

# Self-Optimizing Static Program Analysis

— SOSA —

- **Principal investigator (PI):** Eric Bodden
- **Host institution:** Paderborn University, Germany
- **Proposal duration:** 60 months

Software systems pervade our personal and professional lives, yet their insecurity threaten our society. To assure that software systems are dependable and secure, one must reason about their code. Static program analysis enables such reasoning. It can be applied to individual software components, and it can show not only the presence but also prove the absence of bugs and vulnerabilities. Yet, to be useful to software developers, static analyses must be adapted to the context in which they are used. Studies show that poorly adapted analyses slow down rather than assist development. They report large sets of false warnings that distract developers from actual bugs, which the analyses often miss. They often run so long that results are reported when they are already outdated.

SOSA's main research hypothesis is that one can *generate* precise and efficient static analyses of software systems by making static analysis self-aware and *self-optimizing*. With SOSA, a static analysis conducts analyses and optimizations not just of programs but *of itself*. This is a groundbreaking paradigm shift: no previous research has regarded complex program analyses themselves as the primary object of automated analysis and optimization.

SOSA will introduce, for the first time, static analyses whose execution is not pre-defined by their creators but is inherently self-adaptive. In result, analyses automatically adapt themselves to yield a performance/precision tradeoff that is optimal with respect to how the analysis is deployed and which program it analyzes. With SOSA, static analyses will report true and relevant warnings at minimal analysis time without requiring manual optimizations by end users.

SOSA will boost progress in the field of program-analysis research, mapping the landscape of static-analysis optimizations and how and where they are best applied. By enabling software developers to optimally deploy static analyses with ease, SOSA will help secure millions of software systems.

## A. Extended Synopsis of the Scientific Proposal

### A.1. Motivation and Aims

Software has become omnipresent in our everyday personal and professional lives. For our society and industry to function, it is essential that these systems are dependable and secure. Also the EU has recognized this: in 2023, it has put forward a draft for the Cyber Resilience Act [50]. If implemented as currently suggested, it will demand a secure software engineering methodology to be used for any software-enabled product sold in the European Union. For many software developing companies, this will entail a radical change. They will not only need to reason about the security of their software design but also will need to effectively deploy automated software assurance tools. Software assurance will be mandated for software developers throughout the EU.

Static program analysis, including variants such as abstract interpretation [11] and formal software verification [2, 14], is *the* primary technique to enable such assurance. Opposed to testing and fuzzing, static analysis covers all possible program executions, thereby not only showing the presence but also proving the absence of bugs and vulnerabilities. It can analyze incomplete code, such as libraries and frameworks, which one cannot execute in isolation. In modern applications, often 80-90% of the code come from libraries and frameworks [36].

**Although static analysis is such a powerful tool, for decades it has struggled to obtain widespread adoption.** This will become highly problematic when the EU now plans to mandate it on a large scale. While particularly its security variant Static Application Security Testing (SAST), on which I will focus here, has become part of many continuous integration pipelines, studies show that if such analyses are poorly adapted to the development context, they can slow down rather than assist development [3, 10, 38]. They then report large sets of false warnings. These distract developers from actual vulnerabilities, which the analyses also often miss. Moreover, they often run so long that results are reported when they are already outdated [31].

**With the EU soon *demanding* the widespread adoption of this technology, static analyses will need to be effectively deployed and configured by hundreds of thousands of software engineers who are novices to static analysis.** This generates an urgent need for automation: we need a technology that can optimally configure and deploy a static analysis for any given usage context. The goal of SOSA is to yield that very automation.

SOSA will address the above challenges through a radical paradigm shift:

**SOSA’s main research hypothesis is that one can *automatically generate* precise and efficient static analyses of software systems by making static analysis self-aware and self-optimizing.**

No previous project has engaged in this idea, its feasibility has therefore never before been tested. With SOSA, I seek to explore to which extent it is possible for a static analysis to conduct analyses and optimizations not just of programs but *of itself*. This is a *groundbreaking paradigm shift*: never before have researchers regarded complex program analyses themselves as the primary object of automated analysis and optimization.

**By combining research from static analysis, programming languages, just-in-time compilers and autonomic systems, SOSA will for the first time create static analyses that are essentially self-aware [27], that will self-adapt, to self-optimize.** With this, SOSA will:

- enable software engineers to use static analyses with ease, leaving any required customizations and optimization to the analysis itself,
- enable researchers to finally map the landscape of static-analysis optimizations and how and where they can be most successfully applied,
- aid static analysis and related fields such as software verification and validation in solving the most daunting research challenges that currently stand in the way of a widespread industry adoption, and thus indirectly
- help secure the millions of software systems that we all have learned to rely on.

### A.2. A User’s Perspective

Figure 1 shows the *current state of the art in static program analysis*: A software developer deploys and configures the analysis tool and then runs it using a general-purpose execution environment, for instance a Java Virtual Machine, or these days frequently also a Datalog solver. This will result in initial analysis reports, usually at first containing many false warnings and missing many actually essential bugs and vulnerabilities. **Many developers will disappointedly abandon the tool at this stage, the more perseverant ones will now engage in a trial-and-error loop in which they attempt to reconfigure parts of the analysis and maybe even**

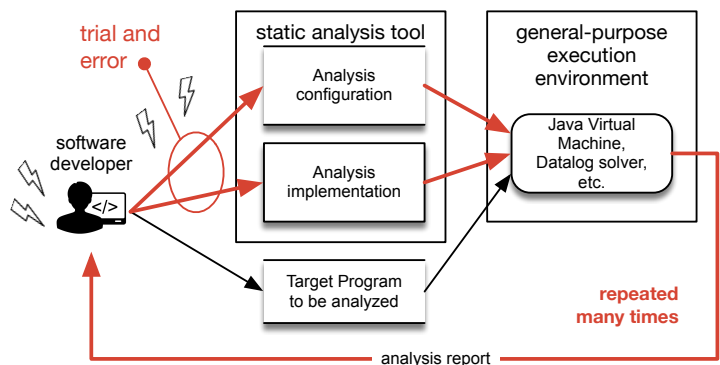


Figure 1: State-of-the-art static analyses solutions force the user to repeatedly perform complex configuration

**alter parts of the analysis implementation.** The human is *in* the loop. In the past years I have led industry-financed projects ranging from SMEs with 20 employees to Fortune 100 companies with dedicated SAST research teams, and all were facing the same daunting challenges, even in cases that involved tools from leading industrial vendors.

**Figure 2 shows how SOSA’s paradigm shift will address those challenges.** With SOSA, software developers must no longer adapt the analysis at all because the analysis itself becomes self-aware [27] and self-optimizing. **Developers instead declaratively define what to analyze but not how the analysis should execute.** They then *automatically*

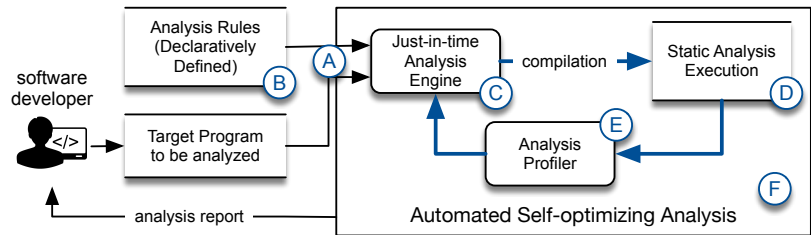


Figure 2: Envisioned solution from the user’s point of view. (A)–(F) relate to objectives / research challenges described in A.4.

**receive precise and useful analysis reports for the target programs that they provide.** The human is taken *out of* the loop. The analysis adapts to—and optimizes itself for—these programs automatically: A just-in-time analysis engine analyzes not only the target program in question *but also the analysis itself* and how well it succeeds in operating on that program. This is enabled by a specialized execution engine that is aware of the domain-specific<sup>1</sup> concepts that define the static analysis as well as a special analysis profiler that allows the analysis to uncover, e.g. where it loses precision or performance.

**Concrete example optimizations** *What can this actually look like?* My first example relates to **taint analysis**, which is crucial for detecting **injection vulnerabilities and data leaks**. Grech et al. have previously investigated how to speed up taint analysis by performing a control-flow-insensitive analysis, i.e., disregarding the actual ordering of program statements [16]. Such an analysis would correctly signal data leaks in code such as `x = secret(); print(x);`, yet it would also cause a false leak warning in the inverse case `print(x); x = secret();`. A self-optimizing static analysis would seek to efficiently recognize these situations and compute control-flow information to disambiguate them exactly where needed. Recent work in sparse value flow analysis, for instance, has shown that similar on-demand computations can lead to large runtime savings [18, 26, 34].

As a second example, let us consider **buffer overflow detection**, which requires the exact computation of array indices—something typically conducted by a *constant propagation analysis*. In the general case, full constant analysis requires one to use the so-called monotone framework [25], which can computationally become very expensive. Yet, in many cases, programs access buffers only through linear arithmetic of the form `buffer[x*w+y]`. For such programs, the analysis can then self-optimize by performing a *linear* constant analysis, which can be very efficiently executed in the *inter-procedural distributive environments* framework [39]: it is linear in the size of the program and can cope well even with infinitely wide domains such as the domain of all integers. Currently, analysis experts would typically implement either linear constant analysis, which can become imprecise in case of non-linear program computations, or full constant analysis, which can be expensive. SOSA will automatically derive both implementations from one declarative definition and choose the optimal one depending on the target program.

The beauty of a self-optimizing static analyzer is that it can attempt to apply such optimizations to *any* static analysis that even novice developers define. The analyzer itself will recognize under which situation e.g. a flow-insensitive analysis component pays off, and where sparse program analysis actually helps. It can even make different choices depending on the level of speed or assurance the user desires. For instance, while analyses running in an integrated development environment (IDE) must be tuned for speed, and should focus on the developers’s current edit context [13], analyses running in a continuous integration (CI) environment can run on the system level and spend more time. In result, SOSA will enable non-experts to define and use static analyses with ease, freeing them from the burden of implementing them precisely and efficiently.

Other examples of possible optimizations include sparsification, the choice of optimal context information, of analysis direction, of widening operators, of the right pointer analysis, etc. SOSA will, for the first time, yield tool automation that will comprehensively compute optimal choices automatically.

### A.3. State of the Art and Novelty of the Approach

Self-awareness has been studied in the context of autonomic and self-aware computing [21, 27], but so far has not been applied to static analysis. According to Giese et al. [27], a *self-aware* system learns models capturing models about itself and its environment, and reasons using these models, allowing them to act based on their knowledge and reasoning. The solution we envision will also be *self-adaptive* in that the analysis uses such actions to adapt its own analysis strategy, optimizing its own execution.

Static-analysis researchers have previously developed approaches to the declarative definition of static analyses. However, no current approach allows a static analysis to reason about itself and its own execution. Most current

<sup>1</sup>Note that when I refer to “domain-specific” concepts or optimizations, the domain I refer to is the one of *static analysis* itself.

forms of declarative definitions are not domain specific, i.e., have no understanding of what a program analysis actually is. Using Doop [8], one implements static analyses in the general-purpose logic programming language Datalog, an approach that is popular among static analysis experts, and which I have also used myself [12]. Yet, while such an approach succeeds in making use of *generic* optimizations on the level of Datalog language and engine, it cannot easily support optimizations that wish to exploit the semantic properties of the defined static analysis. Recent approaches have extended Datalog with the concept of lattices [24, 35, 47], an essential mathematical structure in static analyses. Yet, while this support enables one to conduct a broader set of static analyses, e.g. constant propagation, which pure Datalog cannot efficiently support, it does not enable other optimizations. To give just some examples, researchers have shown that one can greatly improve the performance of static analyses (1) through so-called staging, in which the analyses is divided into individual, increasingly precise and costly stages [4, 6, 22, 30, 33, 43, 44], (2) by making the analysis sparse, i.e., having it operate on lean data structures such as value-flow graphs [9, 34, 37, 45] and (3) through parallelization [19, 42]. Yet, these were typically implemented as “one-off” optimizations, applied to a single static analysis in a single application context. For the first time, SOSA will provide a *dedicated domain-specific* analysis definition language. This will allow the analysis to understand its own execution and where these and other powerful static analysis optimizations can best be applied—automatically.

For decades now, researchers have investigated just-in-time optimization and compilation for general-purpose programs. GraalVM [55] is a versatile framework for implementing efficient program execution environments. It can interface with LLVM [29], which itself has been used to implement powerful just-in-time compilers. Truffle [54, 55] is a frontend to GraalVM that allows one to support multiple differing input languages. We will investigate GraalVM and Truffle but also LLVM as potential technology base for SOSA’s just-in-time optimization and compilation. All these technologies have been developed for compiling programs, not static analyses of programs, and will require significant, non-trivial extensions. We will also investigate the utility of partial evaluation, automated program specialization and multi-stage programming [23, 48, 49, 52, 53], as SOSA’s goal can be seen as the desire to specialize a complex program, the static analysis, to a complex input: the target program at hand.

#### A.4. Research Challenges and Objectives

To achieve its goal, SOSA must solve a number of key research challenges. I sort these challenges into five key objectives (A)–(F). These objectives also induce related work areas, appropriately labeled in Figure 2.

**(A) Joint intermediate representation of static analysis and target program.** SOSA’s goal is to automatically adapt and optimize the execution of a (declaratively defined) static analysis of a given target program. One key challenge in enabling this will be to design and implement an intermediate representation (IR) that incorporates the relevant aspects of *both* entities, the analysis and the target program. This aspect is important because this *joint* representation will enable the analysis to reason about itself and its own execution. The IR must allow for different flavours of static analysis, e.g., flow-(in)sensitive, context-(in)sensitive, field-(in)sensitive, etc., and should allow not just for ahead-of-time but also just-in-time analysis and optimization. Many such automated optimizations should be supported, including automated staging, sparsification and parallelization. No prior project has designed a “joint IR” with such versatility, which is also not surprising because SOSA is the first project with the explicit need for such a representation.

**(B) Declarative definition of static analyses in domain-specific terms.** While also existing approaches that base on Datalog seek to define *what* an analysis should detect and signal, without defining *how* the analysis should execute, the genericity of Datalog makes it virtually impossible for automated static analyses to analyze, understand and optimize themselves *in domain-specific terms*. SOSA’s objective is thus to design a language in which, on the one hand, analysis rules themselves can be written, understood and manipulated by both static analysis experts and non-expert developers, yet on the other hand one also enables powerful automated *domain-specific* reasoning through the self-adaptive analysis engine, using the above IR. To enable such reasoning, the language must not only have a clearly defined, intuitive syntax and semantics but must also provide the optimization access to domain-specific constructs. For instance, the self-adaptive analysis engine should be able to understand which rules define flow functions, which define a lattice, what their mathematical properties are, how these can be exploited, etc.

**(C) Self-adaptivity and self-optimization through self-awareness.** SOSA seeks to enable the static analysis to become self-aware of its own semantics and execution, thus for the first time allowing itself to apply powerful, domain-specific optimizations, oftentimes even combining sets of otherwise individual (formerly “one-off”) optimizations, and even specializing these optimizations with respect to the target program that is analyzed. One key challenge will thus be to derive—ideally automatically uncover—a model of the analysis itself that allows the analysis to reason autonomously about where it succeeds and where it fails in terms of its optimizations goals, and why, so that it can adapt its optimization strategies based on those insights. Another challenge I foresee is communication towards the software developer. For instance, the analysis might uncover that one might speed up its own execution dramatically but under a slight loss of precision. How to best communicate such tradeoffs between pareto-optimal solutions? How can users make an informed judgement?

