an enhanced type propagation pass that in turn allows resolution of dynamic calls. Once this is done, the IFDS-based taint analysis

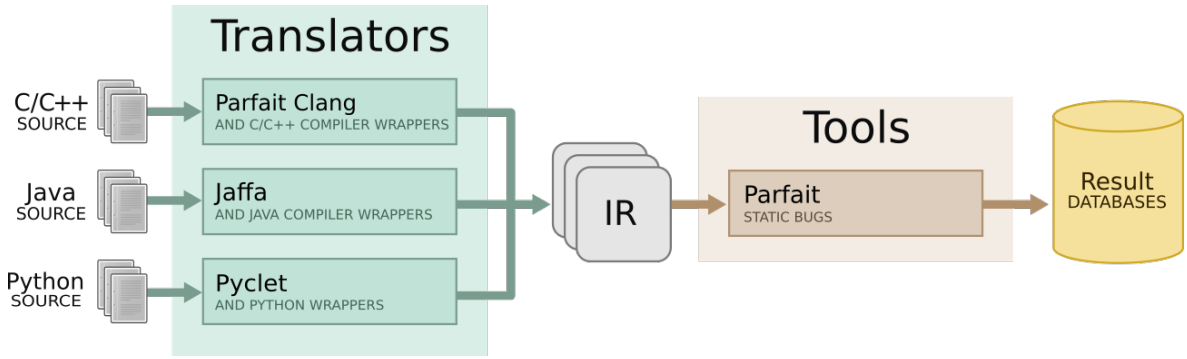

Figure 5: Parfait's Multi-language Architecture

developed for Java can be used for the analysis of Python code. Parfait's architecture which supports C, Java and Python is shown in Figure 5. Due to the assumptions made the analysis has imperfect precision, but it is *good enough* and remains scalabe. This is in keeping with the overall philosophy of practical and useful analysis in an industrial context.

Python analysis takeaways

- Relying on common design idioms trades off precision for scalability, but in practice may be sufficient
- Taint analysis demonstrates that static analyses can be reused across different languages.

2.2 Dynamic Analysis

Where Java EE applications were already stretching the limits of static analysis, the advent of languages like JavaScript and Python did impose a shift to dynamic analysis. Our experience with static JavaScript analysers [NHG19, JGHZ19] indeed confirmed that existing static analysis frameworks for JavaScript do not scale to large code bases yet with the desired level of precision, prompting us to explore other avenues.

Affogato Affogato [GHJ18] is an instrumentation-based dynamic analysis tool for Node.js and was our first attempt at industrial dynamic analysis. Because of the runtime overhead they impose [SKBG13, KTSS18], adoption of dynamic program analysis tools in industry has traditionally been low. With Affogato, our aim was to design a dynamic taint analysis tool that was suitable for test-time analysis (e.g. $< 2\times$ slowdown), where tests would provide the inputs and execution setup.

Our initial investigations very quickly revealed that "traditional" dynamic taint analysis, which requires heavy instrumentation to track the flow of tainted values through code and memory, would not only never meet our performance requirements, but were also causing so many crashes that they were practically unusable. First, we discovered that the slowdowns incurred by the analysis would often trigger race conditions or exceptions in code using constructs like `setTimeout` and `setInterval`. Second, we realised that the common dynamic analysis strategy of "transparently" proxying objects to intercept operations like field reads and writes often broke the semantics of programs relying on object identity. While it can be argued that these are bad programming practices, the fact is that they are common enough that they cannot be ignored. More recent work solves this issue by avoiding dynamic proxies altogether [KTSS18].

