The 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 a variety of tools that detect security vulnerabilities in an industrial setting. The main driving forces for adoption of program analysis tools by a development organisation are low false positive rate, ease of integration in the developer's workflow, scalability to handle industrial size systems and results that are easy to understand. Even if one the above dimensions is not supported, the tool will not be used in practice.

We show how the analyses of program analysis tools have changed over more than a decade due to differences in languages, e.g., code written in systems-level languages like C tend to focus on memory-related vulnerabilities, in contrast to languages like Java, JavaScript and Python where the focus is more on injection vulnerabilities in web or cloud applications. Based on language, static or dynamic analysis approaches are needed, including hybrid approaches.

We conclude with our vision on Intelligent Application Security – how program analysis tools will keep changing to enable the DevSecOps model given the fertile ground that the DevOps model provides today. We foresee different program analysis tools working together by sharing information, including the results they produce, while addressing newer security issues such as those related to supply chain issues. In this way, program analysis tools would be extended with relevant machine learning techniques and be integrated in all different phases of the code development, building, testing, deployment and monitoring cycle.

Keywords: static Analysis, dynamic Analysis, Industrial Scale Application, DevOps

1 Introduction

Security of applications is a continuously changing domain. However, from a 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, program analysis tools have evolved over the years from detecting general software bugs to detecting security-related defects.

As software systems have evolved, the types of security vulnerabilities have also evolved. Thus tools developed for a specific class of applications cannot always be used for another class of applications. For instance, those that work on system code are not useful for web-based applications. This imposes additional burden on the tool developers to keep them relevant. If these tools are not easy to integrate with a developer's workflow, the developer has to master a number of them individually. It is imperative that the analysis tool developers and the application developers 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 application developers to provide such information is not practical. Similarly asking a majority of application developers to use tools such as Semmle [HVdM06, VHDM07] is not realistic. Such tools help security researchers to explore security-related issues; most developers however are not security experts and 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.

Because most developers involved in developing large complex applications are skilled, the defects that are present in their code are because of the complex interactions between different components. Hence, program analysis tools that are more advanced than simple linters and scanners are required. However, more advanced tools impose a high cognitive load on the developers. The developers need to understand how to use the tools, understand the reports they produce, and find a solution to fix any indicated vulnerability without breaking changes. Thus, overall, our tools focus on detecting well defined security vulnerabilities and presenting results that can be understood by application developers rather than security researchers.

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 security analysis tools will have an impact on the software development landscape that continues to change rapidly.

2 Evolution of Application Domain and Techniques

Static analysis has a rich history starting with applications including type inference, compiler optimisations (without looking at the exact functionality), simple error checking (e.g., `lint`) and sophisticated correctness checks [Tho21]. In the context of type inference and compiler optimisations, the techniques had to be sound as any unsoundness can alter the semantics of the original program. However, tools like `lint` which were related to finding errors, could have some imprecision. Abstract interpretation [CC77] was the first technique to define a framework for different abstractions of program executions. This technique was extended with other techniques to analyse programs without executing them. Depending on the available resources, one could use program analysis techniques to address compiler optimisations as well as bug-detection in realistic programs.

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$^+$10] where the focus was not on soundness but finding bugs that matter on large codebases. While some tools did focus on verification of key-properties [FHRS08], they were usually in the domain of relatively small embedded systems. As noted by the Soundy Manifesto [LSS$^+$15], many of the tools made assumptions on what language features were supported (e.g., MISRA-C[1] for the automotive domain). Such assumptions were not valid for some of the codebases of interest to us which included the Solaris operating system. Given the size of the codebases, techniques that helped the developers understand the code were also needed.

In the early 2000s, however, Java[2] 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, where an attacker alters the program logic by injecting malicious inputs, taint analysis had to be developed. Although such a taint analysis is developed in the context of Java, it needs to be usable for C programs, to determine which buffer overflow defects could be exploited by an external attacker. Because of various security vulnerabilities reported, the techniques had to address both the JDK [3] (the Java Platform) and Java-based applications.

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.

In Figure 1, Parfait Systems, Parfait Platform and Parfait Web-Applications are part of what is called SAST (static application security testing), Frappé is a tool that uses some of the underlying techniques in the SAST tools, and tools such as Affogato and Gelato are part of DAST (dynamic application security testing).

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, we

[1]https://www.misra.org.uk/

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

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

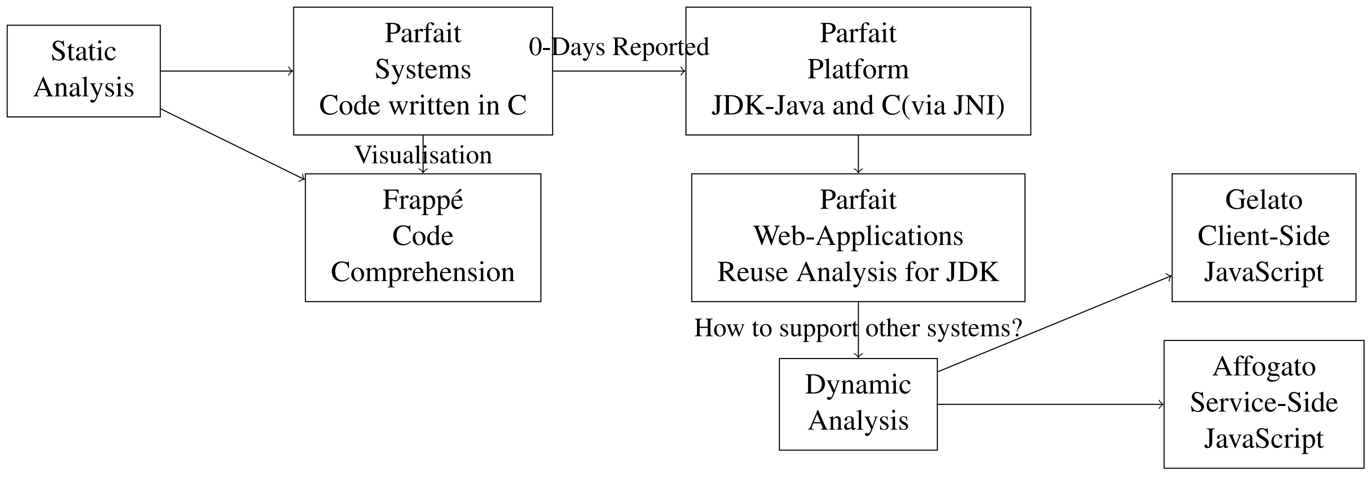

Static Analysis
Parfait Systems Code written in C
0-Days Reported
Parfait Platform JDK-Java and C(via JNI)
Visualisation
Frappé Code Comprehension
Parfait Web-Applications Reuse Analysis for JDK
Gelato Client-Side JavaScript
How to support other systems?
Dynamic Analysis
Affogato Service-Side JavaScript

Figure 1: Summary of Evolution

collected data from a variety of development organisations on their experiences with the use of commercial static analysis tools. It must be emphasised that owing to licensing requirements, we ourselves did not run the various tools used by the different groups. The aim was to focus the research on challenges faced by these development organisations on the use of static analysis tools. The collected data concluded that all the tools were suffering from the following three major flaws: 1. They produced large amounts (e.g. 30%-80%) of false positives; 2. They took days or weeks to analyse large (e.g. million lines of code) 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 highly scalable 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%. The above requirements are derived from the experience of the product groups who were using a variety of commercial tools. They are also derived from the desire to run Parfait as part of the nightly builds. During such nightly builds, the entire system is built and tested, leaving a small window to run Parfait. While over time, hardware became faster, the size of systems being analysed also increased. Hence, in our experience, the time requirements on our tools did not really change.