Ⓓ **Automated compilation of static analyses.** In the past two years we were lucky to be able to develop the commercial static analyzer *Contrast Scan* [↗](#) in collaboration with *Contrast Security Inc.*. Astonishingly, this project revealed that one can obtain significant speedups, sometimes of multiple orders of magnitudes by implementing algorithmics in Rust instead of Java, which avoids runtime overhead due to the repeated garbage collection of short-lived objects. (Rust runs without the necessity for garbage collection.) Another surprisingly successful low-level optimization arranges the analysis’ internal state in specially optimized data structures, e.g. by specially selected types and combinations of sets and maps. Yet, the optimal selection here may depend on the program under analysis. The objective here is thus to investigate whether one can generalize these findings to a wide range of static analyses, and whether one can make also such low-level optimizations easily accessible by “compiling” the static analysis such that it will use the data structures and memory-access patterns that are ideal for the analysis conducted of the target program.

Ⓔ **Domain-specific profiling of static analyses.** Self-adaptivity can yield self-optimization only if the analysis can obtain the required situational awareness. This raises the challenge of obtaining detailed profiling information about program-analysis runs, on a level of abstraction that allows the self-adaptive analysis to identify potential also for domain-specific optimizations. A key technical challenge will be to establish and maintain a rigorous traceability between (1) the executing static analysis and (2) the original target program and (3) analysis rule definitions, both from which the binary was derived. Current approaches largely lack such traceability, which greatly hinders progress in the field. SOSA’s automation approach is an important enabler towards such traceability.

Ⓕ **Systematic assessment of static-analysis optimizations.** Previous work has almost exclusively evaluated the effects of *individual* optimizations of static analyses. This is because multiple optimizations can often be sensibly combined, and thus many possible combinations may exist. Yet, one given optimization may impede or counteract another, or there might even be synergies between optimizations. Through its automation, SOSA will, for the first time, enable one to map the landscape of these optimizations, and find combinations optimal for certain classes of programs and analysis-deployment scenarios. A challenge will be to explain *why* certain combinations prove optimal in a given case.

As shown, while SOSA seeks to address real-world challenges, it cannot be funded by private corporations because of the associated high risks these challenges involve. Any commercial exploitation of the project’s results will not be able to commence until after several years into the project.

## A.5. Methodology and Research Plan

SOSA comprises seven work packages. While WP1, a classification of existing static-analysis optimizations, will be conducted jointly by the entire initial research staff, WP2–WP7 will each be led by one Ph.D. student financed by the ERC. In addition, two PostDocs, financed out of my university budget, will combine their own independent research agenda with SOSA’s methodology and act as experts for WP2–WP4, resp. WP5–WP7.

**WP1: Classifying static analysis optimizations** Deliberately as an initial first team activity, the entire initial research staff will systematically survey, and thereby classify, optimizations of static analyses from the scientific literature. This shall serve as a foundation to map the design space of possible optimizations which SOSA then seeks to eventually apply automatically. The joint activity will also foster initial team building.

**WP2: Joint Intermediate Representation, addresses Objective Ⓐ, 1st Ph.D. student.**

Based on the previous classification, we will research the required concepts for the joint IR for the target program and the static analysis. For one, the IR will allow for powerful ahead-of-time optimizations, i.e., optimizations that execute *before* the analysis commences. These include, e.g., query optimizations that exploit sharing effects between multiple related queries [41] but also staging and sparsification. Such optimizations require only information about the analysis, not the target program. Yet second, the IR will also enable just-in-time optimizations, which run *during* analysis. To this end, the IR must incorporate information about the target program as well. Recent work on Sparse IFDS by us and others show encouraging first results [17, 26] in this area. We will evaluate whether it is advisable to use one and the same IR or separate IRs for both ahead-of-time and just-in-time optimizations. We will formalize and implement the IR(s).

**WP3: Declarative Analysis-rule Definition Language, addresses Objective Ⓑ, 2nd Ph.D. student.**

We will investigate the optimal language constructs, syntax and semantics for the envisioned domain-specific static-analysis rule definition language (DSL). This language will allow software developers who are no experts in static analysis to specify *what* a static analysis should compute without stating *how* this result should be computed. Rules expressed in the language will translate into the above IR. The language will be rich enough to allow one to express a variety of relevant static-analysis problems, e.g., taint and tystate analysis, pointer analysis as well as constant propagation. These analyses are essential to identifying security vulnerabilities in real-world programs, e.g. use-after-free or cross-site scripting. Yet, the language should be simple enough for developers to use, which we will validate through appropriate user studies. As we did in previous work, we will employ user-centered design [32] to find define the DSL as user-friendly as we can.

**WP4: Self-Awareness and Self-Optimization, addresses Objective Ⓒ, 3rd Ph.D. student.**

We seek to make the static analysis self-aware by allowing it to reason about its own execution on the given



target program. In particular, we will build on concepts from autonomic and self-aware computing [15, 21, 27] to allow the analysis to build a model of its execution, thereby allowing itself to identify optimization potential in its own execution. In the past, I have gained experience in researching self-adaptive systems as a principle investigator within the German Research Foundation’s (DFG) Collaborative Research Center 901 *On-the-fly computing*. One of the most popular architectures for autonomic computing is MAPE-K [21]. However, MAPE-K is typically restricted to heuristics that are fixed ahead of time [27]. We will thus also investigate model-based learning and reasoning (LRA-M) loops, in which gathered empirical data are used as a basis for an *ongoing* learning process, as part of which observations are abstracted into models capturing potentially relevant aspects of the system (here the analysis) and its environment (here the execution environment) [15].

**WP5: Automated Analysis Compilation, addresses Objective ④, 4th Ph.D. student.** Initial successes during our development of the commercial static analyzer *Contrast Scan*  $\varnothing$  over the past two years have shown that one can obtain significant speedups, sometimes of multiple orders of magnitudes by arranging the analysis’ internal state in specially optimized data structures, e.g. by specially selected types and combinations of sets and maps. This is likely due to optimized memory and cache access patterns. The effectiveness of a given data-structure choice, however, can depend on the program under analysis. We thus will design, and implement such data structures but also implement a way to compile a given static analysis such that it then makes efficient use of these data structures. Compilation can benefit from information collected on previous analysis runs, as described next.

**WP6: Domain-specific profiling, addresses Objective ⑤, 5th Ph.D. student.** Here we seek to correlate analysis runs, and the performance and precision observed during those runs, with analysis configurations. To identify the potential for powerful just-in-time optimizations, we will follow a white-box approach, i.e., collect not just data about the analysis output but also about its internal workings: In which program parts does the analysis computation remain precise, where does precision deteriorate? Which parts of the analysis cause the deterioration, and how essential are these parts? Which analysis optimizations counter-act one another and where? No previous project has dedicated itself to developing a profiler that can answer such questions specific to the domain of static analysis, let alone integrate with a compilation of static analyses.

**WP7: Large-Scale Evaluation, addresses Objective ⑥, 6th Ph.D. student** Through its automation, SOSA will for the first time enable one to map the landscape of (combinations of) analysis optimizations. We plan to conduct large-scale experiments, backed by Paderborn University’s supercomputing cluster Noctua, in which we use SOSA’s infrastructure to automatically explore and map the design space. We will focus on extracting correlations between the effectiveness optimizations and their combinations on one hand, and properties of the analyzed program on the other hand. To address the challenge of explaining *why* (combinations of) optimizations work *when*, we will collaborate with my colleagues from Paderborn University’s Collaborative Research Center *Co-constructing Explainability*.

We will make available online, as open source, all code and benchmarks, as properly curated and documented artifacts. To this end will also provide an appropriate data management plan.

## A.6. Risk Analysis and Feasibility

Self-optimization is a disruptive vision, which is why it entails several technical risks. Nonetheless, given my own unique expertise in the field, both foundational and in practice [5], I am the best person suited to foresee those risks and mitigate them to the extent possible. The research in the areas of self-optimization and just-in-time optimization are particularly risky because the required machinery will be very complex: a self-analysis reasoning about a static analysis that itself reasons about a program. Only a few people can design the required machinery such that one can avoid bugs in that machinery early on and maintain it in the long run. We will mitigate this risk, by lowering the complexity, by modularizing the solution as much as possible [20], and by building on top of my group’s in-depth experience in building complex industrial-grade tools with excellent test-and-build infrastructure.

## A.7. Expected Impact

As a practical impact, SOSA will yield a quantum leap towards software security on a global scale. It may be the key enabler to help scale static analysis to the scale required by the planned introduction of the Cyber Resilience Act. This will benefit all application domains that make use of software components.

In the scientific community, SOSA will cause a paradigm shift with respect to how we will design and implement static analyses in the future. Whereas currently the static-analysis community frequently itself engages in trial-and-error, addressing specific precision and performance problems as they are observed when running static analyses of real-world code, SOSA will provide the first platform to identify and mitigate such problems automatically. By observing the joint effects of applying also multiple analysis optimizations in combination, and in a way that is specialized to the target program, SOSA will finally allow one to map the entire landscape of static analysis optimizations. At last we will understand which optimizations are required when, and which ones pay off when exactly. The open-source artifacts SOSA will produce will allow researchers worldwide to comparatively evaluate existing and upcoming static-analysis and optimization techniques.

## B. Curriculum Vitae

Born February 20th, 1980, in Aachen, Germany

Heinz Nixdorf Institute [↗](#) at Paderborn University [↗](#), Fürstenallee 11, 33102 Paderborn

ACM Distinguished Member · Seven ACM/IEEE Distinguished Papers · >10600+ citations · h-index  $\geq 44$   
+49 5251 606563 · [eric.bodden@upb.de](mailto:eric.bodden@upb.de) [↗](#) · <https://bodden.de/> [↗](#) · ORCID 0000-0003-3470-3647 [↗](#)

Eric Bodden is one of the world's leading and most influential researchers on automated program analysis. He has made multiple fundamental contributions to the field, including:

- FlowDroid: Efficient and precise taint analysis for Android [1]
- SPLlift: Static dataflow analysis for software product lines in minutes instead of years [7]
- Synchronized Pushdown Systems: Highly precise and efficient demand-driven pointer analysis [46]

Although he has just become eligible for an ERC Advanced Grant, his work has already won seven distinguished paper awards and cited many thousand times. The static analysis frameworks [Soot](#) [↗](#), [FlowDroid](#) [↗](#) and [PhASAR](#) [↗](#) that he maintains are annually used by hundreds of researchers and resemble a large fraction of the central research infrastructure in the field. Bodden has lead the commercial development of [Contrast Scan](#) [↗](#).

### Education

2006 – 2009 Ph.D. in Computer Science (no grade), McGill University, Canada

2000 – 2005 Diploma in Computer Science (with distinction), RWTH Aachen University, Germany

### Current positions

since 2016 Professor for Secure Software Engineering, Heinz Nixdorf Institute at Paderborn University

since 2016 Director for Software Engineering & IT-Security, Fraunhofer Institute for Mechatronic System Design (IEM), Paderborn

### Previous positions

2013 – 2015 Department Head for Secure Software Engineering, Fraunhofer SIT, Darmstadt

2013 – 2015 Cooperative Professor (W3) for Secure Software Engineering, Fraunhofer SIT & Technische Universität Darmstadt

2012 – 2015 Emmy Noether Research Group Leader, Technische Universität Darmstadt

2011 – 2015 Claude Shannon Research Group Leader, European Center for Security and Privacy by Design, Technische Universität Darmstadt

2009 – 2011 Postdoctoral Researcher, Software Technology Group of Prof. Dr. Mira Mezini, Technische Universität Darmstadt

### Career Contribution Awards

2019 **ACM Distinguished Member** for Outstanding Scientific Contributions to Computing, awarded in the first possible year

2014 **Heinz Maier-Leibnitz-Preis of German Research Foundation (DFG), EUR 25.000**, the highest award given by the German Research Foundation (DFG) for young scientists (10 people per year all over Germany, over all subjects)

### Distinguished Paper Awards

2023 **IEEE SANER 2023 Distinguished Paper Award** for *Enhancing Comprehension and Navigation in Jupyter Notebooks with Static Analysis*, with Ashwin Prasad, Jiawei Wang, Li Li

2022 **IEEE SCAM 2022 Distinguished Paper Award** for *To what extent can we analyze Kotlin programs using existing Java taint analysis tools?*, with Ranjith Krishnamurthy, Goran Piskachev

2019 **POPL 2019 Distinguished Paper Award** for *Context-, Flow-, and Field-sensitive Data-flow Analysis Using Synchronized Pushdown Systems*, with Johannes Späth, Karim Ali

2018 **ESEC/FSE 2018 Distinguished Paper Award** for *Do Android Taint Analysis Tools Keep Their Promises?*, with Felix Pauck, Heike Wehrheim

2017 **ISSTA 2017 Distinguished Paper Award** for *Just-in-time Static Analysis*, with Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Justin Smith, Emerson Murphy-Hill

2012 **ISSTA 2012 Distinguished Paper Award** for *RefaFlex: Safer Refactorings for Reflective Java Programs*, with Andreas Thies

2008 **ISSTA 2008 Distinguished Paper Award** for *Racer: Effective Race Detection Using AspectJ*, with Klaus Havelund

### Fellowships and Other Awards (selection)

2022 **Alexander von Humboldt Foundation: Scout in the Henriette Herz Scouting Programme**

2021 Amazon Research Award, USD 60.000

2017 Oracle Collaborative Research Award, USD 100.000

2016 Distinguished Reviewer at ACM International Conference on Software Engineering (ICSE)

- 2016 Distinguished Reviewer at ACM International Conference on Automated Software Engineering (ASE)
- 2016 **German IT-Security Award, 1st Place, EUR 100.000**
- 2015 Oracle Collaborative Research Award, USD 100.000
- 2014 German IT-Security Award, 2nd Place, EUR 60.000
- 2014 Oracle Collaborative Research Award, USD 72.600
- 2013 **Fraunhofer “Attract” Award on Secure Software Engineering, EUR 2.494.640**
- 2013 Google Faculty Research Award (with Prof. Patrick McDaniel), USD 100.000
- 2012 **Emmy Noether Fellowship of the German Research Foundation (DFG), EUR 829.396**

### Supervision of Graduate Students and Postdoctoral Fellows

- Currently Advising five PostDocs and 17 Ph.D. students at Paderborn University and Fraunhofer IEM
- 2012 – 2023 In this period advised six PostDocs, graduated 12 and advised 28 in total Ph.D. students, advised over 80 Master students at Paderborn University and Fraunhofer IEM

### Recent Teaching Activities

- Since 2021 Master Course: Designing Code Analysis for Large-Scale Software Systems 2 (DECA 2)
- Since 2016 Bachelor Course: Secure Software Engineering
- Since 2016 Bachelor Course: Software Engineering Lab
- Since 2014 Master Course: Designing Code Analysis for Large-Scale Software Systems (DECA)
- 2017 – 2020 Master Course: Build It, Break It, Fix It
- 2017 – 2018 Master Project Group: SICS—Secure Integration of Cryptographic Software
- 2016 – 2017 Master Project Group: VisuFlow—Visualising Data Flows in Static Code Analyses

**Since I designed the DECA course, it spread around the world: it is also being taught, in person, by Joshua Garcia at University of California, Irvine, by Karim Ali at the University of Alberta, by Norbert Siegmund at Leipzig University and by Filip Krikava at Czech Technical University in Prague, and available on Youtube.**