Hence, we turned our attention to an unsound, yet highly tunable and transparent alternative: taint inference [Sek09]. Taint inference uses string similarity metrics to infer the flows of values between instrumented program points. While conceptually simple, taint inference allows for non-intrusive instrumentation and is a perfect fit for Node.js (and JavaScript) programs where most values are strings and most objects can easily be converted to a JSON string and back. Taint inference can also easily be tuned for performance, precision and recall by: 1. Increasing or decreasing the number of instrumentation points in the program. 2. Varying similarity metrics and thresholds. For example, tuning Affogato to only instrument taint sources and sinks and only infer flows on exact matches would yield a high-performance, high-precision, low-recall analysis. Another benefit of taint inference was its ability to cope with external code. Indeed, most programming language virtual machines (e.g., Java, Python, JavaScript, and Ruby) include functionalities that are not implemented in the host language and thus outside of the scope of instrumentation-based dynamic analysis. While external code is traditionally handled through modelling, taint inference can infer flows simply by comparing inputs and outputs to external code. An obvious drawback is that taint inference is driven by heuristics than a fixed algorithm and offers very little in the way of formally reasoning about the underlying analysis. Nevertheless, we found that Affogato favourably compared to state-of-the-art approaches.

Affogato takeaways

- From an industrial usage perspective, analysis of languages with dynamic features such as JavaScript and Python, is not as mature as analysis for languages such as C and Java.
- Heavyweight dynamic analysis not only slows down applications, but often break them in unexpected ways.
- Taint inference is a practical, yet unsound way of dynamically tracking the flow of tainted values in JavaScript programs.

Gelato The main goal of Gelato is to detect client-side vulnerabilities, such as DOM-based XSS and OWASP Top 10 server-side issues without having access to the server-side code [HLK22]. Because dynamic analysis relies on inputs to execute, in Gelato we mainly focus on novel techniques for input generation in web applications. Modern web applications are extremely complex, and achieving decent code coverage requires one to address several challenges on the client side: browser interaction, form filling, static and dynamic link extraction, framework modelling, event generation, and state-aware crawling to name a few. Modern client-side JavaScript frameworks like React, Angular, Knockout, etc., further add to the complexity by introducing domain-specific languages (DSLs) that obfuscate control- and data-flows. From our experience, open-source security scanners haven't caught up to the latest advances in web application technologies yet, leading to wide variations in coverage between applications that uses different technology stacks.

Only after Gelato could automatically and reliably crawl a wide variety of web applications could we focus on client-side security analysis. For client-side analysis, we use a staged taint inference technique to reduce the number of false positives. In many aspects, Gelato attempts to emulate an attacker workflow. After a reconnaissance phase (e.g. crawling) that prioritises state-space exploration using a framework-aware static call graph, Gelato starts scanning the application for vulnerabilities by instrumenting client-side code and directing the execution towards sensitive statements. Then, to get past input guards and reach deeper into client-side code, Gelato collects and solves constraints on

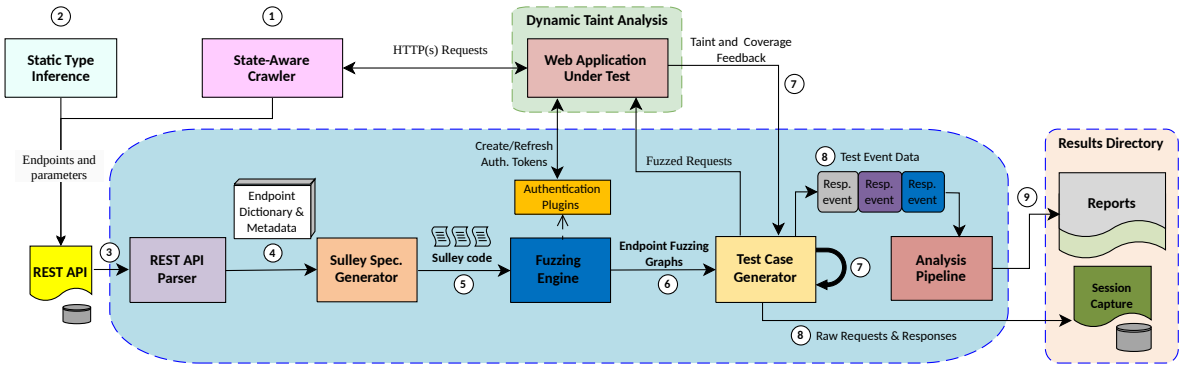

Figure 6: The BackREST web fuzzing architecture

the inputs. Finally, Gelato uses taint inference to determine if attacker-controlled inputs have reached sensitive code, in which case it reports a vulnerability.