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. Furthermore, 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 their 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 (with general details such as value flows between function parameters, as well as bug-type specific information such as whether the function returns a pointer to a heap allocation) 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.

Generation of an intermodule callgraph is required to produce a bottom-up module ordering for analysis. For C and C++, this is restricted to processing of static calls; when processing an individual module, a refined intra-module callgraph is generated. For Java (where the analysis had a different focus, which will be discussed in more detail shortly) some processing of dynamic calls in the intermodule callgraph proved necessary, though even in this case approximations are used to maintain scalability (a points-to analysis is not performed, even for the intra-module callgraph, as it was found to be too expensive).

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 analyses 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.

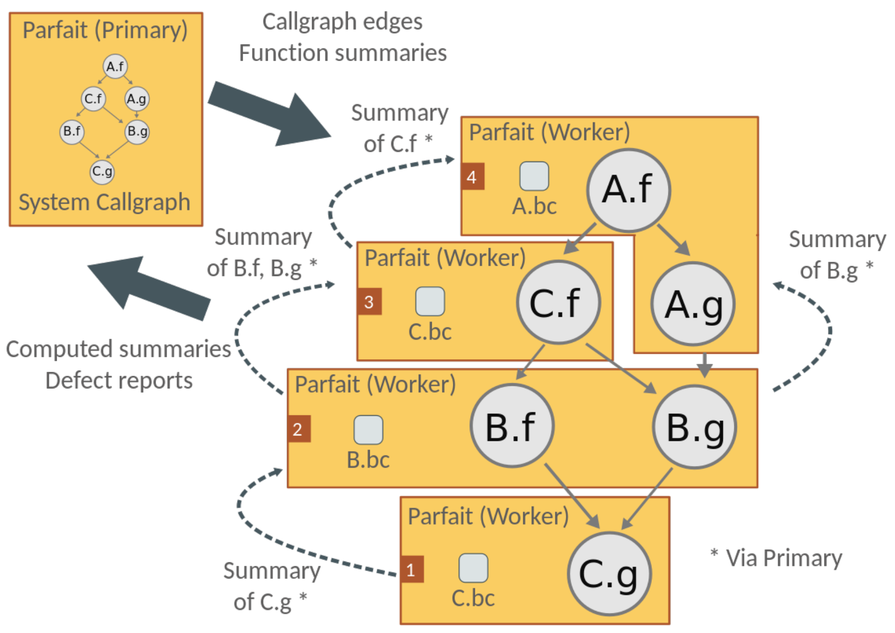


Figure 2: On-demand, bottom-up, modular, incremental and summary-based analysis in Parfait. Yellow boxes represent modules; "X.y" represents a function *y* within module *X*. Solid arrows represent the distribution of summaries by the primary process to workers processing modules, and the return of summaries generated by the worker; dotted arrows represent the effective transfer of summaries between modules, from callee to caller.

- 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.
- Since 2007, these observations have also been confirmed by other practitioners [SvGJ+15].

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. The key points to note are that on line 375 in `archDesc.cpp` can call the function `operator` on line 230 of `dict2.cpp`. As the function `operator` can return `NULL`, the deference on line 376 can result in a potential null pointer exception.

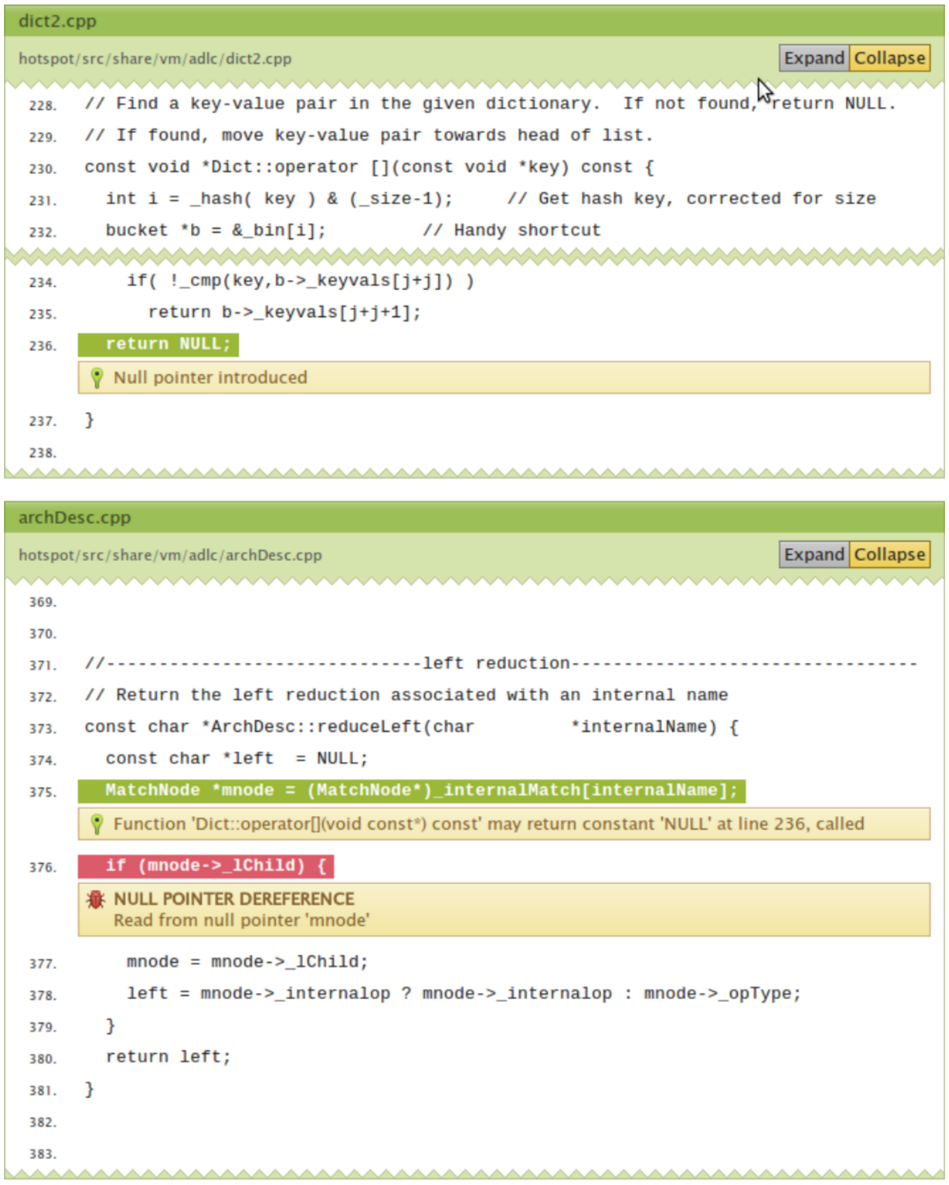
A developer reviewing this report and attempting a fix might want to investigate other uses of the dictionary, for example. Predicting which defect will require further investigation and what questions a developer will ask is, however, extremely difficult. It was decided at the time that this kind of 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. As shown on the left-side of the figure, the user has requested to see the difference between version `v3.4-rc7` and `v3.4`. From the list the files, some of the changes in `ip_set_hash_ip.c` are shown. Frappé also shows some of the details from the included files. For instance, the call to `htable_size` on line 409 is the reason the code-snippet in `ip_set_ahash.h` is displayed.