### Organization of Scientific Meetings

- 2024 **Program Chair** for IEEE Secure Development Conference (SecDev)
- 2023 **Organizer** of Dagstuhl Seminar *23181 Empirical Evaluation of Secure Development Processes*, co-organized with Brendan Murphy, Sam Weber, Laurie Williams
- 2023 **Program Chair** for IEEE Secure Development Conference (SecDev), co-chaired with Tuba Yavuz
- 2021 **Program Chair** for Doctoral Symposium at ACM SIGSOFT Int. Symposium on Software Testing and Analysis (ISSTA), co-chaired with Wei Le
- 2018 **Program Chair** for ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)
- 2017 **General Chair** of Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), co-chaired with Wilhelm Schäfer
- 2017 **Program Chair** of International Symposium on Engineering Secure Software and Systems (ESSoS), co-chaired with Mathias Payer
- 2016 **Program Chair** of International Symposium on Engineering Secure Software and Systems (ESSoS), co-chaired with Juan Caballero
- 2013 **Program Chair** of SC: International Conference on Software Composition, co-chaired with Walter Binder

### Institutional Responsibilities

- 2023–2024 Chairman of the Board of Heinz Nixdorf Institute, Paderborn University
- Since 2016 Director for Software Engineering and IT-Security at Fraunhofer IEM

### Commissions of Trust

- 2017–present Steering Committee of Joint Mtg. of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)
- 2017–present Steering Committee of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)
- 2022–present Editorial Board, IEEE Security & Privacy
- 2021–present Editorial Board, ACM Transactions on Software Engineering and Methodology (TOSEM)
- 2019–present Editorial Board, Springer Journal on Empirical Software Engineering (EMSE)
- 2016–2019 Editorial Board, IEEE Transactions on Software Engineering (TSE)

### Memberships of Select Scientific Societies

- 2020–present Member of the IFIP Working Group 2.4 on Software Implementation Technology
- 2019–present ACM Distinguished Member



**Appendix: All on-going and submitted grants and funding of the PI (Funding ID)**

**Current grants**

Project title	Funding source	Amount (Euros)	Period	Role of PI	Relation to current ERC proposal
Proof-Carrying Services (SFB 901 On-the-fly Computing), Part 3	DFG	265.200	07/2019–06/2023	Subproject co-lead	Efficient static-analysis implementations
Automated risk analysis with respect to open-source dependencies (Transfer project)	DFG	500.100	07/2021–06/2024	Subproject co-lead	Static analysis, but for a very specific purpose (OSS dependencies)
Secure Integration of Cryptographic Software (SFB 1119 CROSSING), Part 3	DFG	296.900	07/2022–06/2026	Subproject co-lead	Static analysis for identifying crypto misuses
SAIL: SustAInable Life-cycle of Intelligent Socio-Technical Systems (SAIL)	MKW NRW	300.000	09/2023–08/2027	Principal Investigator	Software-engineering support for data-insentive software systems

**On-going and submitted grant applications**

No other applications

## C. Ten-Year Track Record: Eric Bodden

### C.1. Ten representative publications

- [1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, **Eric Bodden**, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Ocateau, and Patrick McDaniel. [FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps](#)  $\varnothing$  In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* pages 259–269. ACM, 2014.
  - ▷ The FLOWDROID framework introduced precise and efficient static taint analysis to the software engineering and security communities.
- [2] Li Li, Alexandre Bartel, Tegawende F. Bissyande, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, **Eric Bodden**, Damien Ocateau, and Patrick McDaniel. [IccTA: Detecting inter-component privacy leaks in Android apps](#)  $\varnothing$  In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 280–291. ACM, 2015.
  - ▷ This paper introduced precise inter-component analysis for Android apps.
- [3] **Eric Bodden**, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. [Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders](#)  $\varnothing$  In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 241-250. IEEE, 2011.
  - ▷ The pioneering paper to investigate the utility and limits of using dynamic analysis to aid static analysis in resolving reflection.
- [4] **Eric Bodden**, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba and Mira Mezini. [SPL<sup>LIFT</sup>: Statically analyzing software product lines in minutes instead of years](#)  $\varnothing$  In *ACM SIGPLAN conference on Programming language design and implementation (PLDI)* pages 355–364. ACM, 2013.
  - ▷ This groundbreaking work shows how to make static analysis of software product lines tractable.
- [5] Siegfried Rasthofer, Steven Arzt, and **Eric Bodden**. [A machine-learning approach for classifying and categorizing Android sources and sinks](#)  $\varnothing$  In *Network and Distributed System Security Symposium (NDSS)*, 2014.
  - ▷ The first paper to show that one can effectively identify taint sources and sinks using machine learning.
- [6] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali and **Eric Bodden**. [Boomerang: Demand-driven flow-and context-sensitive pointer analysis for Java](#)  $\varnothing$  In *30th European Conference on Object-Oriented Programming (ECOOP)*, 2016.
  - ▷ Shows that the long-standing problem of pointer analysis, which is non-distributive and thus computationally hard, can be decomposed into multiple, distributive sub-problems, efficient and precisely solvable.
- [7] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. [Mining apps for abnormal usage of sensitive data](#)  $\varnothing$  In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 426–436. ACM, 2015.
  - ▷ The pioneering work to use semantic features (data flows) in machine learning-based malware detection.
- [8] Sarah Nadi, Stefan Krüger, Mira Mezini, and **Eric Bodden**. [Jumping through hoops: Why do Java developers struggle with cryptography APIs?](#)  $\varnothing$  In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 935–946. ACM, 2016.
  - ▷ The first work to investigate the reasons for the OWASP-Top-2 vulnerability class: Cryptographic Failures
- [9] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and **Eric Bodden**. [Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques](#)  $\varnothing$  In *Network and Distributed System Security Symposium (NDSS)*, 2016.
  - ▷ The tool HARVESTER is capable of extracting secret communication from Android malware despite all static obfuscation and dynamic anti-analysis techniques, yielding close to perfect recall.
- [10] Stefan Krüger, Johannes Späth, Karim Ali, **Eric Bodden** and Mira Mezini. [CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs](#)  $\varnothing$  In *IEEE Transactions on Software Engineering (TSE)* 47 (11), pages 2382–2400.
  - ▷ Introduces a DSL for auto-generating static analyzers for crypto-API misuse validation.

## C.2. Patents

- [1] **Eric Bodden**, Christopher Goodfellow, and Howard Hellyer. Method and system for performance profiling of software. IBM, 2003, US patent no. 7765094, issued on July 27th, 2010.
  - ▷ Patents an automated profiling environment for Java virtual machines; strongly relates to Objective [E](#).
- [2] Siegfried Rasthofer, Marc Miltenberger, and **Eric Bodden**. Apparatuses, mobile devices, methods and computer programs for evaluating runtime information of an extracted set of instructions based on at least a part of a computer program. Fraunhofer, 2015, EU patent no. EP3029595B1, issued June 4th, 2022.
  - ▷ Patents our approach for harvesting runtime values from software applications (NDSS 2016).

## Awards and honors.

My research has frequently received the highest awards on the international and national level. In 2022, I was elected a *scout* in the Henriette Hertz Scouting Program of the **Alexander von Humboldt Foundation**. In this capacity, the foundation entrusts me to award three international rising stars an AvH scholarship single-handedly, only evaluated through an ex-post evaluation. In 2019, I was elected **ACM Distinguished Member at the earliest opportunity**, just after the required 15 years of professional experience in the computing field. In 2020, I was elected into the distinguished IFIP Working Group 2.4 on Software Implementation Technology. In 2014, as the only computer scientist, and as one of ten awardees from all over Germany, I was awarded the **Heinz-Maier-Leibnitz-Prize** by the German Research Foundation (DFG), the DFG’s highest honour for un-tenured scientists, awarded by the federal minister of research and education herself. I am also an **Emmy Noether Fellow** of the DFG, the DFG’s most competitive funding program for young researchers. I am one of the youngest researchers worldwide to have been awarded **seven ACM/IEEE Distinguished Paper Awards**. Four times I have been named **Distinguished Reviewer at TOSEM, ICSE and ASE**. I am a member of the **steering committee of our top tier conferences ESEC/FSE and ISSTA** and **associate editor** for the prestigious ACM Transactions on Software Engineering and Methodology (TOSEM) and Springer Journal on Empirical Software Engineering (EMSE).

In 2013, I was awarded funding for a **Fraunhofer “Attract”** research group, a Fraunhofer funding scheme that allows university researchers to establish a novel research group at a Fraunhofer institute. This funding scheme is highly competitive and resulted in a grant worth almost EUR 2.5M. In 2016, the Horst Görtz Foundation awarded me the first prize at the **German IT-Security Award**, with EUR 100.000 Germany’s highest privately funded research award. In 2014, I scored second place at this competition, receiving EUR 60.000.

But what makes me most proud is that I have also succeeded in teaching my own Ph.D. students as well how to excel in their research. This is evidenced not only by several Distinguished Paper Awards that they won jointly with me, but also by the fact that **eight of my Ph.D. students graduated with Summa cum Laude (so far 13 graduated in total)**, **four won the national Ernst Denert Software Engineering Award**, and **three won Paderborn University’s Doctoral Dissertation Award**.

## Proven leadership in providing research infrastructures essential to the worldwide static-analysis community.

My research group has been widely recognized for publishing and maintaining a **large number of high-quality tools and research infrastructures** such as program-analysis frameworks, benchmarks and data sets, as evidenced, for instance by my invitation to give the **opening keynote** for the 2022 IEEE International Working Conference on Source Code Analysis and Manipulation. Since 2016, my group and me have been the chief maintainers of Soot [28, 51], one of the **most widely used frameworks for the static analysis and transformation of Java and Android applications**. In 2014, we published FlowDroid [1], an extension to Soot for conducting efficient information-flow analysis of Android apps, along with the now widely used DroidBench benchmark suite. Both Soot and FlowDroid are currently used by **hundreds of research groups worldwide** [28, 51]. Soot’s successor SootUp is also seeing widespread adoption.

In 2018, we made available **PhASAR [40], a powerful static analysis framework for C/C++ code, based on the LLVM compiler infrastructure**. Already now, Phasar has a strong, worldwide user base. In 2019, we published **Boomerang/SPDS [46], a highly precise pointer analysis engine for Java**.

**We maintain these technologies so well that they even have widespread commercial deployments.** For instance, FlowDroid is now running as an app-vetting tool in one of the leading Android app stores. In 2020–2022, we teamed up with Contrast Security, to build the commercial static analysis offering **Contrast Scan** [↗](#), whose concepts are entirely built on our tooling. Likewise, PhASAR is has been integrated into an application security testing process within a worldwide leading telecommunications company. This experience, along with the related technical infrastructure, puts me into the unique position to be able to successfully realize the goal of SOSA despite its complex conceptual and technical challenges.

## References

- [1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Oceau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.
- [2] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal—a tool suite for automatic verification of real-time systems. In *International hybrid systems workshop*, pages 232–243. Springer, 1995.
- [3] 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. *Communications of the ACM*, 53(2):66–75, 2010.
- [4] Eric Bodden. *Verifying finite-state properties of large-scale programs*. PhD thesis, McGill University, June 2009. Available in print through ProQuest.
- [5] Eric Bodden. The secret sauce in efficient and precise static analysis: The beauty of distributive, summary-based static analyses (and how to master them). In *ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis (SOAP 2018)*, ISSSTA '18, pages 85–93, New York, NY, USA, 2018. ACM.
- [6] Eric Bodden, Patrick Lam, and Laurie Hendren. Partially evaluating finite-state runtime monitors ahead of time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(2):7:1–7:52, June 2012.
- [7] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. Spllift: statically analyzing software product lines in minutes instead of years. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 355–364, 2013.
- [8] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. *ACM SIGPLAN Notices*, 44(10):243–262, 2009.
- [9] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 55–66. ACM, 1991.
- [10] Maria Christakis and Christian Bird. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 332–343, New York, NY, USA, 2016. Association for Computing Machinery.
- [11] Patrick Cousot. Abstract interpretation. *ACM Computing Surveys (CSUR)*, 28(2):324–328, 1996.
- [12] Andreas Dann, Ben Hermann, and Eric Bodden. Modguard: Identifying integrity & confidentiality violations in java modules. *IEEE Transactions on Software Engineering*, 2019.
- [13] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. Just-in-time static analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSSTA 2017, pages 307–317, New York, NY, USA, 2017. ACM.
- [14] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.
- [15] Holger Giese, Thomas Vogel, Ada Diaconescu, Sebastian Götz, and Samuel Kounev. Architectural concepts for self-aware computing systems. In *Self-Aware Computing Systems*, pages 109–147. Springer, 2017.
- [16] Neville Grech and Yannis Smaragdakis. P/taint: Unified points-to and taint analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.
- [17] D. He, H. Li, L. Wang, H. Meng, H. Zheng, J. Liu, S. Hu, L. Li, and J. Xue. Performance-boosting sparsification of the ifds algorithm with applications to taint analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 267–279, 2019.
- [18] Dongjie He, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li, and Jingling Xue. Performance-boosting sparsification of the ifds algorithm with applications to taint analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 267–279. IEEE, 2019.
- [19] Dominik Helm, Florian Kübler, Jan Thomas Kölzer, Philipp Haller, Michael Eichberg, Guido Salvaneschi, and Mira Mezini. A programming model for semi-implicit parallelization of static analyses. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 428–439, 2020.
- [20] Dominik Helm, Florian Kübler, Michael Reif, Michael Eichberg, and Mira Mezini. Modular collaborative program analysis in opal. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 184–196, 2020.
- [21] Markus C Huebscher and Julie A McCann. A survey of autonomic computing—degrees, models, and



- applications. *ACM Computing Surveys (CSUR)*, 40(3):1–28, 2008.
- [22] Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. Data-driven context-sensitivity for points-to analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.
- [23] Neil D Jones, Carsten K Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.
- [24] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*, pages 422–430. Springer, 2016.
- [25] John B Kam and Jeffrey D Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.
- [26] Kadiray Karakaya and Eric Bodden. Two sparsification strategies for accelerating demand-driven pointer analysis. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2023. To appear.
- [27] Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu. *Self-Aware Computing Systems*. Springer, 2017.
- [28] Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, October 2011.
- [29] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.
- [30] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. Precision-guided context sensitivity for pointer analysis. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.
- [31] Francesco Logozzo. Static analysis for security at the facebook scale. In *CurryOn*, 2016.
- [32] Travis Lowdermilk. *User-centered design: a developer’s guide to building user-friendly applications*. O’Reilly Media, Inc., 2013.
- [33] Jingbo Lu and Jingling Xue. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.
- [34] Magnus Madsen and Anders Møller. Sparse dataflow analysis with pointers and reachability. In *International Static Analysis Symposium*, pages 201–218. Springer, 2014.
- [35] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. From datalog to fix: A declarative language for fixed points on lattices. In *ACM SIGPLAN Notices*, volume 51, pages 194–208. ACM, 2016.
- [36] Mike Pittenger. Open source security analysis: The state of open source security in commercial applications. *Black Duck Software, Tech. Rep*, 2016.
- [37] Ganesan Ramalingam. On sparse evaluation representations. *Theoretical Computer Science*, 277(1-2):119–147, 2002.
- [38] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspán, Emma Soderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 598–608. IEEE, 2015.
- [39] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1-2):131–170, 1996.
- [40] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. Phasar: An inter-procedural static analysis framework for c/c++. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 393–410, Cham, 2019. Springer International Publishing.
- [41] Qingkai Shi, Rongxin Wu, Gang Fan, and Charles Zhang. Conquering the extensional scalability problem for value-flow analysis frameworks. Technical report, 2019.
- [42] Qingkai Shi and Charles Zhang. Pipelining bottom-up data flow analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 835–847, 2020.
- [43] Nishant Sinha and Chao Wang. Staged concurrent program analysis. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE ’10*, page 47–56, New York, NY, USA, 2010. Association for Computing Machinery.
- [44] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: Context-sensitivity, across the board. *SIGPLAN Not.*, 49(6):485–495, June 2014.
- [45] Johannes Späth, Karim Ali, and Eric Bodden. Ideal: Efficient and precise alias-aware dataflow analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.
- [46] Johannes Späth, Karim Ali, and Eric Bodden. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, 3(POPL):48:1–48:29, January 2019.
- [47] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. Inca: A dsl for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software*

- Engineering*, pages 320–331, 2016.
- [48] Walid Taha. *Multi-stage programming: Its theory and applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.
- [49] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '97, page 203–217, New York, NY, USA, 1997. Association for Computing Machinery.
- [50] European Union. Cyber resilience act. <https://digital-strategy.ec.europa.eu/en/policies/cyber-resilience-act>, 2022.
- [51] R Vallee, P Co, E Gagnon, L Hendren, P Lam, and V Sundaresan. Soot—a java bytecode optimisation framework. In *Proceedings of Cascon'99*, pages 125–135, 1999.
- [52] David Van Horn and Matthew Might. Abstracting abstract machines. In *ICFP*, volume 45, pages 51–62. ACM, 2010.
- [53] Guannan Wei, Yuxuan Chen, and Tiark Rompf. Staged abstract interpreters: Fast and modular whole-program analysis via meta-programming. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.
- [54] Christian Wimmer and Thomas Würthinger. Truffle: a self-optimizing runtime system. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 13–14, 2012.
- [55] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 187–204, 2013.