Although Gelato is primarily focused on detecting client-side vulnerabilities, automated crawling also exercises, albeit indirectly, server-side code, making it an invaluable tool to infer server-side APIs. Indeed, by abstracting the HTTP requests of a crawling session into $endpoints \rightarrow parameters \rightarrow values$ mappings, one obtains a fairly accurate, albeit incomplete, API of the application under test. This inference component is used to determine if a revised version of the applications exposes any new endpoints. It is left to the developer to decide whether these new endpoints are deliberate or not.

Combining the API inference capabilities of Gelato with an API-driven fuzzer led to a fully automated web fuzzing solution. Further extending this solution with coverage and taint feedback loops from Affogato yielded BackREST [GHS+22], which detected several 0-days in popular NPM packages. Figure 6 illustrates the BackREST architecture, where components (1) and (7) correspond to Gelato and Affogato respectively. Other components are part of the fuzzer we extended.

Gelato takeaways

- Web application scanners cannot handle modern web application frameworks.
- Leveraging static and dynamic analysis to prioritise state-space exploration before targeting sensitive code is an efficient strategy to uncover vulnerabilities.
- Adding server-side feedback loops enhances vulnerability detection capabilities.

3 Lessons Learned

What have we learned over the course of more than 15 years developing industrial program analysis tools? First and foremost, in an industrial context, tool adoption is driven by developers, and not by program analysis, or security experts. This is similar to the findings reported by the Tricorder project [SvGJ+15]. Apart from defect reports, developers also need tools to understand and navigate through large codebases.

Failing to address the needs of *developers* is a guaranteed path to failure. While this might seem obvious, it is often tempting as a researcher to pursue the best possible *theoretical* solution while ignoring the *practical* constraints of your intended users. Computing resources are finite, developer time is precious and security is only one of the many aspects of software development. A security

analysis that is fast enough to fit in the development life cycle, with a low false positive rate and that can be adapted to evolving developers' needs thus has a much higher chance of success. As a consequence, much of our work has been focused on adapting academic research to the realities of industry and making the necessary compromises to drive tool adoption all the while delivering analyses that detected thousands of defects over the years and raised the bar of security at Oracle.

At a high level, our journey from static to dynamic to hybrid analysis was a natural consequence of our core requirements of delivering precise, scalable, and usable analyses for a variety of programming languages, frameworks, and applications. Of course, languages, frameworks and development practices are constantly evolving and the constraints we faced in our specific context at a specific point in time, might not hold anymore. For example, since our work on Affogato and Gelato, developers are embracing optional typing for various dynamic programming languages (e.g. TypeScript adds types to JavaScript, Python 3.5 added supports for type hints, PHP 8.0 added advanced typing support). Although type hints are generally non-binding, we believe that they could be leveraged to help improve the precision and scalabability of static analyses for dynamic languages. Further research, that goes beyond types, to provide good static abstractions is also required.

Another example includes the shift from monolithic to micro-service architectures and its consequence on analysis scalability. Micro-services, as their name suggests, are typically fairly small in size and more amenable to highly complex analyses. Where our challenge lied in analysing tightly-coupled but massive code bases, the next generation of industrial applications will require reasoning about massively distributed computations across small code bases.