To support advanced queries, Frappé was also extended with a graph-based representation of code that can be interactively queried using PGQL [vRHK+16], which is a generalisation of CodeQL [4]. The standard queries available as part of CodeQL are not sufficient for our use cases. Hence the user has to write their own rules in a Datalog-like language. Because most of the developers are not well versed in analysis-related techniques, a graph-based interactive querying framework was essential.

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.

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



dict2.cpp

hotspot/src/share/vm/adlc/dict2.cpp

Expand Collapse

```
// Find a key-value pair in the given dictionary.  If not found, return NULL.
// If found, move key-value pair towards head of list.
const void *Dict::operator [](const void *key) const {
  int i = _hash( key ) & (_size-1);     // Get hash key, corrected for size
  bucket *b = &_bin[i];         // Handy shortcut
```

```
    if( !_cmp(key,b->_keyvals[j+j]) )
      return b->_keyvals[j+j+1];
  return NULL;
```

Null pointer introduced

```
}
```

archDesc.cpp

hotspot/src/share/vm/adlc/archDesc.cpp

Expand Collapse

```


//------------------------------left reduction---------------------------------
// Return the left reduction associated with an internal name
const char *ArchDesc::reduceLeft(char         *internalName) {
  const char *left  = NULL;
  MatchNode *mnode = (MatchNode*)_internalMatch[internalName];
```

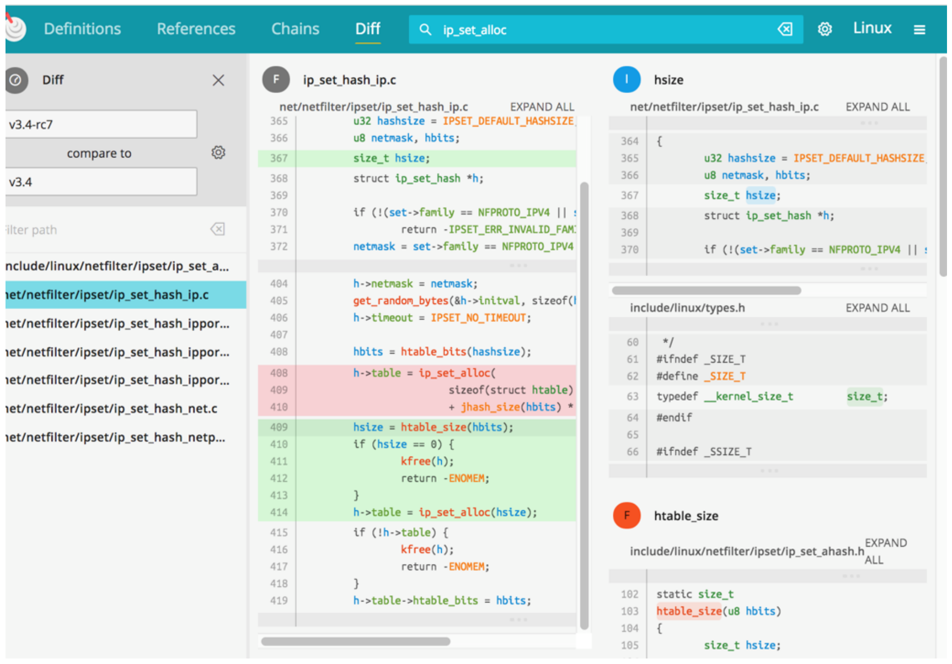
Function 'Dict::operator[](void const*) const' may return constant 'NULL' at line 236, called

```
  if (mnode->_lChild) {
```

NULL POINTER DEREFERENCE
Read from null pointer 'mnode'

```
    mnode = mnode->_lChild;
    left = mnode->_internalop ? mnode->_internalop : mnode->_opType;
  }
  return left;
}
```

Figure 3: A source view of an OpenJDK defect report


Definitions
References
Chains
Diff
ip_set_alloc
Linux
Diff
v3.4-rc7
compare to
v3.4
ilter path
nclude/linux/netfilter/ipset/ip_set_a...
et/netfilter/ipset/ip_set_hash_ip.c
et/netfilter/ipset/ip_set_hash_ippor...
et/netfilter/ipset/ip_set_hash_ippor...
et/netfilter/ipset/ip_set_hash_ippor...
et/netfilter/ipset/ip_set_hash_net.c
et/netfilter/ipset/ip_set_hash_netp...
F
ip_set_hash_ip.c
EXPAND ALL
I
hsize
EXPAND ALL
EXPAND ALL
F
htable_size
EXPAND ALL

net/netfilter/ipset/ip_set_hash_ip.c

```
        u32 hashsize = IPSET_DEFAULT_HASHSIZE
        u8 netmask, hbits;
        size_t hsize;
        struct ip_set_hash *h;

        if (!(set->family == NFPROTO_IPV4 || s
                return -IPSET_ERR_INVALID_FAM
        netmask = set->family == NFPROTO_IPV4

        h->netmask = netmask;
        get_random_bytes(&h->initval, sizeof(
        h->timeout = IPSET_NO_TIMEOUT;

        hbits = htable_bits(hashsize);
        h->table = ip_set_alloc(
                        sizeof(struct htable)
                        + jhash_size(hbits) *
        hsize = htable_size(hbits);
        if (hsize == 0) {
                kfree(h);
                return -ENOMEM;
        }
        h->table = ip_set_alloc(hsize);
        if (!h->table) {
                kfree(h);
                return -ENOMEM;
        }
        h->table->htable_bits = hbits;
```

net/netfilter/ipset/ip_set_hash_ip.c

```
{
        u32 hashsize = IPSET_DEFAULT_HASHSIZE
        u8 netmask, hbits;
        size_t hsize;
        struct ip_set_hash *h;

        if (!(set->family == NFPROTO_IPV4 || s
```

include/linux/types.h

```
 */
#ifndef _SIZE_T
#define _SIZE_T
typedef __kernel_size_t         size_t;
#endif

#ifndef _SSIZE_T
```

include/linux/netfilter/ipset/ip_set_ahash.h

```
static size_t
htable_size(u8 hbits)
{
        size_t hsize;
```