# Part B2 — The Scientific Proposal

## A. State-of-the-art and objectives

### A.1. Aims, Objectives and Challenges

By introduction of the planned EU’s Cyber Resilience Act [87], software developers will soon have to use software assurance techniques on a large scale. Static program analysis, including variants such as abstract interpretation [18] and formal software verification [3, 24], is *the* enabler for automated reasoning about a software’s code base. In this work we focus on static *data-flow* analysis, the most widely spread technique for static application security testing.

With commercial static-analysis offerings such as Micro Focus, VeraCode, Synopsys, CheckMarx, GitHub, etc., just to name a few, having become commonplace, one might think that static analysis is ready to be adopted on a large scale. And in fact, decades of static-analysis research have yielded the discovery of novel algorithms, data structures and design principles that make static analyses more precise and scalable than ever before [2, 49, 54, 56, 60, 65, 69, 73, 76, 79, 81, 82, 94].

Yet, studies show that it is key to adapt static analyses properly to the intended deployment context [89]. If *not* adapted well, developers don’t find the static analysis tools useful, frequently find them even hindering their work [5, 16, 37]. In particular, many analyses slow down rather than assist development [16]. They report large sets of false warnings that distract developers from actual bugs and vulnerabilities, which the analyses also often miss. They often run so long that when results are reported they are already outdated [51].

It is therefore a great foundational and practical challenge to build static analysis tools such that they are simple enough to be used not only by experts but by non-expert developers, such that they can likewise successfully and constructively interact with these tools. SOSA will rid the world of this challenge, paving the way to large-scale adoption of static analysis, by introducing automation to static-analysis customization and optimization.

In the static-analysis community, the challenges of precision, recall and performance have long been a major focus of the research. And in fact, static-analysis research has led to a large number of what I call “one-off optimizations” of static analyses, e.g. [22, 31, 48, 50, 74]. In virtually all of these scientific publications, the optimizations have been applied to one concrete instance of one static analysis, implemented in one specific static analysis tool. Moreover, to be useful to software developers, these developers would need to be aware of, identify and successfully apply such optimizations and adaptations on their own.

**Yet, with current algorithms and tools, as I know from first-hand experience in many industry projects in this space, the optimal adaptations may even take static-analysis experts weeks if not months to implement.** And in many cases even these experts are having a hard time finding an optimal static-analysis customization at all. The challenge here is that many of the required adaptations heavily depend on the class of analyzed programs and hence are not available out of the box in current program analysis frameworks, nor is it clear upfront which combination of adaptations will yield the best result for the context at hand.

**Figure 1 shows this *current* state of the art in static program analysis:** A software developer deploys and configures the analysis tool and then runs it using a general-purpose execution environment, for instance a Java Virtual Machine, or these days frequently also a Datalog solver. This will result in initial analysis reports, usually at first containing many false warnings and missing many actually essential bugs and vulnerabilities. **Many developers will disappointedly abandon the tool at this stage, the more perseverant ones will now engage in a trial-and-error loop in which they attempt to reconfigure parts of the analysis and maybe even alter parts of the analysis implementation.** The human is *in* the loop. In the past years I have led a number of industry-financed projects, ranging from SMEs with 20 employees to Fortune 100 companies with dedicated SAST research teams, and all were facing the same daunting challenges.

**The goal of SOSA** is to remedy the situation through a radical paradigm shift:

**SOSA’s main research hypothesis** is that one can *automatically generate* precise and efficient static analyses of software systems by making static analysis self-aware and self-optimizing.

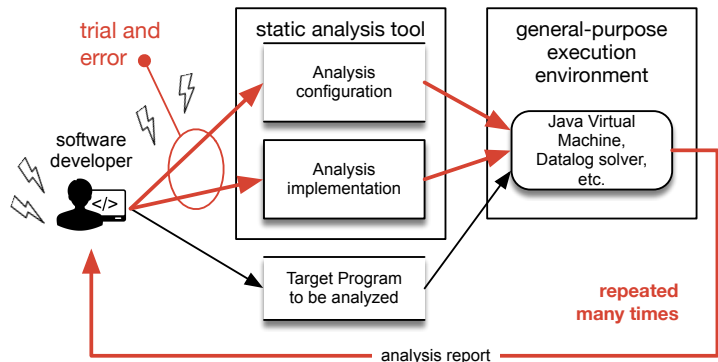


Figure 1: Current state of the art

**This paradigm shift is groundbreaking: never before have researchers regarded complex program analyses themselves as the primary object of analysis.** Figure 2 illustrates the envisioned solution. The circled letters in the figure relate to objectives and research areas that I describe further below. With SOSA, software developers must no longer adapt the analysis at all because the analysis itself becomes self-aware and self-optimizing. **Developers instead declaratively define what to analyze but not how the analysis should execute. They then automatically receive precise and useful analysis reports for the target programs that they provide.** The human is taken out of the loop. The analysis adapts to—and optimizes itself for—these programs automatically: a just-in-time analysis engine analyzes not only the target program in question *but also the analysis itself* and how well it succeeds in operating on that program. This is enabled by a specialized execution engine that is aware of the domain-specific<sup>1</sup> concepts that define the static analysis as well as a special analysis profiler that allows the analysis to uncover, e.g. where it loses precision or performance. Principles from autonomic and self-aware computing will allow the analysis engine to self-adapt towards an execution optimal for the given usage context.

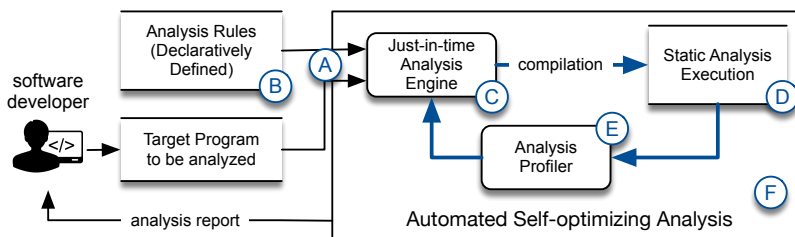


Figure 2: Envisioned solution from the user’s point of view, (A)–(E) relate to objectives / research challenges described below.

With this shift, SOSA will radically change the way in which static analyses are expressed, implemented, executed, optimized and maintained. For the first time, the project will introduce static analyses that are inherently self-optimizing. Thus, the goal of SOSA is to research novel concepts, methods and algorithms that will enable and make succeed such automated self-optimization. In B1, I have given examples of what such optimizations could look like for taint analysis and buffer overflow detection. Current static analyses preclude automating such optimizations because the analyses are usually implemented in an imperative style. This not only defines *what* the analysis should detect but also defines *how* this detection should be conducted, erasing any degree of freedom.

With this shift, SOSA will radically change the way in which static analyses are expressed, implemented, executed, optimized and maintained. For the first time, the project will introduce static analyses that are inherently self-optimizing. Thus, the goal of SOSA is to research novel concepts, methods and algorithms that will enable and make succeed such automated self-optimization. In B1, I have given examples of what such optimizations could look like for taint analysis and buffer overflow detection. Current static analyses preclude automating such optimizations because the analyses are usually implemented in an imperative style. This not only defines *what* the analysis should detect but also defines *how* this detection should be conducted, erasing any degree of freedom.

**SOSA’s paradigm shift will be enabled by novel mechanisms to define static analysis rules in a declarative manner. This will enable analyses that can be successfully operated and customized by software developers who are no experts in static analysis.** This declarative nature of the analysis definition requires these developers to only define *what* the analysis should detect and report, while the “*how*” will be deliberately left to be determined by the self-optimizing analysis engine on its own. Based on that definition, SOSA instead *automatically synthesizes* a highly customized and optimized static-analysis implementation, specifically tailored to the problem at hand. In particular, the automated adaptation and optimization will consider multiple static-analysis optimizations in combination, and it will assess the effect that this combination has on end-user goals such as the precision, recall and running time of the analysis. Moreover, the solution will have the ability to apply optimizations not just ahead-of-time, i.e., prior to the execution of the analysis, but also just-in-time, i.e., while the analysis is running. This will allow the analysis to re-adapt and thus further optimize its own implementation based on an introspection into its own execution. This is what I mean when I speak of self-optimization and this resembles SOSA’s main goal.

SOSA will even investigate a **compilation-based approach in which SOSA’s tool chain, given a program under analysis and a declarative static-analysis definition as an input, produces a program-analysis instance that, when executed, uses data structures and parallelized accesses to those data structures which are optimized to the execution at hand.** Enabling such self-adaptivity and self-optimization requires one to design and implement the analysis according to a completely novel engineering methodology, in which the analysis engine must be enabled to systematically reason about the analysis itself, and its own execution.

I call the resulting concept “self-optimizing static program analyses”. With it, SOSA will enable static analysis and related fields such as software verification and validation to solve the most daunting research challenges that currently stand in the way of a widespread industry adoption. Particularly, self-optimizing static analyses will yield a plethora of practical advantages over the existing state of the art: **Since self-adaptive analyses self-optimize, the analysis definitions no longer require manual optimizations themselves, yielding analysis definitions that are easy to understand and evolve also by non-expert software developers.** Developers will merely define *what* an analysis should detect but no longer *how* the analysis should conduct this detection. SOSA’s solution will be built in such a way that, where helpful, it will provide the developer feedback on domain-specific bugs or optimization opportunities *directly with respect to the declarative analysis definition.* No existing analysis approach supports such a feature. Further, because the analysis optimizations are domain-specific (details below), because they combine sets of otherwise individual one-off optimizations, and because they are adapted to the target program, **it is likely that in realistic application scenarios one might**

<sup>1</sup>Note that when I write about “domain-specific” concepts or optimizations, the domain I refer to is the one of *static analysis* itself.



gain performance improvements of several orders of magnitude, along with improved precision. This is particularly true when paired with the compilation scheme for static analyses that I envision, which has the potential of yielding an additional, significant performance boost. **By enabling software developers to make optimal use of static analyses without manual intervention, SOSA will allow developers to adopt static analysis on a large scale and thus will help secure millions of software systems.**

**But SOSA will also boost progress in science itself.** With self-optimizing static analyses we researchers will, for the first time, obtain a testbed that allows us to assess in detail the relative effects that not just individual optimizations of static analyses but also their combinations have when applied to different kinds of static analyses, applied to various kinds of programs under analysis. For the first time, this will allow researchers to map the landscape of program-analysis optimization, thereby boosting research in the field, and in nearby fields such as abstract interpretation and formal verification. The open-source artifacts I will produce in SOSA will allow researchers worldwide to comparatively evaluate existing and upcoming static-analysis techniques.

While SOSA seeks to address real-world challenges, it cannot be funded by private corporations because of the associated risks and also because any exploitation of the project’s results will not be able to commence until after several years into the project.

To achieve its goal, SOSA must solve a number of key research challenges, which I outline next. I sort these challenges according to six key objectives (A)–(F). These objectives induce related work areas, which are appropriately labeled in Figure 2.

**Objective (A): Joint intermediate representation of static analysis and target program** By design, the declarative definition mechanism must *not* limit *how* a static analysis executes. Hence, to enable the analysis to reason about how it should execute, and to optimize this execution, one requires another, separate representation that facilitates such reasoning—effectively a dedicated intermediate representation (IR). The final solution will, in fact, likely comprise multiple such IRs. A simple high-level IR could allow one to optimize static-analyses *ahead of time*, i.e., prior to their execution. For instance, recent work by Shi et al. [72] has shown that one can gain impressive speedups by finding a combined evaluation strategy for entire *sets* of taint-flow properties, where the analysis evaluation shares computation among these properties. While such optimizations can be applied ahead of time, without knowledge of the program under analysis, others do require such knowledge. For this reason, and to allow the analysis to reason about and to optimize its own execution, the IR must *jointly* include information about both the analysis and the target program, as well as the analysis’ execution on that target program. For instance, as we and others showed in earlier work [8, 12, 36, 50, 53, 75, 77], one can greatly speed up some static analyses by executing them in consecutive, increasingly precise stages. In such a setting, the analysis within each stage depends on analysis results that have already been obtained by previous analysis stages, applied to the same program. Another, more recent work by He et al. on SparseDroid [31] shows similar potential: given a concrete analysis problem, in this case encoded in the IFDS framework [67], and given a program under analysis, SparseDroid determines which flow-function computations do not have any effect on certain analysis values in certain parts of the program, so that their execution on these values can be safely skipped. As IR, the authors use a specialized sparse value-flow graph that encodes the semantics of the analysis as it is applied to the specific program under analysis—a special form of a *joint* IR. The challenge here will be to find an optimal IR, or a set of interrelated IRs, that allow for powerful optimizations, including automated staging, sparsification and parallelization, enabling both automated ahead-of-time and just-in-time optimizations.

**Key Research Challenges:** Concepts and methods for a joint intermediate representation combining both, aspects of the analysis and the target program, enabling an automated derivation of optimized analysis variants that implement techniques such as staging, sparsification, parallelization, etc.