At a more concrete level, the client analyses we developed (e.g. buffer overflow, use after free, caller-sensitive method, taint analysis) were unsurprisingly driven by the types of defects that were the most critical and prevalent at the time. However, we did discover along the way that the *support* analyses required (e.g. constraint solving, partial evaluation, points-to, string, and call graph analyses) are very much driven by programming languages and the way they influence programming practices. System programming languages like C forces reasoning about arithmetic constraints while object-oriented Java requires precise virtual call resolution. Finally, the concrete implementations of all these analyses (modular, bottom-up, demand-driven, and summary-based with access-path aliasing) were directly determined by our practical constraints of scalability, precision and usability.

As a research organisation, we are tasked with enabling developers in different product groups to use the tools we have developed. Despite our best efforts, transferring our research tools to production teams remains our greatest challenge. Beyond the human (e.g. advertising, training, etc.) and financial (e.g. funding new positions) aspects of tech transfer, managing the technical debt is a constant struggle [GJK+20]. Indeed, the process of research is one of trial-and-error that calls for quick prototyping in the early stages. As the project matures, however, one needs to gradually shift away from prototypes towards a minimum viable product (MVP) that can be on-boarded by a production team. Finding the optimal time to start a shift towards an MVP is, however, extremely challenging. Do it too soon and you are wasting research resources on engineering as an MVP that may never be adopted, do it too late and the cost of re-factoring your prototype increases exponentially, jeopardising the transfer. Identifying the inflection point where the shift from prototype to MVP needs to happen is still an open problem.

4 Our Vision: Intelligent Application Security

Up until recently, the different phases of software development (i.e. design, code, build, test, deployment, monitoring, etc.) were divided across dedicated and often separate teams. Our research and

development thus focused on delivering different tools to prevent security vulnerabilities from being introduced in products during the building (i.e., static analysis) and testing (i.e., dynamic analysis) phases of the software development life cycle.

With applications running on the cloud, the surface available to attackers has increased. Previous when an application was run on a machine that was not accessible to the external world, issues such as buffer-overflows were harder to exploit. Tools such as firewalls provided a level of security that is not sufficient for cloud-based applications. Cloud providers offer many APIs to manage the applications. At a high-level of abstraction, fixing security issues in the cloud is not different from fixing security issues on applications that run on-premises. That is, static and dynamic analysis tools remain important. However, these tools now need to handle identity and access management (IAM) which is one of the main security mechanisms for a cloud deployment. Tools also need to address infrastructure security. Not only are applications run on the cloud, developers are using cloud-services as part of their development practices. Hence all future tools need to be able to run as part of a service.

The definition of code is becoming broader. For instance, GitHub Actions in build systems and declarative descriptions of the infrastructure required for running the application are nothing but specific types of programs. These programs introduce different types of vulnerabilities that are not fully understood [GGTP19, LPSS21]. As researchers, we need to be vigilant and be able to develop analyses that can detect and prevent these new types of vulnerabilities.

In recent years, the wide adoption of DevOps practices has created exciting challenges and opportunities for program analysis tools. DevOps breaks the barriers between the different phases of software development, and increases velocity and automation. Nowadays, it is not uncommon for the same team to oversee its own software from coding to deployment, with monthly, weekly, or even daily release cycles. One simply cannot expect teams to inboard a myriad of different tools and integrate them all in their environment anymore. The onus is now on tool developers to provide the necessary integrations to enable the use of their tools in highly automated DevOps environments. While this increases the engineering overhead for tool developers, we believe it also creates interesting research opportunities. Where tools were previously worked independently, they will now be able to share information and complement each other. For example, a static analysis could forward uncertain vulnerability reports for a dynamic analysis to confirm at test time and avoid having developers review false positives.

Similarly, after vulnerable dependencies are discovered through software composition analysis, a runtime monitor could help developers prioritise fixes based on the vulnerable codes their application executes. Suggesting code patches for reported defects is yet another example of inter-tool collaboration that is enabled by DevOps. Combining tools into a single security infrastructure and enabling such sharing of information is what we call the Intelligent Application Security (IAS) and it is our vision for the future. Where DevSecOps promotes seamless and automated security throughout the software development life cycle, we propose IAS as way to achieve this goal.