Figure 4: Frappé Example

Frappé takeaways

- 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 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. Analysing the JDK, which is a library, also presented unique challenges from a program analysis perspective. Libraries can be called from arbitrary programs and operate on arbitrary objects which are external to the analysis (i.e. open-world assumption). In application-centric static analysis, models for external code (e.g. libraries) are often derived manually or automatically, to effectively enable closed-world analysis of partial (e.g. application-only) programs. Nowadays, static analysers even include pre-built models of commonly used libraries and offer tooling for users to define their own code models. In library-centric static analysis, however, it is not straightforward to pre-build models of "common" applications. Thus, 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]. The key realisation behind MGA is that the state of objects allocated in libraries, which can be modelled precisely using heap abstractions, will often be influenced by objects that were allocated in external (e.g. application) code, which can only be modelled imprecisely using type information. By allowing both representations to co-exist and containing the imprecision of type-based models, MGA over-approximates the call-graph within the JDK more precisely than a pure type-based analysis.

Java platform analysis takeaways

- While analysis frameworks can be made language-agnostic, analyses are often language-specific.
- Whole-program analysis of libraries requires modelling of *all* possible applications.

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]. Furthermore, while our models do not handle concurrency explicitly, they abstract each "request" as triggering a specific trace of the system, which may interact (e.g., via our data-flow analysis) with other requests. Properties such as race-conditions detection are out of scope, however.

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 large C codebases and the JDK, our requirements were pretty clear by that time: the analysis had to be bottom-up and demand-driven. Unfortunately, achieving on-demand analysis through slicing, as we did for the JDK, proved to be too wasteful for large-scale application analysis due to the high number of overlapping, yet independently processed slices. Furthermore, the heap-based analysis we used for the JDK required all object allocation sites to either be analysed or modelled to establish aliasing relations. While we could circumvent this issue in JDK analysis by using our simple MGA abstraction, the modelling costs to support the analysis of industrial Java EE applications and all their libraries were deemed too high at the time (novel abstractions for Java EE have been proposed since then [AFK$^+$20]). This led to the development of a "pure" bottom-up and on-demand analysis that was better suited for partial program analysis of Java EE applications.

Inspired by the Boomerang approach [SNQDAB16] that uses access paths [LR92] to resolve points-to relations in a modular fashion instead of using whole-program heap modelling, we implemented a bottom-up access path-based taint analysis on top of the IFDS framework [RHS95]. Where heap-based points-to analyses alias two variables if they point to the same object on the abstract heap, access-path analyses instead represent heap objects by how they are accessed from an initial variable. As soon as two variables share an identical access path, they are aliased, which can be achieved in a modular fashion and without the need for allocation sites. However, trading off the soundness of Boomerang, which alternates between top-down and bottom-up analyses, for an unsound but purely bottom-up 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 web applications written in 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,

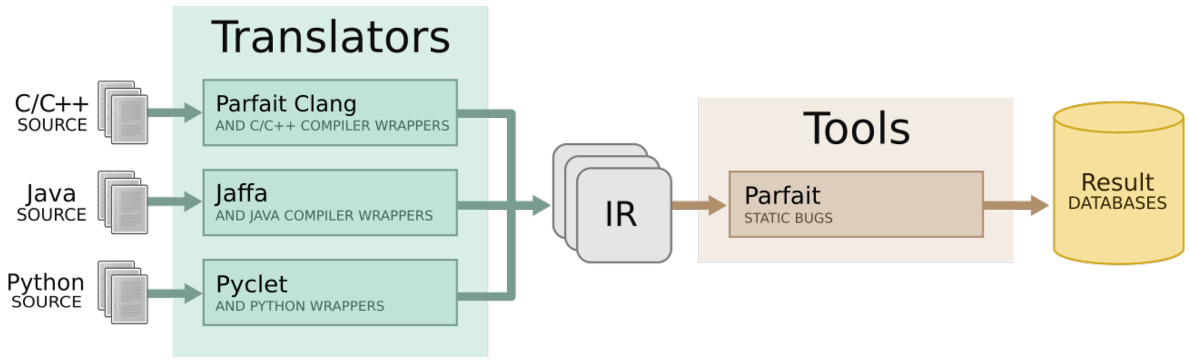


Figure 5: Parfait's Multi-language Architecture

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. Web applications written in Python use a variety of frameworks. We have provided manual specifications (e.g., sources of taint) of frameworks that are used within our organisation.

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. The resulting callgraph may not be perfect; particular limitations include inability to process the effects of dynamic modification of object and classes, such as insertion of or replacement of methods, and to recognise function or method resolution via dynamic techniques (such as when looking up a function by name via a string, rather than calling it directly). Dynamic code execution via the *eval* function is also unsupported. For application code which makes minimal use of such idioms, however, the callgraph is sufficient to allow meaningful analysis, especially given the design goals of favouring production of some useful bug reports over performing a sound analysis.

Once a callgraph has been generated, the IFDS-based taint analysis 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 scalable, in keeping with the overall philosophy of practical and useful analysis in an industrial context. While the notion of "good enough" is subjective, the quality of the results of our analysis has been accepted by the users of our tool.

Python analysis takeaways

- Relying on common design idioms, which do not use the rich variety of dynamic features available, trades off precision for scalability, but remains sufficiently precise in practice.
- 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, such as SAFE [LWJ$^+$12] and TAJS [JMT09] did not yet scale to large code bases with the desired level of performance and precision, prompting us to explore other avenues.

Affogato The un-typed and highly dynamic nature of JavaScript makes static analysis extremely challenging. For example, runtime code generation, dynamic object prototypes and properties, variadic functions, and functions as first class citizens are commonly cited features of the language that make static analysis difficult. Work by Richards et al. [RLBV10] highlights additional hard-to-analyse constructs and shows, through empirical evaluation, that they are too commonly used to be simply ignored. As a consequence, static analysis of JavaScript code is both highly complex and imprecise. For example, constructing a precise static call graph for Node.js applications is still a challenge [NTM21].