**Objective (B): Declarative definition of static analyses in domain-specific terms** Current static analyses are largely implemented in imperative languages, particularly Java or C/C++, or in a general-purpose logic programming language such as Datalog. Imperative implementations define not only *what* an analysis should detect and report but also *how* this detection should be conducted. While Datalog, as a declarative language, allows the execution strategy to be chosen rather freely, a Datalog solver’s internal reasoning is limited to Datalog constructs. A generic datalog solver has no knowledge of what a static analysis is and how its execution could be optimized in domain-specific terms. To allow for self-adaptation and optimization, SOSA will thus implement a radical paradigm shift, researching a *domain-specific* declarative analysis definition language (textual or visual), backed by an automated engine that itself decides on the best analysis strategy. Analysis rules expressed in the language will translate into the above IR. A key challenge is to design the definition language such that it has exactly the right level of expressiveness: it must be expressive enough to cover a reasonably broad set of possible static analyses, yet at the same time its expressiveness must be restricted enough so as to allow for powerful automated domain-specific optimizations. General-purpose languages are too expressive: they are Turing-complete, thus defying many optimizations. And even Datalog, while not Turing-complete, has no information about the domain of static analysis. This not only defies any domain-

specific optimizations but also precludes the analysis engine from providing to the software developer error messages or optimization hints that are specific to the static analysis at hand. We instead envision a solution in which SOSA’s solution could provide hints such as: “The requirement to report *all* program paths to the bug is responsible for 80% of the analysis time. If knowing a single path to the bug is sufficient, this can cause significant savings without loss of precision.” The solution should be able to present such explanations and optimization hints to the user *in terms of its declarative definition*.

**Key Research Challenges:** Concepts and methods for an easy-to-use *domain-specific* declarative language for expressing static-analysis rules, along with mechanisms that give useful explanations and optimization hints with respect to these rules.

**Objective ©: Self-adaptivity and self-optimization through self-awareness** One of the most radical ideas of SOSA is to make, for the first time, static analyses self-aware of their own existence, their own purpose and goals and particularly their own execution. Using this self-awareness, the analysis will then be able to apply to itself powerful, domain-specific optimizations, oftentimes even combining sets of otherwise individual (formerly “one-off”) optimizations, and even specializing these optimizations with respect to the target program that is analyzed. To enable such reasoning about itself, one must choose the right abstraction of the analysis. One key challenge will thus be to derive—ideally automatically uncover—a model of the analysis itself, that allows the analysis to autonomously reason about where it succeeds and where it fails in terms of its optimizations goals, and why, so that it can adapt its optimization strategies based on those insights. I plan to establish this self-awareness using data collected from a dedicated static-analysis profiler, details of which I outline further below (Objective ⑤). Another challenge is, however, how to establish self-awareness based on this data. While there exist well-established mechanisms for autonomic systems such as MAPE-K, and the lesser known LRA-M [26, 34], it is currently unclear, which of these, or which related approaches, are best suited for the task at hand. It is an open challenge to assess their utility in this particular context, and how to instantiate and potentially adapt them to optimally fit the goal of SOSA. Another challenge I foresee is the communication of sensible choices towards the software developer. Given that multiple, partially conflicting optimization goals might exist (e.g. performance, precision, memory), the optimization might frequently yield pareto-optimal solutions, which could, for instance, be parameterized by user-selected priorities. For instance, the analysis might uncover that one might speed up its own execution dramatically but under a slight loss of precision. Or, while an analysis executing as part of an integrated development environment must return the first results with maximal efficiency and precision at least for the developer’s current scope [22], one may afford a longer running time, also increasing recall, upon analyzing a merge request in a continuous-integration pipeline. Irregular acceptance tests might even afford long-running analyses. How can one best communicate to the user such tradeoffs between pareto-optimal solutions? How can one allow the user to make an informed judgement? SOSA will be the first project to systematically address such questions.

**Key Research Challenges:** Concepts and methods for a mechanism that enables self-awareness and autonomic decision making in a static analysis with respect to optimizations of its own execution. The derivation of an automated optimization engine which learns from profiled analysis runs in order to better optimize future runs (ahead-of-time) or even the remainder of a current run (just-in-time). Further, an extension of this engine to communicate to the user sensible choices of pareto-optimal solutions.

**Objective ④: Automated compilation of static analyses** Current static analysis algorithms work through an abstract *interpretation* of the target program, i.e., they deconstruct and interpret each program statement one by one, which in itself is very time consuming. In SOSA I will thus set myself and my team the tough challenge of designing, implementing and evaluating a *compilation scheme* that avoids slow interpretation by compiling jointly both the static analysis and the target program. This compilation may amount to allocating analysis and program-specific data structures for efficient data access and management, but we also plan to investigate implementations backed by special programming languages and runtimes. For instance, a previous project with an industrial partner revealed that one can obtain significant speedups, sometimes of multiple orders of magnitudes by implementing algorithmics in Rust instead of Java, which avoids runtime overhead due to the repeated garbage collection of short-lived objects. (Rust runs without the necessity for garbage collection.) Another surprisingly successful low-level optimization arranges the analysis’ internal state in specially optimized data structures, e.g. by specially selected types and combinations of sets and maps. Further, SOSA will seek to automatically generate a parallelized implementation to make best use of multi-core architectures. We will further investigate to what extent such compilation and low-level customization can be performed just-in-time, or at least in a profile guided manner (see Objective ⑤), i.e., where the compilation is optimized based on previously profiling information collected from previous runs.

**Key Research Challenges:** Concepts and methods for an efficient compilation engine for a static analysis of a given program. The scheme will include the allocation of optimal data structures for efficient access to analysis state, as well as the sound parallelization of such accesses.

**Objective (E): Domain-specific profiling of static analyses.** Self-adaptivity can yield self-optimization only if the analysis can obtain the required situational awareness. This raises the challenge of obtaining detailed profiling information about program-analysis runs, and importantly on a level of abstraction that allows the self-optimizing analysis to identify potential also for domain-specific optimizations. A key technical challenge will be to establish and maintain a rigorous traceability between three artefacts: (1) the executing (potentially compiled) analysis on the one side, and both (2) the original target program and (3) the analysis rule definitions on the other side, both from which the static analysis was derived. Current static-analysis frameworks face a similar traceability challenge when it comes to tracing reports in terms of the analysis’ intermediate representation back to the source code. However, in virtually all existing frameworks, such traceability is established by manual means, which makes this state of the art brittle and error-prone. The automation envisioned in SOSA raises both an opportunity and a key challenge, an opportunity because the analysis automation can maintain required trace links automatically, a challenge because we seek to link back information about the runtime execution to all of the three artifacts mentioned above, plus the analysis’ internal IR. Another key challenge will be to automatically identify the root causes of performance and precision degradation (along with memory blow-ups) during static analysis. Such degradation often arises when analyses seek to *soundly over-approximate* at certain hard-to-analyze program locations, as this can cause even exponential blowups in the size of the analysis abstraction [49]. Yet, often there is not a single program location that can be made out as the “culprit” of this degradation because frequently also the context of this code location matters, e.g.: Are there loops surrounding the location? Are there recursive calls or data structures involved? In problematic cases, the answer to these questions is frequently “yes”. The challenge is then to identify the scope of the performance/precision-degrading code, and related static-analysis rules, so that the automation has a chance of deriving suitable optimizations.

**Key Research Challenges:** Concepts and methods for a *domain-specific* profiling tool for static-analysis runs, providing traceability between the generated analysis, the original static analysis and target program, and the analysis’ own intermediate representation. Further, a means to identify root causes of performance and precision degradation (and memory blow-up) as well as the scope of the target program’s code and the static-analysis rules that cause this degradation.

**Objective (F): Systematic assessment of static-analysis optimizations** Currently we are lacking a systematic assessment of the extent to which (existing and yet to be invented) optimizations for various kinds of static analyses are applicable also to other analyses and under which circumstances they are applicable at all. The final objective of SOSA is thus to provide the capability to conduct such an assessment, and to also then indeed execute such assessment with respect to a sizeable number of optimizations that are known to date and that are published during the course of the project. Examples for such optimizations are various kinds of staged analyses, sparsification of static analyses, optimal choice of contexts, different policies on when to merge static-analysis information, different heap representations, etc. The assessment should evaluate how these optimizations, both individually and in various combinations, affect the analysis result in terms of precision, recall and analysis time, possibly even in terms of comprehensibility in terms of the analysis result. The latter is important because complex results might be correct, yet hard to identify as such by developers [5]. The assessment should further yield guidance outlining what kinds of programs can usually best be analyzed using which kinds of optimizations and using which combinations of optimizations. An interesting aspect is here whether and to what extent multiple analyses may cancel each other out or even support one another. This information not only greatly advances the state of the research but can also be used to inform heuristics in the automated optimization. To maximize the benefit for related disciplines such as abstract interpretation and formal verification, an additional explicit objective is to evaluate which kinds of optimizations might be beneficial also in those contexts.

**Key Research Challenges:** A systematic assessment of a wide range of static-analysis optimizations and the joint effects of such optimizations, also w.r.t. different classes of target programs. An assessment of which optimizations might benefit also related disciplines such as abstract interpretation and formal verification.

## A.2. State of the art

In the following I summarize the relevant state of the art in the areas of autonomic and self-aware computing, static analysis, compilers and virtual machines, and how SOSA will use and advance this state.

### A.2.1. Autonomic and self-aware computing

While this project focuses on program analysis, compilation and optimization, to realize the coordination of the self-adaptation and optimization I will draw on research results from the areas of autonomic and self-aware computing [34, 45]. Here I will make use of the extensive experience that I have gained as a part-project leader within the German Research Foundation’s (DFG) Collaborative Research Center 901 *On-the-fly computing*, in which, together with more than a dozen PIs, we jointly research how to build techniques and processes for automatic on-the-fly configuration and provision of individual IT services out of base services that are available

on world-wide market. Within this center I am collaborating with Prof. Gregor Engels, one of the leading software engineering researchers for self-adaptive systems [20]. Prof. Engels has agreed to also collaborate with me on SOSA. The topic of intelligent technical systems is also the core topic of the interdisciplinary Heinz Nixdorf Institute at which I am employed. In preparation of this proposal I have investigated specifically the well-known MAPE-K architecture but also lesser known related architectures such as a model-based learning and reasoning loop (LRA-M) [26, 34]. While MAPE-K is designed to realize self-adaptive systems by acting on environmental information using pre-defined rules, LRA-M makes the system self-aware by learning from environmental information over time a model of itself and the environment. This model is then further reasoned about to guide actions strategically. In SOSA, such actions will optimize the analysis execution itself, which makes the system self-adaptive.

### A.2.2. Optimizing program analysis through staging, sparsification and parallelization

SOSA seeks to automate the application of optimizations to program analyses. Not all but many optimizations fall into the categories staging, sparsification and parallelization. In “staging” of a static analysis, the analysis as a whole is sub-divided into individual, typically subsequent stages. In previous work I and others have used such staging to obtain significant speedups of the analysis execution [8, 12, 36, 50, 53, 75, 77]. The general idea is to introduce early stages that conduct very efficient pre-analyses whose results can then help avoid unnecessary computations within more expensive analyses at later stages. My Clara framework [12] for the optimized execution of static typestate analysis, for instance, uses first a syntactic check whose cost is linear in the size of the program, which is followed by a flow-insensitive pointer analysis (with cubic complexity), which itself is followed by a comparatively expensive flow-sensitive data-flow analysis. In experiments we were able to show that the introduction of the first two stages can save the majority of the analysis time. But coming up with the right design for the staging, and implementing it, is currently a task for experts. With SOSA, for the first time we will be able to apply such staging optimizations (and others) *automatically* to static analyses.

In recent years, more and more researchers, including myself, have explored the capabilities of *sparse* data-flow analysis [15, 41, 54, 64, 79]. While typical static data-flow analyses operate directly on the program’s control-flow graph or a basic-block graph, sparse analyses instead operate on a (comparatively smaller) value-flow graph composed of definition-use chains. Sparse analyses are related to SOSA, as they create an intermediate representation—the value-flow graph—that combines elements of the analyzed program (its variable definitions and uses) with a specific program analysis (a definition-use-analysis). In that sense, the value-flow graph can be seen as one simple example of the kind of intermediate representations that one might envision useful in the context of my planned IR (Objective  $\textcircled{A}$ ). In SOSA I will build on this state of the art, deriving more versatile representations and sparseness-based optimizations.

A number of works have also sought to *parallelize* static program analysis. Static data-flow analysis is a problem that is not trivial to parallelize, due to the intrinsic dependencies in its computation, e.g. a dependence on program order in flow-sensitive analyses. Recent work by Qingkai and Zhang [74] has described a way to relax one such dependence, namely calling dependence, which in turn can then lead to significant speedups obtained by a more parallel execution. In SOSA I will seek to detect automatically when such parallelization is possible to then automatically generate an appropriately parallel execution schedule. A new programming model recently proposed by Helm et al. seeks to enable parallel static data-flow analyses [32] and can thus inform the design of SOSA’s analysis definition language and runtime mechanisms.

### A.2.3. Conceptual analysis frameworks

Current top-of-the-line program analyses are largely flow, field and context-sensitive. Virtually all flow and context-sensitive static analysis follow either the so-called *call-strings approach* or the *functional approach*, once jointly introduced by Sharir and Pnueli [71]. The call-strings approach has the advantage that it is very general: it can be applied to any static data-flow analysis that fits the monotone framework [40]. Yet, it has the drawback that it often analyzes callee procedures more frequently than necessary, thus unnecessarily wasting computation time [9]. Moreover, one must bound the amount of context the analysis uses, and choosing the bound poorly might jeopardize performance and precision [42]. The functional approach has the advantage of providing unlimited context-sensitivity (hence no bound is required), yet is only applicable to analysis problems whose flow functions distribute over the chosen merge operator—so-called IFDS or IDE problems [67]. In cases where the analysis problem fits such a framework, one has the advantage that one can compute precise solutions (equivalent to the theoretically optimal but generally uncomputable MOP solution [42]), and moreover that highly efficient solution algorithms exist. Interestingly, such distributive program analysis have been shown to particularly benefit from sparsification [31, 41]. Moreover, in recent work, my team and I were able to show that it is sometimes possible to decompose such analysis problems that are actually not distributive, for instance, pointer analysis or general constant propagation, into sub-problems that are, in fact, distributive, and thus can be efficiently solved using IFDS or IDE solvers, assuming some extra analysis code that then composes the results of individual distributive computations to the final analysis result. As we were able to show in a recent, award-winning work, such an approach can provide significant speedups and gains in precision compared to previous approaches [80]. One goal of the optimization engine I seek to develop within SOSA will be to identify



the potential for such a decomposition of analysis problems automatically. With SOSA, one will thus be able to define program analyses without requiring any knowledge of these frameworks and their properties. Instead SOSA will automatically decompose the analysis problem as needed and choose the optimal analysis framework for each sub-analysis. The problem of understanding and optimizing static analyses automatically is a very challenging one. Thus it helps to apply “divide and conquer”, which is eased by a modular implementation of static analyses that aid such decomposition.