The DevSecOps software lifecycle is depicted in Figure 7, which is extracted from [CIO19]. Each grey shield positions an existing software security component in the DevOps lifecycle. On the left-hand side of the diagram, next to the build and test phases, are SAST (Static Application Security Testing) and DAST (Dynamic Application Security Testing), our two traditional areas of expertise. Our latest IDE extensions for Parfait bridge the gap from SAST to securing coding, and a tool like BackREST blurs the boundary between SAST, DAST, and penetration testing. And this is what IAS is all about: unleashing the full potential of DevSecOps through seamless integration *and* collaboration between traditionally separate and independent security tools.

Achieving IAS will be challenging. IAS will only come to fruition only when all of the following issues are addressed.

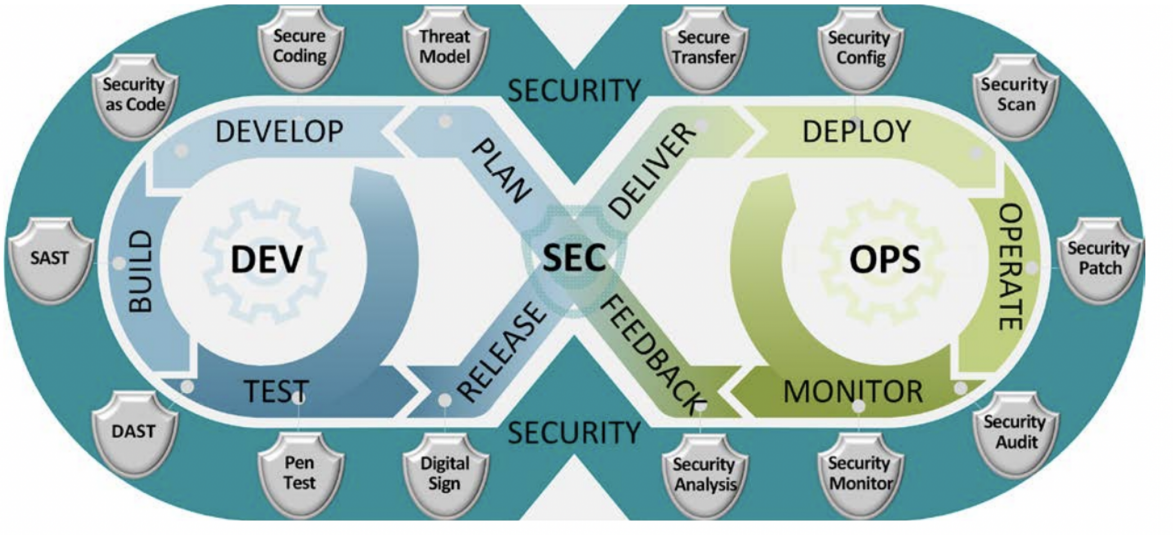

Figure 7: DevSecOps Software Lifecycle

1. Tool collaboration can only be achieved through common abstractions. This is an area where program analysis is crucially lacking. Many tools indeed use abstractions that were crafted for specific client analyses, but fall short when applied to a different problem. A good example of this is separation logic, which is very much limited to reasoning about certain classes of memory issues. Also separation logic, while very precise, does not scale to large code bases. That is why Infer[4] allows the developer to replace separation logic with abstract interpretation for detecting certain types of defects.

   Enabling collaboration of tools operating in the same domain (e.g. SAST) is only one aspect. Developing abstractions that enable cross-domain tool collaboration is vital. The question of the right abstraction that can be used in the develop, build, test, deploy and monitor, and patch generation phases would need to be answered.

2. DevOps has increased the velocity of the software development lifecycle. Although changes in development practices (e.g., shifting from monolith to micro-service architecture) will require adaptations to existing strategies, scalability and the ability to analyse sub-systems of a large system will remain a major requirement to drive tool adoption.