Affogato [GHJ18] was our first attempt at industrial-scale dynamic analysis. Affogato is an instrumentation-based dynamic analysis tool for Node.js. Because of their brittleness and the runtime overhead they impose, adoption of heavy-weight dynamic program analysis tools in industry has traditionally been low. To enable precise dynamic analysis, approaches such as [SKBG13, KTSS18] indeed install proxies around each object to intercept and analyse field reads and writes, as well as function calls, arguments, and return values. They similarly wrap primitive values such as number and strings to track operations on them.

Our initial investigations very quickly revealed that these "traditional" dynamic taint analyses, 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 and wrappers altogether [ATBT22].

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. Hence, we turned our attention to an unsound, yet highly tunable and transparent alternative: taint inference [Sek09]. Where dynamic taint analysis precisely tracks the flows of tainted values through the execution of a program, taint inference instead uses string similarity metrics to infer the flows of values between instrumented program points, hereafter called *watchpoints*. The art of taint inference thus boils down to striking a balance between performance and precision by instrumenting just enough watchpoints to infer correct flows without impacting performance too severely. Fortunately, taint analysis has a long history [CLO07, TPF$^+$09, SAP$^+$11, WR13, ARF$^+$14, KTSS18] and watchpoints of interest (e.g. reads of request parameters, sanitisation routines, and writes to files, databases, and HTTP responses) are well known and creating a basic taint inference analysis for a specific vulnerability can often be achieved by defining 3-5 watchpoints.

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 rather than a fixed algorithm. Hence it offers very little in the way of formal 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 use 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 the inputs. To keep the approach practical for real-world applications, path constraints are only collected for certain operations based on our experience from analysing JavaScript applications. To help get passed input guards, we first collect runtime values of interest, such as operands in conditional statements and arguments in string function calls (e.g., the string.substring built-in function). Then, we attempt to solve path constraints involving tainted (i.e. Gelato-controlled) values by replacing them with previously logged runtime values. 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 indirectly exercises 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, REST API of the application under test,

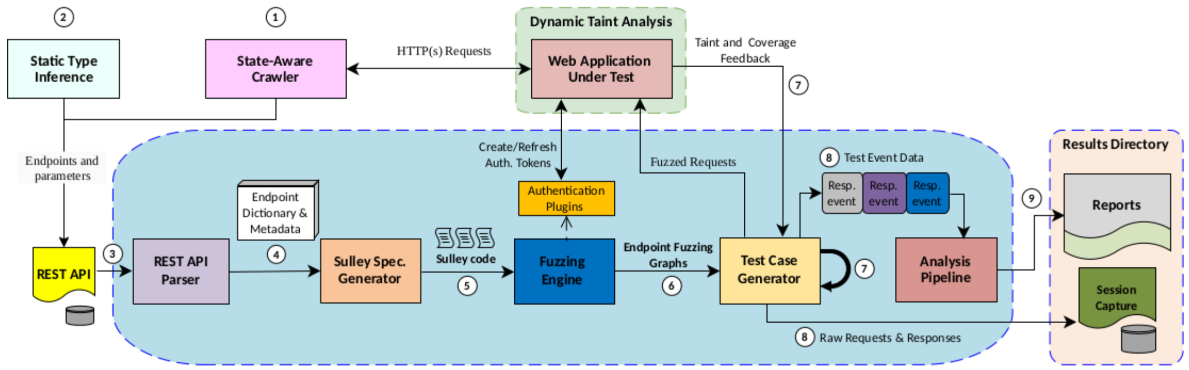


Figure 6: The BackREST web fuzzing architecture

where *endpoints* correspond to publicly accessible URLs, *parameters* list the HTTP request parameters associated to an *endpoint* and *values* list the values each *parameter* can take. This REST API of the application can then be presented to the developers as an easy-to-edit OpenAPI[5] (previously known as Swagger) specification. Developers can then iteratively help Gelato discover more APIs by providing additional endpoints, parameters, or values. Inferred specifications can also serve as a baseline to detect possibly undesired API changes after an application is updated. For example, given a new version of an application, developers can review newly exposed endpoints and decide whether they are deliberate or not.

BackREST The client-side of a web application is often the primary gateway to attack the server-side. Where Gelato and Affogato focused on the client- and server-side respectively, BackREST leverages these technologies to assess the security of web applications holistically. Figure 6 presents the architecture of BackREST [GHS$^+$22], a feedback-driven, REST-based, gray-box fuzzer that

1. Leverages Gelato ① and a static type inference analysis ② to automatically infer REST APIs.
2. Uses the inferred APIs ③ to seed an Oracle-internal REST-based fuzzer (RESTFuzz) ④, ⑤, ⑥, ⑧, ⑨.
3. Leverages Affogato's dynamic taint inference to focus the fuzzing session on endpoints and parameters that triggered tainted flows ⑦.

Only by exploiting the synergy between Gelato, Affogato, and a RESTFuzz could BackREST detect several 0-days in popular NPM packages that were missed by state-of-art scanners. In essence, Gelato provides sufficient seeds for the fuzzer to get deep into applications whereas Affogato provides sufficient feedback to detect attacks that would otherwise have gone unnoticed.

Gelato and BackREST 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.

[5]https://swagger.io/specification/

- The synergy between Gelato, Affogato and RESTFuzz allowed BackREST to uncover vulnerabilities that were missed by other scanners.

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 [SAE$^+$18], although there are key differences in overall approach; we have opted to perform sophisticated analyses, using techniques such as incremental analysis to reduce analysis times and better allow integration with regular builds, and successfully achieved low false-positive counts via careful design and implementation of analysis strategies, whereas Tricorder eschews sophisticated analysis citing large investment requirements and significant work necessary to reduce false-positive rates. There may be fundamental differences in (for example) the nature of the code bases in question that necessitate these trade-offs in one case but not the other, but it is worth noting that many of our conclusions are the same, and in particular that developer experience is a critical factor in the ultimate success of analysis tools.

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 (supplementing defect reports with tools to assist the developer in navigating and comprehending a large codebase further increases that chance). 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. Table 1 summarises the scalability results for the deployed version of Parfait. These results are based on running our experiments on a 2.6 GHz Intel Xeon E5-2690 or similar machine. It is important to note that the lower performance on the JDK is because of the open-world assumptions.