#### A.2.4. Modularization of static analysis, fostering parallel execution

Chord, whose development is discontinued, was one of the first static-analysis frameworks that attempted to encourage modular and compositional static analysis definitions [58, 59]. Datalog rules are compositional by definition, which is why also Doop [14] allows for modular analysis definitions. Helm et al., former colleagues of mine, present a domain-specific, rather principled approach to decomposing static program analyses [33]. Their goal is not so much a performance optimization of the static analysis but rather to obtain a lightweight design in which an otherwise large and monolithic analysis is decomposed into smaller sub-analyses that all support one another, and whose execution is automatically scheduled in an appropriate fixed-point iteration. In this design, the sub-analyses are effectively composed to a reactive system, implemented using a so-called blackboard system [61]. Where possible, the sub-analyses execute in parallel and await one another where required. While the analyses themselves are written in an imperative style (in Scala code), scheduling constraints and data exchange between analyses are largely defined in a declarative manner. For instance, the scheduler needs to know if an analysis is pessimistic, i.e., computes its final solution starting from a sound approximation, or optimistic, starting from an unsound one. This is because to guarantee soundness and termination, inter-dependent pessimistic and optimistic analyses must not execute in parallel. In Helm’s approach, the decomposition and declarative annotation were performed manually by the authors. Within SOSA I plan to design the DSL (Objective  $\textcircled{B}$ ) such that an appropriate decomposition can be conducted *automatically* and where one can also infer properties such as “the analysis is an optimistic one” automatically from its definition. For instance, a may-points-to analysis is likely identifiable as an optimistic analysis because its meet operator is defined to move towards a more approximative solution (starting from a potentially unsound one). These and other properties will then be exploited not only to find a correct and optimal scheduling of sub-analyses but also to trigger automated optimizations of the sub-analyses themselves. Prof. Mira Mezini, who with her research group invented this modular approach, has agreed to collaborate with me on SOSA.

#### A.2.5. Declarative approaches to static program analysis

While researchers have greatly advanced the state of the art in program analysis through the introduction of declarative analysis definitions mechanisms, they are all insufficient for what I propose in this work. The static-analysis framework Doop [14] is one of the most widely known and used approaches in the field, I myself have quite successfully used Doop in recent work [19]. Doop implements its static analyses not in a general-purpose imperative programming language such as Java or C/C++, but instead in the logic programming language Datalog. This allows users to define and maintain static-analysis rules that are simple to write, read and reason about. The declarative nature of the language also means that in general users need not be concerned with how the rules are evaluated, as this decision is entirely left to the Datalog engine.

To enable fast execution, there is now a lot of support for automated optimizations on the level of the Datalog language itself. Doop is frequently used in combination with highly optimized Datalog solvers such as LogicBlox [1] or Souffle [39]. The latter is a Datalog solver specifically designed for the purpose of supporting static analyses: it translates the datalog rules into C++ code implementing a highly optimized Datalog solver for the particular rule set at hand. To some extent Souffle thus appears similar to what I propose here. Yet, a limitation of Souffle and other Datalog engines quite significant to SOSA is that they have no domain-specific knowledge of what a static analysis is and how it can be optimized by exploiting special properties of the analysis. Instead they can only apply generic optimizations on the level of the Datalog rules themselves.

Researchers have recognized this and have started to add some first domain-specific constructs: A particular limitation of Datalog is that it lacks the ability to express lattices. Lattices are essential to static analyses because their use allows one to assure not only termination of a static data-flow analysis but also allows one to conduct a systematic merging of analysis information at control-flow merge points. This is particularly important for analysis problems that have infinite domains such as constant propagation, which reasons about all integers. The analysis computation cannot terminate for such problems unless lattices are used. Because pure Datalog lacks lattices, one cannot use it to conduct such analyses. Flix [55] is an extension of Datalog with certain primitives that *do* allow analysis writers to express lattice-based computations. Recently, also the Souffle engine has been extended with similar support [29]. Opposed to Flix, Souffle has been used to analyze real-world programs. Also, it supports parallel rule evaluation. Hence all in all, Souffle could certainly be an interesting technology base for SOSA. Yet, currently Souffle is limited to executing general programs written in (an extension of) the general-purpose language Datalog. This limits the domain knowledge it can extract from the static-analysis definitions. To enable powerful domain-specific extensions, the tool and its input language would thus require significant extensions.

CodeQL [27] is the domain-specific rule definition language of GitHub’s static analysis tool LGTM [28]. While CodeQL is largely declarative, it is also relatively expressive and low-level. This makes it harder to establish self-awareness, and to implement domain-specific optimizations in general. As confirmed with Max Schaefer from GitHub, for this reason the LGTM tool, like Doop, also leaves most optimizations to its Datalog engine.

Another relevant approach is IncA [83], a domain-specific language (DSL) for the definition of *incremental* static program analyses. IncA shares with SOSA the spirit that program analyses can benefit from a declarative definition because this relieves developers from coding and hand-optimizing the analyses manually. We will investigate the IncA DSL as a potential starting point for the DSL that we seek to build in SOSA. IncA’s current language design looks promising but it may be too limited for the optimizations that we seek to conduct. IncA is evaluated also through a Datalog engine. Similar to Souffle and Flix, its current implementation also supports the definition of lattices.

FlowSpec [78] is a recent approach to the declarative specification of *intra*-procedural flow-sensitive data-flow analysis. So far, FlowSpec has been evaluated for conciseness and expressiveness, as these seem to have been its author’s main goals. In SOSA we will consider FlowSpec’s language elements to see whether they can positively impact our own DSL design. As SOSA seeks to support also inter-procedural analysis, its DSL will certainly need to go beyond FlowSpec’s.

No approach mentioned here supports any of the domain-specific optimizations that I have in mind, e.g. automated staging or sparsification just to name a few. Some do support parallelization, on the level of Datalog.

### A.2.6. Compilation of program analyses, partial evaluation and multi-stage programming

The Datalog engine Souffle mentioned above works by translating code written in a variant of Datalog to C++ code, which can then be compiled to machine code with an off-the-shelf C++ compiler. With some additional machinery, this mechanism could already be used to obtain a profile-guided compilation approach [62] in which one uses profiling information obtained from some static-analysis runs to optimize the C++ code that is generated for subsequent runs. Naturally, though, such a source-code based approach would lead to some non-negligible round-tripping time, such that the overall optimization approach might not pay off. Hence in SOSA we will likely have to investigate more direct approaches to compilation. Over the past few years, my group has designed and implemented the Phasar framework [70] for static analysis and transformation of C/C++ programs on top of the popular LLVM framework [47]. LLVM allows one to build relatively efficient compilation approaches, and in essence real-world just-in-time compilers, which optimize the execution not just from one run to the next but even within the same run. During SOSA I will thus research to what extent self-adaptive static analysis can benefit from such tighter integration and faster round-tripping. GraalVM is a versatile framework for implementing efficient program execution environments [92, 93]. It can interface with LLVM. Truffle [92, 93] is a frontend to GraalVM that allows one to support multiple differing input languages. We will investigate this tool chain as a potential technology base for defining the declarative definition language we envision.

SymCC [63] is an approach that already builds on top of LLVM. It has the goal to accelerate the symbolic execution of some target program by compiling this symbolic execution directly into the program’s binary instead of symbolically executing the program in an interpreted way as is usually the case. To this end, SymCC parses the program’s source code and translates it to LLVM bitcode using LLVM’s frontend Clang. Then next SymCC instruments this program with additional code that performs a symbolic execution of certain selected values in addition, i.e., alongside the regular concrete program execution. The work on SymCC is promising in the sense that the authors were able to show that this kind of approach can greatly outperform traditional interpreter-based approaches. Yet, it cannot be directly reused for SOSA because symbolic execution is quite different from static analysis in the way it executes. In particular, static analysis performs a fixed-point iteration using an abstract domain, which means that a static analysis can traverse the program statements in an order quite different from the regular execution order. For these reasons, I do not actually believe that such a tight integration of the analysis with the analyzed program would be suitable for data-flow analysis. In SOSA we thus rather focus on compilation schemes for the efficient allocation of data structures and on parallelization.

### A.2.7. Application-level virtual machines

In terms of its architecture, the solution I propose with SOSA has some similarity to the architecture of application-level virtual machines such as the Java virtual machine (JVM). Their just-in-time compilers (JIT) profile the hosted applications’ execution to optimize that very same execution on the fly. Some virtual machines further include ahead-of-time optimizations and pool optimized code for efficient reuse on later runs [30]. The main difference between a JVM and its JIT to SOSA is that within SOSA we do not seek to execute general-purpose Java code but instead a specific static analysis. I will design *domain-specific* representations of this static analysis at all levels of its execution, so that also domain-specific optimizations can be applied. Thus, while I do plan, in fact, to reuse ideas regarding the architecture and design of application-level virtual machines, the specific intermediate representations, the analyses on them, the transformations between them, and the optimizations within them will certainly differ from (and complement) those within general-purpose JITs.

### A.2.8. Profiling

While there has been substantial prior research on profiling, little research has been conducted on domain-specific profiling. MetaSpy [4] is an infrastructure that enables one to quickly prototype new profilers for their domain-specific languages and models. Savrun-Yeniçeri et al. [68] developed a profiler that helps compare and evaluate programs implementing the same algorithms written in different languages and thus enabled cross-language comparison. This profiler is built on top of the Truffle framework and instruments the AST nodes that Truffle generates from the source code. Since then, a number of further profiling tools have been written using Truffle. While virtually all of them seek to enable the profiling of imperative programming languages, not declarative DSLs, we will nonetheless investigate their utility for SOSA.

## B. Methodology

The work program comprises seven work packages, WP1–WP7. In combination, their execution will allow SOSA to fulfil its main goal: the conception and realization of a framework for self-optimizing static analysis.

WP	Tasks	Year 1				Year 2				Year 3				Year 4				Year 5				
		Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	
	<i>Project Lead</i>																					
WP1	Classifying existing static-analysis optimizations	entire team																				
WP2-4	<i>Self-aware Analysis</i>																					
WP2	Joint intermediate representation																					
WP3	Declarative Analysis Definition Language																					
WP4	Self-Awareness and Self-Optimization																					
WP5-7	<i>Analyzing and evaluating Static Analysers</i>																					
WP5	Automated Analysis Compilation																					
WP6	Domain-specific Profiling																					
WP7	Large-scale Evaluation																					

Figure 3: Work packages, milestones and project schedule

**Schedule and composition of the research team.** Figure 3 provides a rough schedule of the work program. As PI, I will be involved in all work packages. I will guide the research on the novel concepts and methodology we develop, I will oversee and inform the design of individual parts of the solution as well as the overall solution architecture. Also, to safeguard the validity of the scientific results obtained, I will thoroughly discuss and oversee the design and conduct of all experiments.

Besides me as PI, the research team will comprise six Ph.D. students funded out of this project as well as two PostDocs financed by Paderborn University. The Postdocs will combine their own independent research agenda with SOSA’s methodology and act as experts for WP2–WP4 and WP5–WP7, respectively, and WP1 jointly with the PI. WP2 to WP7 will each be headed by one of the six Ph.D. students. Each WP is designed to yield a good Ph.D. topic. All initially available scientific staff will jointly contribute to the initial WP1. All scientific staff will also be involved in the actual implementation of the envisioned solution, which will be aided further through student assistants. In the following I describe all work packages and how I mitigate risks within them.

### WP1: Classifying existing static-analysis optimizations, entire team

As explained above, there exists a large body of work on what I call “one-off optimizations” for static analyses, i.e., optimizations that may have the potential of being applicable to a large variety of static analyses and target programs, yet in the past have been evaluated only in the context of specific static analyses and target programs. These optimizations comprise approaches that use staging, sparsification and parallelization, or other clever optimization tricks. The goal of this preparatory work package is to produce a systematic overview of these optimizations in terms of an ontology. This ontology will then directly inform the design of the envisioned joint intermediate representation and indirectly also the design of the declarative analysis definition language. To obtain the ontology, we will complete a systematic literature review [43] of these optimizations—a process that I am familiar from previous research [25,57]. This is a preparatory work package that the entire team will contribute to. In fact, together with two of my existing Ph.D. students, I have already started this work in preparation for this proposal. Within the project, we will thus merely finalize this initial investigation.

The key contribution of Work Package 1 lies in the systematization of knowledge in existing static-analysis optimizations.

## WP2: Joint Intermediate Representation, addresses Objective (A), 1st Ph.D. student

Next, based on classified static-analysis optimizations and based on a semantics for the target programming languages (we will consider primarily Java and C/C++), within this work package we will design the desired intermediate representation (IR). We will first consider a representation suitable for powerful automated *ahead-of-time* optimizations of the analysis' execution. Akin to how query optimizers for SQL find similarities in cascaded queries to optimize the way in which expensive database accesses are organized, my team and I will evaluate to what extent one can leverage similarities between individual static-analysis rules to optimize their evaluation ahead-of-time, i.e., before the program to be analyzed is actually known. The recent approach by Shi et al. [72] is an inspiration in this space but is too restricted because it considers taint analysis only. Nonetheless, we will consider their work and related research as a starting point. Because, by definition, ahead-of-time optimizations execute prior to the analysis itself, thus are unaware of the target program.

Next we will evaluate to what extent the same IR is already suitable—or else needs to be augmented—to also facilitate effective *just-in-time* optimizations. The key difference here is that just-in-time optimizations are applied when the target program *is* known, and these optimizations therefore can and should exploit specific program properties, and can even make use of profiling information (see WP5), informing the optimizations where they might best pay off. To best support such optimizations we will consider using an IR that comprises both at the same time information about the target program and information about how the analysis applies to that target program. As one specific example, one can envision a control-flow graph that specifically encodes which nodes' flow functions actually can have an effect on which kinds of data-flow values, or which effects—when combined—cancel each other out. Recent work on Sparse IFDS shows encouraging first results [31, 41] and we will consider this work and related research as a starting point.

Within this work package we will design and formally define this IR. Next in WP3 we will construct a translation that creates this IR automatically from an analysis definition written in the envisioned DSL.

The key novelty of Work Package 2 is the design of an intermediate representation that supports (1) both ahead-of-time and just-in-time optimizations of *static analyses*, and that to this end (2) combines both information about the static analysis with information about the target program.

## WP3: Declarative Analysis-rule Definition Language, addresses Objective (B), 2nd Ph.D. student

The envisioned language is meant to be declarative, i.e., will allow developers to specify *what* a static analysis should compute without stating *how* this result should be computed. As explained, a key challenge will be to design the language with exactly the right level of expressiveness, so that the language is versatile enough to support a rich variety of static program analysis, but not too expressive to prohibit *automatic* analysis and optimization (as surveyed in WP1) of such analyses that are defined in that language. Because more expressive static analysis definitions complicate automated optimizations of the defined analyses, both goals are conflicting. Thus there will be a front of pareto-optimal solutions to choose from. To find those solutions, I plan to move towards them from two directions. At the one end of the spectrum there are existing *declarative* general-purpose languages such as Datalog and Flix. These languages lack almost any sort of domain-specific constructs that would allow for automated domain-specific optimizations (the only exception being the fixed-point construct that Flix offers). These languages are fully declarative and fully generic, they offer few language constructs but also virtually no domain-specific optimizations.

The other end of the spectrum is resembled by the existing *imperative* implementations of static-analysis frameworks such as Soot, WALA, Chord or Phasar. Those implementations do comprise a large number of domain-specific constructs, for instance a number of implementations of various conceptual analysis frameworks, specialized data structures and so on. These frameworks are fully imperative, though, which means that, for any given run, their evaluation strategy is deterministically pre-defined by the analysis-framework configuration, leaving no room for self-adaptation and self-optimization. CodeQL and IncA, which I both described earlier, are special in the sense that they are declarative, yet relatively expressive, thus lying in between. We thus consider them as important pieces of related work that we will examine closely. We will also take care that the language allows for modular analysis definitions, as this has been shown to foster automatic scheduling and parallelization of static analyses (see Section A.2.4). As I have done in the past [12], I plan to formally define and document the DSL's syntax as well as its static semantics and runtime semantics (the runtime here meaning the time of static analysis). Further, we will design and implement a frontend that translates the language into the intermediate representation from WP2.