3. DevOps encourages hyper-automation. Through IAS, we aim to automate tasks such as vulnerable dependency detection and update, back-porting of security patches, detection, confirmation and repair of defects [GPR19] at the configuration and code levels, hardening of applications, and online attack detection and mitigation.

Of course, achieving this level of automation will require years of research and development, but it is very clear already that it will require highly precise program analyses. Removing humans from the loop virtually reduces the margin for error to zero, meaning that tools will need to triage defects and patches autonomously to separate those that are suitable for automated processing from those that require human attention.

[4] https://fbinfer.com/

DevSecOps empowers individual teams to manage their whole secure software development lifecycle. Where security used to be the realm of a few security experts overseeing many projects, DevSecOps encourages every member of a team to contribute to software security. As a consequence, the user base for security tools will diversify, their level of expertise will vary greatly and usability will remain critical. This will be especially challenging for IAS, which will increase complexity by operating across the entire DevSecOps loop.

We are currently working on tools related to supply chain security including software composition analysis and analysis of build infrastructures, exploring the use of ideas from program synthesis to generate security protections that can be deployed at run-time [VGB+22] and malware detection [TCC+21]. Our vision is to integrate the work we have reported here as well our current work under a single offering called IAS.

References

[AB16] Steven Arzt and Eric Bodden. Stubdroid: Automatic inference of precise data-flow summaries for the android framework. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 725–735. IEEE, 2016.

[AKS15] Nicholas Allen, Padmanabhan Krishnan, and Bernhard Scholz. Combining type-analysis with points-to analysis for analyzing Java library source-code. In *Proceedings of the SOAP Workshop*, pages 13–18. ACM, 2015.

[ARF+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[ASK15] Nicholas Allen, Bernhard Scholz, and Padmanabhan Krishnan. Staged points-to analysis for large code bases. In B. Franke, editor, *Compiler Construction*, LNCS 9031, pages 131–150, 2015.

[BBC+10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communication of the ACM*, 53(2):66–75, 2010.

[CGK15] Cristina Cifuentes, Andrew Gross, and Nathan Keynes. Understanding caller-sensitive method vulnerabilities: A class of access control vulnerabilities in the java platform. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State of the Art in Program Analysis*, pages 7–12, 2015.

[CIO19] U.S. Department of Defense Chief Information Officer. Dod enterprise devsecops reference design version 1.0. `https://dodcio.defense.gov/Portals/0/Documents/DoD%20Enterprise%20DevSecOps%20Reference%20Design%20v1.0_Public%20Release.pdf`, August 2019.

[CKL+12] Cristina Cifuentes, Nathan Keynes, Lian Li, Nathan Hawes, and Manuel Valdiviezo. Transitioning Parfait into a development tool. *IEEE Security and Privacy*, 10(3):16–23, May/June 2012.

[EGLPAMF20] Tiago Espinha Gasiba, Ulrike Lechner, Maria Pinto-Albuquerque, and Daniel Mendez Fernandez. Awareness of secure coding guidelines in the industry - a first data analysis. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 345–352, 2020.

[GGTP19] MIchele Guerriero, Martin Garriga, Damian A. Tamburri, and Fabio Palomba. Adoption, support, and challenges of infrastructure-as-code: Insights from industry. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 580–589, 2019.

[GHJ18] François Gauthier, Behnaz Hassanshahi, and Alexander Jordan. AFFOGATO: runtime detection of injection attacks for node.js. In *Companion Proceedings for the ISSTA/ECOOP Workshops*, pages 94–99. ACM, 2018.

[GHS+22] François Gauthier, Behnaz Hassanshahi, Benjamin Selwyn-Smith, Trong Nhan Mai, Max Schlüter, and Micah Williams. Experience: Model-Based, Feedback-Driven, Greybox Web Fuzzing with BACKREST. In *European Conference on Object-Oriented Programming, ECOOP*, volume 222 of *LIPIcs*, pages 29:1–29:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[GJK+20] François Gauthier, Alexander Jordan, Padmanabhan Krishnan, Behnaz Hassanshahi, Jörn Guy Süß, Sora Bae, and Hyunjun Lee. Trade-offs in managing risk and technical debt in industrial research labs: an experience report. In *Proceedings of the 3rd International Conference on Technical Debt*, pages 98–102, 2020.