Codebase	Non-commented Lines of Code	Number of Bug Types	Analysis Time	Runtime in KLOC per min
Oracle Linux Kernel 5 (UEK5)	16.5 Million of C/C++	30	15 minutes	1,100
JDK8 u172	4.5 Million Java, 1.53 Million C++	11 Java, 30 C++ and 4 JNI	100 minutes	61
Internal Web Application	5.4 Million Java	8	5 minutes 30 seconds	982

Table 1: Summary Data on Parfait's Scalability

Figure 7 shows the trend in the number of defects (because of commercial sensitivity, we cannot report the exact number of defects) fixed over a period of 6 years. The two peaks seen in the figure arise when a new analysis is introduced in Parfait. These new analyses generate actionable reports that were previously missing because the analyses were not mature enough to be deployed in practice.

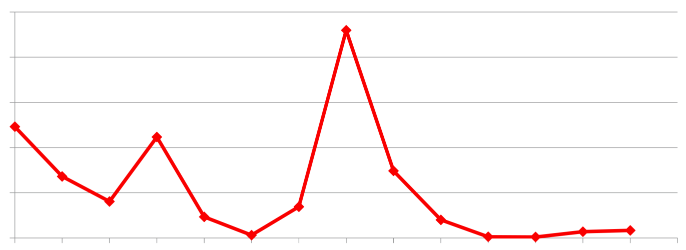


Figure 7: Evolution of Defects Fixed Over Time

It is important to note that the improvements in performance and recall came at least partly as a result of experimentation and trial-and-error. As an example, our implementation of whole-program points-to analysis in Parfait proved unscalable for large codebases such as the JDK and this drove the development of a less precise but more scalable approach (as discussed in 2.1).

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 and Python 3.5 added supports for type hints). Although type hints are generally non-binding, we believe that they could be leveraged to help improve the precision and scalability 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. The incremental analysis described in Figure 2 would need to be extended to analyse micro-services based architectures.

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.

In recent years, the wide adoption of DevOps (see Figure 8), 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.

Thus our vision has two components. The first is based on on our experience of evolving developer-centric tools. The second is based on our observations of how organisations are requiring support for the operational side of software development especially monitoring and providing actionable feedback. Both these aspects need to be addressed for the next generation of security analysis tools to be successful.

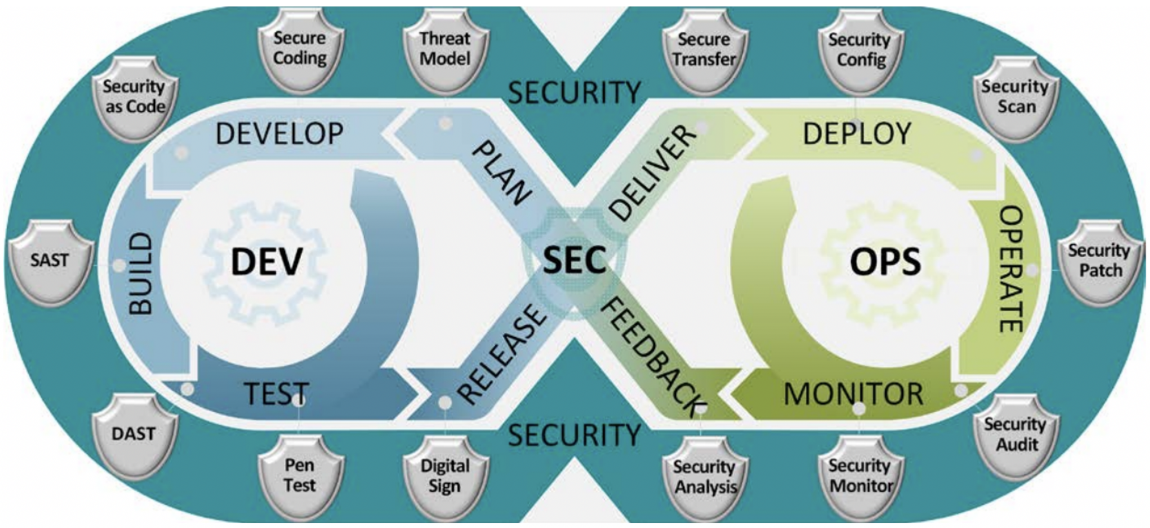


Figure 8: DevSecOps Software Lifecycle

The tools we have discussed so far have been focused on a range of phases within the DevOps cycle, though there is a noticeable lean toward the *development* side of the cycle. Parfait focuses on development, as does Frappé while Affogato, Gelato and BackREST focus on security testing. This leaves *operations* without coverage, but more importantly, it must be recognised that loosely integrating a selection of security-focused tools into a set of DevOps processes only goes partway to achieving the notional *DevSecOps* model in which security permeates the entire cycle.

Throughout the development and deployment of the various application security tools, common themes have emerged; we have discussed lessons learned for individual cases (3) but can perhaps summarise the lessons as follows: *tools must not create friction within the processes*. Indeed, they should ideally reduce the friction perceived by actors in the loop. For individual tools, we have detailed some of the various decisions motivated by this goal, often involving compromises in the precision or comprehensiveness of analysis (guided by strategies developed to minimise the impact on security). For example, one of the main goals of Parfait's development was to minimise false positive reports, as this reduces the burden consumers of the tools (mainly developers). Additionally, reasonable analysis time and supporting incremental analysis are fundamental to Parfait's suitability for inclusion in continuous integration pipelines, so that execution of analysis can be automated. These aspects work to reduce friction when incorporating Parfait. Similar motivations have guided the development of other tools. However, there are still pain points. In many cases, the tools require fine-tuning to be most effective; for a developer, tuning configurations may be seen as a distraction from their "main" task of writing code. Similarly, interrupting developers' workflows with reports from tools such as Parfait might also be perceived as a distraction; the process of producing and testing a fix, even for a correct report, is time taken away from other development. The feedback process from security testers, using tools such as Gelato and Affogato, involves the manual creation of tickets in a tracking system for developers to address; the problem then needs to be understood, and communicated to different people in different phases of the cycle.

The initial integration of tools also needs to be as simple as possible. 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 previously worked independently, they should now need to 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 (as part of continuous integration) and avoid having developers review false positives; feedback from such dynamic analysis could also be used to automatically generate configuration for static analysis, improving its precision. Conversely, unwanted behaviour noticed as part a monitoring solution (e.g., based on our initial work [VGB^{+}22]) deployed in an operating application could guide static and dynamic analyses to "look harder" at particular points in the code by enabling more complex but targeted analyses. Ideally, these interactions between various tools at various stages would be autonomous—the tools would act intelligently together.

As well as increasing the interaction between existing tools and simplifying integration, we need to consider the many changes in the software development landscape. 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. At the same time, the rise in machine learning has created opportunities to integrate learning-based techniques with program analysis techniques to enhance the overall experience of the the latter tools. We already live with intelligent applications. For example, many systems (e.g., email, text-based messaging, search) make use of predictive text. Microsoft's Visual Studio IntelliCode[6] provides code completion suggestions based on thousands of open source projects.