To mitigate the risk of developing a language that practitioners cannot understand, as we have done in the past [21], we will design the language's form, syntax and semantics through a user-centered design process [52]. This process comprises surveys, interviews and interaction studies prior to the design, and in-lab user-experience studies to validate the design. It is then iterated until a satisfactory solution is obtained. For any user studies we will obtain approval from Paderborn University's ethics committee.



Note that there might be conflicting optimization goals for a static analysis (precision, recall, speed, result complexity, etc.). Hence, in some cases one will only be able to obtain pareto-optimal solutions. In these cases, user feedback guide the choice between such solutions, e.g. by prioritizing among these goals (one could use different profiles depending on the use of the analysis, e.g., checking pull requests, feedback in the IDE), or by even adjusting analysis rules. Also these aspects will be covered by the planned user studies.

Within this work package my team and I will leverage my extensive expertise not just in static analysis but also in programming languages and domain-specific languages for expressing static analyses [10, 11, 46]. We will develop a first prototype for the specific sub-domain of static *security* code analyses. This gives us the advantage of being able to evaluate the expressiveness with the help of established benchmarks, such as the OWASP Top 10 and SANS 25 vulnerability categories and the Common Weaknesses Enumeration (CWE) [17]. Many of those known vulnerability categories come with existing test suites that already in the past have allowed us to determine to what extent a static analysis can identify the given vulnerabilities, and with which level of precision. Once we have developed a solid understanding of pareto-optimal solutions within this domain, we will then broaden the spectrum, evolving the DSL to cover other static-analysis problems as well, for instance to detect functional programming mistakes or potential performance bottlenecks.

The key novelties of Work Package 3 are (1) the design of an easy-to-understand, yet machine readable, declarative analysis definition language that (2) can be translated into the intermediate representation from Work Package 2 and thus enables automated optimizations of analyses denoted in this language, and (3) the use of a user-centered design process to guide that language’s design and to assure its usability.

#### WP4: Self-Awareness and Self-Optimization, addresses Objective ©, 3rd Ph.D. student

The previously described activities will inform this core work package, which will realize an engine for the automated application of static-analysis optimizations (Objective ©), as they were surveyed in Work Package 1, and even combinations thereof. The engine will realize both ahead-of-time and just-in-time optimizations, the latter of which will be based on information from a profiler for static-analysis runs (see Work Package 6 below). I plan to develop the profiler and optimization engine in close coordination, as I anticipate that their operation will be highly intertwined.

A natural question is whether automated machine learning (ML) may play a role in finding the best optimization strategies based on profiling data. To find an answer to this question, I plan to collaborate with my colleagues in the local research center [SustAInable Life-cycle of Intelligent Socio-Technical Systems \(SAIL\)](#) ↗, in which I am a PI, and where I already collaborate with numerous AI experts. While ML might have useful applications in this context, a challenge is that many ML methods come with a high computational cost themselves, which might preclude their use in a just-in-time optimization setting. To mitigate this risk, we will initially evaluate ML techniques (particularly supervised ML and evolutionary algorithms) for the use case of profile-based optimization, in which one first profiles a number of complete static-analysis runs and then applies the ML techniques/algorithms to find an optimal analysis setup for the next run. But then in a second step we will seek to assess the performance of these approaches and—where necessary and viable—will derive heuristics that efficiently approximate the search such that it can be used in a just-in-time optimization setting.

To obtain an architecture that best facilitates the notion of self-awareness SOSA requires, I will draw on past research from the field of autonomic and self-aware computing. As explained earlier, MAPE-K and LRA-M are architectures that might be helpful in that regard. Particularly, the optimization engine, along with the analysis execution and the execution-observing profiler might form an LRA-M loop (see Section A.2.1), thus allowing the analysis to really learn a model of itself and its own execution over the target program. To make an optimal choice, my team and I will conduct a literature study to determine suitable candidate architectures before we choose the architecture that we will prototype and test. At the same time, we will implement a representative set of the optimizations that were surveyed in Work Package 1. While in the literature these optimizations were usually assessed as part of a single static analysis, applied to a small number of target programs, SOSA’s automation will allow us to define these optimizations in a generic style, on top of the previously developed IR (Work Package 2). For instance, in previous work I have shown that flow-sensitive tpestate analysis can be accelerated by adding a previous flow-insensitive pointer analysis stage. But this was implemented and assessed using one particular tpestate analysis only. The engine developed in this work package will instead allow us to automate the derivation of such staged analyses directly from a single declarative analysis definition. This enables us to measure the efficacy of such optimizations for a broad range of different static analyses.

To then evaluate optimality of the various design choices within this activity, I plan to establish and publish a benchmark that will allow us to reliably measure prediction accuracy. Using this benchmark, I specifically plan to measure how well the chosen methods can predict the efficiency and precision of various analysis configurations and definitions. Moreover, I plan to assess how well the trained models generalize to various different application scenarios. Work Package 7 will further describe how I plan to evaluate the costs and benefits of SOSA on a very large scale.

The key novelties of Work Package 4 are (1) the combination of methods from autonomic and self-aware computing, data science and static analysis research to obtain a self-optimizing code analysis engine, and (2) the realization of optimizations so that they can *automatically* be applied to a *wide range* of static analyses.

### WP5: Automated Analysis Compilation, addresses Objective [Ⓓ](#), 4th Ph.D. student

As explained earlier, static analyses are typically implemented as an abstract *interpretation* of the target program, which causes computational overhead because the interpreter decomposes and then evaluates the program statements one by one. Thus, as an added twist to the automated analysis optimization from WP4, we will investigate under which circumstances it pays off to instead generate a targeted implementation that uses not just algorithms but also data structures and memory-access patterns optimized towards the specific analysis of the specific target program. To this end, we will design and implement an appropriate compiler backend that will, for instance, generate optimally cascaded maps to allow most efficient access to the analysis internal state.

As explained, related concepts have already established, for instance in the areas of partial evaluation, program specialization and multi-stage programming [13,38,84,85,88,91]. Yet such partial evaluation is known to be very costly. So far few approaches have succeeded in partially evaluating substantial pieces of software, particularly if these have complex inputs [44]. Because target programs are extremely complex inputs, we thus seek to mitigate risk by choosing a simpler approach in which the static analysis is merely instantiated with optimized data structures, and no partial evaluation is conducted. After all, because the compilation itself incurs a cost, it is very much unclear how large the set of use cases is under which this cost is offset by the faster analysis times. We will mitigate the risk of not obtaining a performance gain further by also investigating (1) restricting optimized compilation to *parts* of the analysis, as done in current just-in-time compilers, and (2) using profile-guided compilation, in which one uses profiling information obtained from some static-analysis runs to optimize the code that is generated for subsequent runs.

The compilation-based approach is supposed to retain the same precision and recall that also an interpretation-based variant of the same static analysis would expose. Through empirical comparisons we will validate that our implementation indeed ensures such equivalence. Moreover, we will empirically assess the performance gains (or losses) that incur through compilation and execution of the compiled analysis.

The key novelty of Work Package 5 is the automated, compilation of a static analysis with data structures and memory-access patterns optimized to the analysis of the particular target program.

### WP6: Domain-specific Profiling, addresses Objective [Ⓔ](#), 5th Ph.D. student

Within this work package we will design and implement a domain-specific profiler for that enables the self-optimization engine from Work Package 4 to establish the required situational awareness regarding the analysis' own execution. The goal is to correlate analysis runs, and the performance and precision observed during those runs, with analysis configurations. To identify the potential for powerful just-in-time optimizations, we will follow a white-box approach, i.e., collect not just data about the analysis output but also about its internal workings: In which program parts does the analysis abstraction remain precise and concise? Where does it start to diverge? Which rules are responsible for such divergence? Are these rules effective in removing false warnings in other parts of the program? Which rules counter-act one another and where? To be able to ideally guide the self-adaptive optimization, the profiler will have to understand such domain-specific concepts. We will design the profiler such that its analytics engine can be connected to different interfaces that allow one to profile both interpreted and compiled static analyses. Since profiling itself can involve a significant cost, a challenge that we need to overcome is to find the right level of profiling that both yields sufficient information and itself is sufficiently cheap. Previous research has shown that one can speed up profiling by inlining the profiling code into the profiled application, because then the profiling code itself can become subject to just-in-time optimizations [6]. In case we decide to build our DSL on top of GraalVM, the previously discussed work by Savrun-Yeniçeri et al. [68] will be a great starting point for implementing an appropriate profiler.

The key novelty of Work Package 6 is in the development of profiler that can yield feedback on the execution of static analyses in *domain-specific* terms, and which can profile both interpreted and compiled static analyses.

### WP7: Large-Scale Evaluation , addresses Objective [Ⓕ](#), 6th Ph.D. student

As explained earlier, we will evaluate the costs and benefits of individual techniques and activities within *their* respective work packages. There we will evaluate the relative costs (in terms of required computational resources and remaining human effort) and benefits (mostly in terms of reduction of manual effort) of the approach and tool chain developed within these work packages. Specifically, within the final two years of the project, we will use the then existing solution to evaluate, for instance, to what extent in general automatically optimized static analyses show superior precision, recall, speed, memory consumption, etc. with respect to comparable, pre-existing manually written static analyses. Also we will evaluate to what extent it is possible for the automatic

optimization to automatically navigate through the space of possible pareto-optimal solutions. For instance, we will run large-scale evaluations in which we assess the generated analyses in various application contexts such as IDE integrations and CI/CD build chains.

As an important contribution to the field of static-analysis research, as part of this activity I plan to conduct a large-scale systematic assessment of the extent to which (existing and yet to be invented) optimizations for various kinds of static analyses are applicable also to other analyses and under which circumstances they are applicable at all. As part of Work Package 4, we will already have implemented, as extensions to the automated optimization engine, automatic analysis transformations that mimic a range of optimizations for static analyses that have been published in the scientific literature, as they were surveyed in Work Package 1. We will evaluate how these optimizations, both individually and in various combinations, affect the analysis result in terms of precision, recall, analysis time and memory consumption, and in terms of comprehensibility in terms of the analysis result. For instance, currently the question of which kinds of analyses can be separated into which kinds of stages, remains open. Further, we will seek to classify target programs according to the analysis optimizations that are best applicable when analyzing these programs, e.g.: How effective is which kind of staging when analyzing which target programs? An interesting aspect is here whether and to what extent multiple analysis optimizations may cancel each other out or even support one another when combined.

Important example classes of target programs are mobile applications, web applications, micro services, or serverless applications. For some of these classes there already exist well-established benchmarks that seek to provide a representative sample of applications within the respective class. Some benchmarks particularly popular in the static-analysis community comprise DaCapo [7], Qualitas [86] and the Juliet suite [90]. Yet each of these benchmarks only covers certain domains and aspects, and to some extent they are also slowly getting outdated. Fortunately, with approaches such as Hermes [66] and ABM [23] there now exists comprehensive tool support for curating new benchmarks. We will make use of this tool support to complement existing benchmark suites where useful and necessary for the purposes of this study.

In terms of the static analyses we will assess, as described previously, we will focus on security code analyses first (taint and tpestate analyses as well as constant propagation) and then gradually include also other analyses, for instance, to detect functional programming mistakes or potential performance bottlenecks

The key novelty of Work Package 7 is in the *large-scale, systematic assessment* of the effect that individual optimizations, and combinations of those, have when applied to various static analyses of various classes of target programs. To boost research in the field, the Work Package 7 will report on the static analysis optimizations that show the best effect, and under which circumstances these optimization effects are obtained.

All implementations, experimental setups and results will be faithfully documented and published online, both as continuously maintained open source project and in the form of snapshots at CERN's Zenodo platform, such that full reproducibility can be obtained.

### Scientific and practical impact of the proposal

After its successful completion, SOSA will have brought a radical breakthrough to static-analysis research, and will have set the stage towards ensuring software quality and security on a global scale. To the **static-analysis community**, SOSA will provide a novel open platform to experiment with static analyses and their optimizations, and to automate large-scale studies regarding the efficacy of such optimizations with respect to different classes of target programs. SOSA will yield profound insights into which analysis optimizations apply in which contexts, how they can or cannot support one another. The planned work on self-awareness and self-optimization will push the limits of automation in the static analysis domain. Moreover, SOSA will consolidate, and bring to the next level, the field of declarative definitions for static program analyses. This, and the planned work on the coordinated execution of modular sub-analyses (see Section A.2.4) and on automated compilation of static analyses will boost research in the **programming languages** community.

While SOSA seeks to address real-world challenges, it cannot be funded by private corporations because of the associated risks and also because any exploitation of the project's results will not be able to commence until after several years into the project. Nonetheless, as a long-term consequence, SOSA will enable **software developers** to define, adapt and execute static analyses with ease, without having to worry about how they actually execute. While currently in many cases a successful deployment of static analysis requires the help of analysis experts, SOSA will abolish that need. In result, static analysis will finally enjoy large-scale adoption, which in itself will boost software quality and security on a global scale. SOSA may be the enabler for allowing all those developers to use static analysis technology with ease who will likely soon be demanded to use it once the Cyber Resilience Act is put into law.


## Questions and answers

**How does SOSA address important challenges?** When the cyber resilience act becomes law, it will likely force software engineers to deploy and customize analysis tools. Studies have shown that this is too hard a task for engineers who are novices in static analyses. Automatic self-optimization can help solve this. For the first time, it will make SAST technologies accessible to a wide range of software engineers, also to such who are not experts in static analysis.

**How do we know that SOSA is even feasible?** As described, much previous work has successfully optimized many individual static-analysis designs. Self-optimization and staged optimization has been successfully applied in some contexts such as selecting optimal policies for context sensitivity. Also in other situations we know that the optimal choice of static analysis depends on the combination of analysis and target program. Declarative approaches based on Datalog have shown that one can automatically optimize analysis execution on the logic level. SOSA will provide a declarative language *richer* and more domain-specific than Datalog, which yields further optimization potential. The PI is a leading expert in all relevant technologies: static analysis, DSLs, VMs, compilation.

**What are examples of self-optimization?** In B1 I have given the example of using a flow-insensitive pre-analysis stage for taint analysis. More generally, such staged analyses perform simple and quick pre-analyses to optimize later analysis stages. Sparse static analysis is a special form of this scheme, but many other aspects can be considered during self-optimization: Which analysis direction to choose? Which program entry points? Which context policy? Which analysis algorithm (e.g. IFDS, IDE, monotone framework)? Use on-demand or exhaustive analysis? Which pointer analysis to use? When to store and not to store procedure summaries? What kind of summaries? When is flow-sensitivity important? We envision that SOSA will allow the automated search for optimal solutions w.r.t. all these design choices, along with automated evaluation. Another example we have given in B1 is that of choosing linear vs. full constant propagation in buffer overflow detection. While full constant propagation can be more precise in theory, in practice that difference may not matter for many target programs, and the price it incurs not worthwhile paying.

**Why static analysis and not abstract interpretation or formal verification?** We see abstract interpretation and formal verification as special forms of static analysis. Many techniques of SOSA may transfer to those domains as well. Both abstract interpretation and more general formal verification focuses, however, in static analysis that is provably sound. Our focus here is a different one. We focus instead on analyses that are easier to scale and rather seek to avoid false positives. Many studies have shown that false positives are a primary detractor from static analysis tools. Hence avoiding them seems crucial towards adoption of this technology. When implementing static-analysis optimizations, we will certainly take care, however, to reason about which optimizations maintain soundness and which ones do not. This should tell us very well, which optimizations might also be safe to apply in a verification context.