[GPR19] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.

[HBC15] Nathan Hawes, Ben Barham, and Cristina Cifuentes. Frappé: Querying the linux kernel dependency graph. In *Proceedings of the GRADES'15*, pages 1–6, 2015.

[HLK22] Behnaz Hassanshahi, Hyunjun Lee, and Padmanabhan Krishnan. Gelato: Feedback-driven and guided security analysis of client-side web applications. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 618–629. IEEE, 2022.

[HSC15] Stefan Heule, Manu Sridharan, and Satish Chandra. Mimic: Computing models for opaque code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 710–720, 2015.

[JGHZ19] Alexander Jordan, François Gauthier, Behnaz Hassanshahi, and David Zhao. Unacceptable behavior: Robust pdf malware detection using abstract interpretation. In *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, pages 19–30, 2019.

[JR04] Russell L. Jones and Abhinav Rastogi. Secure coding: Building security into the software development life cycle. *Information Systems Security*, 13(5):29–39, 2004.

[KOAL19] Padmanabhan Krishnan, Rebecca O'Donoghue, Nicholas Allen, and Yi Lu. Commit-time incremental analysis. In *International Workshop on State Of the Art in Program Analysis,SOAP*, pages 26–31. ACM, 2019.

[KTSS18] Rezwana Karim, Frank Tip, Alena Sochurková, and Koushik Sen. Platform-independent dynamic taint analysis for javascript. *IEEE Transactions on Software Engineering*, 46(12):1364–1379, 2018.

[LPSS21] Julien Lepiller, Ruzica Piskac, Martin Sch"af, and Mark Santolucito. Analyzing infrastructure as code to prevent intra-update sniping vulnerabilities. In *TACAS*, number 12652 in LNCS, pages 105–123. Springer, 2021.

[LSS+15] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[Mad15] Magnus Madsen. *Static Analysis of Dynamic Languages*. PhD thesis, Aarhus University, 2015.

[MRS10] Bill McCloskey, Thomas Reps, and Mooly Sagiv. Statically inferring complex heap, array, and numeric invariants. In *Static Analysis*, pages 71–99. Springer, 2010.

[NHG19] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. Nodest: Feedback-driven static analysis of Node.js applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 455–465, 2019.

[OWA] OWASP. OWASP Top Ten. `https://owasp.org/www-project-top-ten/`. [Online; accessed 08-November-2022].

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.

[SAP+11] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4F: Taint analysis of framework-based web applications. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 1053–1068, 2011.

[SB06] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 387–400. ACM, 2006.

[SB15] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, April 2015.

[SBL11] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 17–30. ACM, 2011.

[Sek09] R Sekar. An efficient black-box technique for defeating web application attacks. In *NDSS*, 2009.

[SGSB05] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodik. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 59–76. ACM, 2005.

[SKBG13] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 488–498, 2013.

[SNQDAB16] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[SvGJ+15] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *International Conference on Software Engineering (ICSE)*, pages 598–608. IEEE Press, 2015.

[TCC+21] Haoxi Tan, Mahin Chandramohan, Cristina Cifuentes, Guangdong Bai, and Ryan K. L. Ko. ColdPress: An extensible malware analysis platform for threat intelligence. arXiv:2103:07012, 2021.

[VGB+22] Kostyantyn Vorobyov, François Gauthier, Sora Bae, Padmanabhan Krishnan, and Rebecca O'Donoghue. Synthesis of java deserialisation filters from examples. In *Computers, Software, and Applications Conference (COMPSAC)*, pages 736–745. IEEE, 2022.

[vRHK+16] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. PGQL: A property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, pages 1–6, 2016.