[6] `https://marketplace.visualstudio.com/items?itemName=VisualStudioExptTeam.vscodeintellicode`

Amazon's CodeGuru Reviewer [7] identifies critical issues and hard-to-find performance bugs and suggests ways to fix them. Facebook's GetAFix [BSPC19] finds fixes for lint-based errors and offers them to developers, and Amazon's CodeGuru Profiler [8] finds the most expensive lines of code in an application, based on runtime data, and recommends improvements to the code. These are examples of intelligent coding, intelligent testing, and intelligent monitoring tools that benefit developers, QA staff and operations staff. Integration of learning-based techniques that provide intelligence into all our tools is a significant part of the vision of Intelligent Application Security.

Intelligent Application Security aims to provide an automated approach to integrate security into all aspects of application development and operations, at scale, using learning techniques that incorporate signals from the code and beyond, to provide actionable intelligence to developers, security analysts, operations staff, and autonomous systems.

Here are some examples of instances of intelligent application security:

- A service that warns a developer about a vulnerability that exists in a third-party library the developer is a standard use of a Software Composition Analysis tool. Enhancing the reporting to suggest to the developer to upgrade to a specific editor clean version of the library is example of Intelligent Security Coding. Such a service can be based on an extension of Oracle's Vulnerability Scanning Service [9].

- Examples of Intelligent Security Testing include being able to automatically generate tests for security purposes, or automatically proposing bug fixes to failing security tests as outlined in item 4. A potential deployment model is enhancing the reporting of our static analysers to include such proposed bug fixes.

- Examples of Intelligent Security Monitoring include autonomously blocking attacks against a vulnerable running cloud service, or autonomously updating allowlists being. This is expanded below in item 2.

We are currently working on the following items towards our IAS vision. Our aim is to deliver this via Oracle's Application Dependency Management System [10] where certain components have emerged from our group. This has enabled us to experiment with the necessary steps needed for integration with the DevOps process.

1. Tools related to supply chain security including software composition analysis and analysis of build infrastructures. Here the aim is to have a precise identification of the vulnerable artifacts and also ensure that environment that was used to build them is trust-worthy. To facilitate collaboration with the community, we have recently open-sourced[11] an initial version of a verifier (called Macaron)for certain properties defined by frameworks such as SLSA [EW22]. Macaron can be viewed as a static analysis tool for a class of build systems. Macaron also supports the specification of security policies in CUE to ensure that the build-related workflows are not tampered with. We have enabled the integrated of Macaron with GitHub Actions and published a blog article [12] with a view of building a community who can contribute to our vision.

[7]`https://docs.aws.amazon.com/codeguru/latest/reviewer-ug/welcome.html`

[8]`https://docs.aws.amazon.com/codeguru/latest/profiler-ug/what-is-codeguru-profiler.html`

[9]https://docs.oracle.com/en-us/iaas/scanning/using/overview.htm

[10]https://docs.oracle.com/en-us/iaas/Content/application-dependency-management/home.htm

[11]https://github.com/oracle-samples/macaron

[12]https://blogs.oracle.com/developers/post/macaron-supply-chain-conformance-verifier-and-policy-engine-for-slsa

2. Exploring the use of ideas from program synthesis to generate security protections, in the context of RASP[13] solutions, that can be deployed at run-time [VGB$^+$22]. We have completed an initial design of our autonomous RASP solution[14], that takes into account the feedback from operations and upgrades the synthesised protections.

3. Malware detection [TCC$^+$21] which is related to software supply chain issues but also hopes to address the issue of using system-level binaries and

4. Tools that take vulnerability reports and generate potential patches which is related to automatic program repair [SNGR21]. We are currently looking at repairing `pom.xml` files so that vulnerable versions of artifacts are not used as well as deploying template-based repair for a class of injection vulnerabilities. This will enable us to get valuable feedback from developers which will guide the practical aspects of the research. One key requirement is that the repair-recommender system should propose only a handful of fixes.

5. We are also exploring the use of machine learning techniques to help with bug-triaging based on service requests raised in operations. The challenge that we have identified includes the mapping of information in the service requests to specific code fragments. This is because in our experience many of the service requests do not actually have any code fragments.

We have started to implement our goal of integrating the work we have reported here towards IAS. The challenge of how to transfer this technology from research into practice still remains and needs to be addressed as we make further progress towards our vision.

While completion of current work will be a step towards IAS, the broader vision requires much more—better integration between different tools across all parts of the DevSecOps cycle, additional tools to address modern application design strategies such as microservice architectures , incorporation of techniques such as machine learning to improve precision and reporting, and efforts to further "close the loop" with enhancements such as automatic program repair—all hyper-automated, as termed by Gartner[15]. In our context, the tools we develop need to work smoothly with existing architecture for applications [16] including messaging [17] and event-driven systems [18].

In Figure 9, an initial architecture is presented depicting some of the various integrations that have been discussed. Changes made by developers run immediate (incremental, fast-running) SAST analysis and integration tests; monitoring during the latter can be used to automate necessary changes to RASP configuration that is used in deployment. Similarly, DAST may guide RASP configuration. Any runtime anomalies detected by RASP, together with those detected by periodic DAST runs, can be fed to an automated analysis service which can attempt to verify the presence of a defect (potentially using a targeted static analysis or guided hybrid static/dynamic analysis) and generate test cases and automated repairs. SCA runs (performed periodically, or potentially triggered by incoming vulnerability reports for components) can also produce repairs by selecting non-vulnerable versions of dependencies. While we already have many of the individual pieces, the work on integrating them into an autonomous architecture such as the one presented lies ahead of us. It is the communication between the various components that will use the messaging or event-driven services. Certainly, the journey has only just begun.