**To what extent do you plan to use Machine Learning?** Previous work has suggested that one may use Machine Learning (ML) to detect vulnerabilities directly using an ML model trained on known vulnerabilities, even without the need for static data-flow analysis: so-called ML-SAST approaches. Recent works, however, have shown that these approaches are very limited in what they can achieve: Some vulnerability categories are really hard to learn [95], and many approaches that claimed early successes turn out to have rather biased experiments [35], making it unclear how well they really succeed outside of cross-validation experiments. And even if they do succeed, opposed to data-flow analysis, most ML-SAST approaches only signal *that* a given piece of code may be vulnerable but not really *how* it is vulnerable. Moreover, they typically report that a given *function* or *file* is vulnerable, rarely even a line of code. No study has shown so far, that these kinds of vulnerability reports provide information that is actionable to developers. I believe that these approaches can be useful, but probably rather in prioritizing the execution of static analyses: If an ML-SAST model is very certain of a vulnerability being present at a certain location, one could use a just-in-time analysis approach [22] to prioritize the analysis such that it inspects that part of the code first, and reports actual vulnerabilities, confirmed by data-flow analysis, faster than without the use of ML-SAST. In that sense, machine learning can be one of several tools that SOSA may use to guide static analysis towards its optimal execution. Within [SustAInable Life-cycle of Intelligent Socio-Technical Systems \(SAIL\)](#)  we are currently researching useful combinations of static analysis and machine learning.

**Do you really mean to compile the static analysis? Is this really necessary?** We will have to see what pays off. For quickly terminating static analyses, an ahead-of-time compilation into an actual binary may not be worthwhile. We will thus likely choose for a rather lightweight compilation scheme, in which we generate, in particular, data structures that are targeted towards the analysis and target program, and which the analysis can then efficiently use during its execution. Having stated that, recent experiments with implementing program analyses in Rust have shown us unexpectedly strong speedups, which we cannot yet fully explain, but which promise much potential. Likely these speedups originate from relatively fast accesses to only shortly used memory that—opposed to Java—does not need to be garbage collected. We thus plan to explore whether an analysis backend that make use of such tricks may be beneficial.



## References

- [1] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the logicblox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1371–1382, 2015.
- [2] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. d’Amorim, and M. D. Ernst. Static analysis of implicit control flow: Resolving java reflection and android intents (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 669–679, 2015.
- [3] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal—a tool suite for automatic verification of real-time systems. In *International hybrid systems workshop*, pages 232–243. Springer, 1995.
- [4] A. Bergel, O. Nierstrasz, L. Renggli, and J. Ressa. Domain-specific profiling. In J. Bishop and A. Vallecillo, editors, *Objects, Models, Components, Patterns*, pages 68–82, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [5] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.
- [6] W. Binder, J. Hulaas, P. Moret, and A. Villazón. Platform-independent profiling in a virtual execution environment. *Software: Practice and Experience*, 39(1):47–79, 2009.
- [7] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.
- [8] E. Bodden. *Verifying finite-state properties of large-scale programs*. PhD thesis, McGill University, June 2009. Available in print through ProQuest.
- [9] E. Bodden. The secret sauce in efficient and precise static analysis: The beauty of distributive, summary-based static analyses (and how to master them). In *ACM SIGPLAN International Workshop on the State Of the Art in Java Program Analysis (SOAP 2018)*, ISSTA ’18, pages 85–93, New York, NY, USA, 2018. ACM.
- [10] E. Bodden, F. Chen, and G. Roşu. Dependent advice: A general approach to optimizing history-based aspects. In *AOSD ’09: Proceedings of the 8th international conference on Aspect-oriented software development*, pages 3–14, New York, NY, USA, Mar. 2009. ACM.
- [11] E. Bodden and L. Hendren. The clara framework for hybrid typestate analysis. *International Journal on Software Tools for Technology Transfer (STTT)*, 14:307–326, 2012. 10.1007/s10009-010-0183-5.
- [12] E. Bodden, P. Lam, and L. Hendren. Partially evaluating finite-state runtime monitors ahead of time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(2):7:1–7:52, June 2012.
- [13] D. Boucher and M. Feeley. Abstract compilation: A new implementation paradigm for static analysis. In T. Gyimóthy, editor, *Compiler Construction*, pages 192–207, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.
- [14] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. *ACM SIGPLAN Notices*, 44(10):243–262, 2009.
- [15] J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 55–66. ACM, 1991.
- [16] M. Christakis and C. Bird. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, page 332–343, New York, NY, USA, 2016. Association for Computing Machinery.
- [17] S. Christey, J. Kenderdine, J. Mazella, and B. Miles. Common weakness enumeration. *Mitre Corporation*, 2013.
- [18] P. Cousot. Abstract interpretation. *ACM Computing Surveys (CSUR)*, 28(2):324–328, 1996.
- [19] A. Dann, B. Hermann, and E. Bodden. Modguard: Identifying integrity & confidentiality violations in java modules. *IEEE Transactions on Software Engineering*, 2019.
- [20] R. de Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, D. Weyns, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. M. Göschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Miranda, J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Schäfer, R. Schlichting, D. B. Smith, J. P. Sousa, L. Tahvildari, K. Wong, and J. Wuttke. *Software Engineering for Self-Adaptive Systems: A Second Research Roadmap*, pages 1–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
- [21] L. N. Q. Do. *User-Centered Tool Design for Data-Flow Analysis*. PhD thesis, Universität Paderborn, Oct. 2019.
- [22] L. N. Q. Do, K. Ali, B. Livshits, E. Bodden, J. Smith, and E. Murphy-Hill. Just-in-time static analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2017*, pages 307–317, New York, NY, USA, 2017. ACM.



- [23] L. N. Q. Do, M. Eichberg, and E. Bodden. Toward an automated benchmark management system. In *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2016, pages 13–17, 2016.
- [24] V. D’silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.
- [25] J. Geismann and E. Bodden. A systematic literature review of model-driven security engineering for cyber–physical systems. *Journal of Systems and Software*, 169:110697, 2020.
- [26] H. Giese, T. Vogel, A. Diaconescu, S. Götz, and S. Kounev. Architectural concepts for self-aware computing systems. In *Self-Aware Computing Systems*, pages 109–147. Springer, 2017.
- [27] GitHub Inc. Codeql language. <https://securitylab.github.com/tools/codeql>.
- [28] GitHub Inc. Lgtm static analyzer. <https://lgtm.com/>.
- [29] Q. Gong. *Extending Parallel Datalog with Lattice*. PhD thesis, The Pennsylvania State University, 2020.
- [30] P. F. Hagggar, J. A. Mickelson, and D. Wendt. Single-instance class objects across multiple jvm processes in a real-time system, Jan. 11 2005. US Patent 6,842,759.
- [31] D. He, H. Li, L. Wang, H. Meng, H. Zheng, J. Liu, S. Hu, L. Li, and J. Xue. Performance-boosting sparsification of the ifds algorithm with applications to taint analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 267–279, 2019.
- [32] D. Helm, F. Kübler, J. T. Kölzer, P. Haller, M. Eichberg, G. Salvaneschi, and M. Mezini. A programming model for semi-implicit parallelization of static analyses. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 428–439, 2020.
- [33] D. Helm, F. Kübler, M. Reif, M. Eichberg, and M. Mezini. Modular collaborative program analysis in opal. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 184–196, 2020.
- [34] M. C. Hubscher and J. A. McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys (CSUR)*, 40(3):1–28, 2008.
- [35] L. Hüther, B. J. Berger, S. Edelkamp, S. Eken, L. Luhrmann, H. Rothe, M.-S. Schröder, and K. Sohr. Machine learning in the context of static application security testing – ml-sast. German Federal Office for Information Security (BSI), 2022.
- [36] S. Jeong, M. Jeon, S. Cha, and H. Oh. Data-driven context-sensitivity for points-to analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA), Oct. 2017.
- [37] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don’t software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.
- [38] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.
- [39] H. Jordan, B. Scholz, and P. Subotić. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*, pages 422–430. Springer, 2016.
- [40] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.
- [41] K. Karakaya and E. Bodden. Two sparsification strategies for accelerating demand-driven pointer analysis. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2023. To appear.
- [42] U. Khedker, A. Sanyal, and B. Sathe. *Data flow analysis: theory and practice*. CRC Press, 2009.
- [43] B. A. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 07 2007.
- [44] H. Koo, S. Ghavamnia, and M. Polychronakis. Configuration-driven software debloating. In *Proceedings of the 12th European Workshop on Systems Security, EuroSec ’19*, New York, NY, USA, 2019. Association for Computing Machinery.
- [45] S. Kounev, J. O. Kephart, A. Milenkoski, and X. Zhu. *Self-Aware Computing Systems*. Springer, 2017.
- [46] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini. Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.
- [47] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.
- [48] J. Lerch, B. Hermann, E. Bodden, and M. Mezini. Flowtwist: Efficient context-sensitive inside-out taint analysis for large codebases. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 98–108. ACM, 2014.

- [49] J. Lerch, J. Späth, E. Bodden, and M. Mezini. Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths. In *IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*, pages 619–629, Nov. 2015.
- [50] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis. Precision-guided context sensitivity for pointer analysis. *Proc. ACM Program. Lang.*, 2(OOPSLA), Oct. 2018.
- [51] F. Logozzo. Static analysis for security at the facebook scale. In *CurryOn*, 2016.
- [52] T. Lowdermilk. *User-centered design: a developer’s guide to building user-friendly applications*. O’Reilly Media, Inc., 2013.
- [53] J. Lu and J. Xue. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct. 2019.
- [54] M. Madsen and A. Møller. Sparse dataflow analysis with pointers and reachability. In *International Static Analysis Symposium*, pages 201–218. Springer, 2014.
- [55] M. Madsen, M.-H. Yee, and O. Lhoták. From datalog to flix: A declarative language for fixed points on lattices. In *ACM SIGPLAN Notices*, volume 51, pages 194–208. ACM, 2016.
- [56] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(1):1–41, 2005.
- [57] M. Nachtigall, M. Schlichtig, and E. Bodden. A large-scale study of usability criteria addressed by static analysis tools. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 532–543, 2022.
- [58] M. Naik. Chord: A versatile platform for program analysis. In *Tutorial at ACM Conference on Programming Language Design and Implementation*.
- [59] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. *SIGPLAN Not.*, 41(6):308–319, June 2006.
- [60] R. Nasre, K. Rajan, R. Govindarajan, and U. P. Khedker. Scalable context-sensitive points-to analysis using multi-dimensional bloom filters. In Z. Hu, editor, *Programming Languages and Systems*, pages 47–62, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [61] A. Newell. Some problems of basic organization in problem-solving programs. Technical report, Rand corp santa monica ca, 1962.
- [62] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 16–27, 1990.
- [63] S. Poeplau and A. Francillon. Symbolic execution with symcc: Don’t interpret, compile! In *29th USENIX Security Symposium*, pages 181–198, 2020.
- [64] G. Ramalingam. On sparse evaluation representations. *Theoretical Computer Science*, 277(1-2):119–147, 2002.
- [65] M. Rapoport, O. Lhoták, and F. Tip. Precise data flow analysis in the presence of correlated method calls. In S. Blazy and T. Jensen, editors, *Static Analysis*, pages 54–71, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
- [66] M. Reif, M. Eichberg, B. Hermann, and M. Mezini. Hermes: assessment and creation of effective test corpora. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 43–48, 2017.
- [67] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’95, pages 49–61, New York, NY, USA, 1995. ACM.
- [68] G. Savrun-Yeniçeri, M. L. Van de Vanter, P. Larsen, S. Brunthaler, and M. Franz. An efficient and generic event-based profiler framework for dynamic languages. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, PPPJ ’15, page 102–112, New York, NY, USA, 2015. Association for Computing Machinery.
- [69] B. Scholz, H. Jordan, P. Subotić, and T. Westmann. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 196–206, 2016.
- [70] P. D. Schubert, B. Hermann, and E. Bodden. Phasar: An inter-procedural static analysis framework for c/c++. In T. Vojnar and L. Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 393–410, Cham, 2019. Springer International Publishing.
- [71] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis, 1978.
- [72] Q. Shi, R. Wu, G. Fan, and C. Zhang. Conquering the extensional scalability problem for value-flow analysis frameworks. Technical report, 2019.
- [73] Q. Shi, X. Xiao, R. Wu, J. Zhou, G. Fan, and C. Zhang. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. *SIGPLAN Not.*, 53(4):693–706, June 2018.

- [74] Q. Shi and C. Zhang. Pipelining bottom-up data flow analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 835–847, 2020.
- [75] N. Sinha and C. Wang. Staged concurrent program analysis. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10*, page 47–56, New York, NY, USA, 2010. Association for Computing Machinery.
- [76] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '11*, pages 17–30, New York, NY, USA, 2011. ACM.
- [77] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras. Introspective analysis: Context-sensitivity, across the board. *SIGPLAN Not.*, 49(6):485–495, June 2014.
- [78] J. Smits, G. Wachsmuth, and E. Visser. Flowspec: a declarative specification language for intra-procedural flow-sensitive data-flow analysis. *Journal of Computer Languages*, 57:100924, 2020.
- [79] J. Späth, K. Ali, and E. Bodden. Ideal: Efficient and precise alias-aware dataflow analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA), Oct. 2017.
- [80] J. Späth, K. Ali, and E. Bodden. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, 3(POPL):48:1–48:29, Jan. 2019.
- [81] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for java. *SIGPLAN Not.*, 41(6):387–400, June 2006.
- [82] Y. Sui, D. Ye, and J. Xue. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 254–264. ACM, 2012.
- [83] T. Szabó, S. Erdweg, and M. Voelter. Inca: A dsl for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 320–331, 2016.
- [84] W. Taha. *Multi-stage programming: Its theory and applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.
- [85] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '97*, page 203–217, New York, NY, USA, 1997. Association for Computing Machinery.
- [86] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. The qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference*, pages 336–345. IEEE, 2010.
- [87] E. Union. Cyber resilience act. <https://digital-strategy.ec.europa.eu/en/policies/cyber-resilience-act>, 2022.
- [88] D. Van Horn and M. Might. Abstracting abstract machines. In *ICFP*, volume 45, pages 51–62. ACM, 2010.
- [89] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, 25(2):1419–1457, 2020.
- [90] A. Wagner and J. Sametinger. Using the juliet test suite to compare static security scanners. In *2014 11th International Conference on Security and Cryptography (SECRYPT)*, pages 1–9. IEEE, 2014.
- [91] G. Wei, Y. Chen, and T. Rompf. Staged abstract interpreters: Fast and modular whole-program analysis via meta-programming. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct. 2019.
- [92] C. Wimmer and T. Würthinger. Truffle: a self-optimizing runtime system. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 13–14, 2012.
- [93] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 187–204, 2013.
- [94] X. Xiao and C. Zhang. Geometric encoding: forging the high performance context sensitive points-to analysis for java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 188–198. ACM, 2011.
- [95] Y. Zhao, X. Du, P. Krishnan, and C. Cifuentes. Buffer overflow detection for c programs is hard to learn. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ISSTA '18*, page 8–9, New York, NY, USA, 2018. Association for Computing Machinery.