[13]https://www.gartner.com/en/information-technology/glossary/runtime-application-self-protection-rasp
[14]https://labs.oracle.com/pls/apex/f?p=94065:10:107717911132850:8759
[15]https://www.gartner.com/en/information-technology/glossary/hyperautomation
[16]https://docs.oracle.com/en/solutions/mad-web-mobile/index.html
[17]https://docs.oracle.com/en/solutions/mad-messaging-pattern/index.html
[18]https://docs.oracle.com/en/solutions/mad-event-driven/index.html

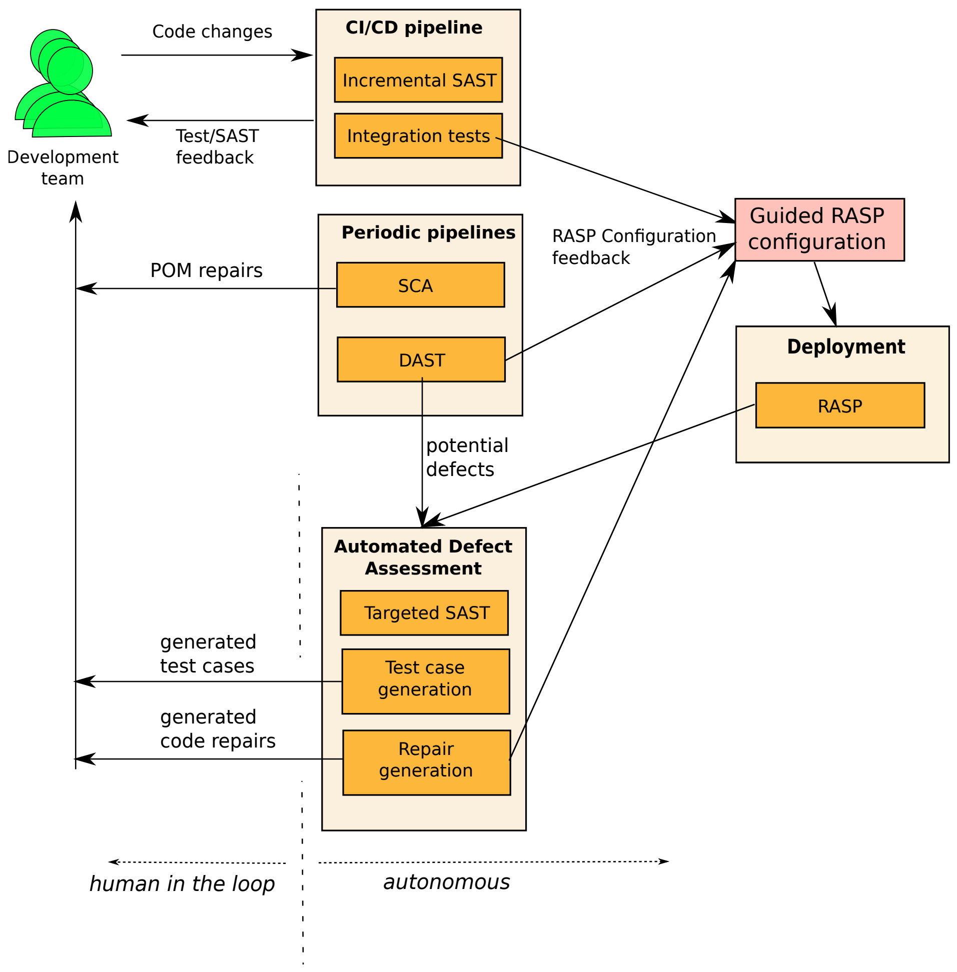


Figure 9: Initial IAS architecture

5 Conclusions

In this paper we have summarised our industrial experiences into the research, development and deployment of program analysis tools. Our main focus on enabling deployment of such tools has been on understanding the key criteria that development organisations inside Oracle take into account in order to deploy these tools without adding to cognitive overload of their developers.

Our experience shows that tool deployment is driven by developers and key driving forces for adoption of program analysis tools are low false positive rate, ease of integration in the developer's workflow, scalability to handle industrial size systems and results that are easy to understand. To enable low false positive rate and scalability of the analysis, different program analysis techniques need to be adapted, based on language, to be able to meet this criteria. Further, based on language, the need for static and dynamic analyses becomes important, with the need for hybrid analyses as well.

Our vision for the future of program analysis tools is integration into the DevOps model to enable DevSecOps. In our vision, DevSecOps can be implemented through Intelligent Application Security – a collaboration of program analysis tools that look into the various aspects of the development, testing, deployment, monitoring cycle, while sharing results amongst the tools to better inform them. At a high-level, the challenges we face include the need to work with existing DevOps and cloud-based service architecture. We are actively researching into this vision to make it a reality.

Acknowledgement We thank the entire program analysis team (both past and present corresponding to approximately 100 full time equivalent staff) for their work on the different projects over the past decade. This paper would not have been possible without their contributions.

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.

[AFK+20] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. Static analysis of java enterprise applications: frameworks and caches, the elephants in the room. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 794–807, 2020.

[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.

[ATBT22] Mark W Aldrich, Alexi Turcotte, Matthew Blanco, and Frank Tip. Augur: Dynamic taint analysis for asynchronous javascript. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–4, 2022.

[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.

[BSPC19] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM Programming Languages*, (OOPSLA), Oct 2019.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*, pages 238–252. ACM, 1977.

[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.

[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.

[CLO07] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206, 2007.

[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.

[EW22] William Enck and Laurie Williams. Top five challenges in software supply chain security: Observations from 30 industry and government organizations. *IEEE Security & Privacy*, 20(2):96–100, 2022.

[FHRS08] Ansgar Fehnker, Ralf Huuck, Felix Rauch, and Sean Seefried. Some assembly required - program analysis of embedded system code. In *International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 28–29, 2008.

[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.

[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.

[HVdM06] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. *codeQuest:* scalable source code queries with datalog. In *ECOOP*, volume 4067 of *LNCS*, pages 2–27. Springer, 2006.

[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.

[JMT09] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In *Static Analysis, 16th International Symposium (SAS)*, pages 238–255, 2009.

[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 Schäf, and Mark Santolucito. Analyzing infrastructure as code to prevent intra-update sniping vulnerabilities. In *TACAS*, number 12652 in LNCS, pages 105–123. Springer, 2021.

[LR92] William Landi and Barbara G Ryder. A safe approximate algorithm for interprocedural aliasing. *ACM SIGPLAN Notices*, 27(7):235–248, 1992.

[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.

[LWJ$^+$12] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. Safe: Formal specification and implementation of a scalable analysis fr amework for ecmascript. In *In International Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2012.

[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.

[NTM21] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. Modular call graph construction for security scanning of node. js applications. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 29–41, 2021.

[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.

[RLBV10] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2010.

[SAE$^+$18] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4):58–66, 2018.

[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.

[SNGR21] Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. Concolic program repair. In *Programming Language Design and Implementation*, PLDI, pages 390–405. ACM, 2021.

[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.

[Tho21] Patrick Thomson. Static analysis: An introduction the fundamental challenge of software engineering is one of complexity. *acmqueue*, 19, 2021.

[TPF$^+$09] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. *ACM Sigplan Notices*, 44(6):87–97, 2009.

[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.

[VHDM07] Mathieu Verbaere, Elnar Hajiyev, and Oege De Moor. Improve software quality with semmlecode: An eclipse plugin for semantic code search. In *Object-Oriented Programming Systems and Applications Companion*, pages 880–881. ACM, 2007.

[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.

[WR13] Shiyi Wei and Barbara G Ryder. Practical blended taint analysis for javascript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 336–346, 2013.