

## Semantics of Software Systems

— S3 —

Principal Investigator (PI): Andreas Zeller

PI's host institution: CISPA Helmholtz Center for Information Security

Project duration: 60 months

### Summary

**WHAT IF we had software bots that tirelessly test, debug, and monitor our software systems?** IT workers are expensive and scarce. So why can't we further automate boring, repetitive activities such as testing and debugging? The problem is that we lack computer-readable *specifications* (so-called *oracles*) for what the system should do or not do. For decades, this *oracle problem* has been a roadblock to automated test generation, trusted software repairs, and accurate monitoring of software.

Building on groundbreaking research to infer input languages of systems, S3 introduces a unified approach to *learning oracles automatically*. It takes a given software system; *infers* and *decodes* its inputs and outputs; and runs *experiments* to extract *models* of how the system behaves, capturing its semantics by predicting output features for given input features.

These models, named *system invariants*, allow to *fully automate* critical software development activities:

**Testing.** System invariants encode *languages* for automatically generating test inputs and provide *oracles* for checking test results: “In the TLS server, the *<payload>* in the *<heartbeat-response>* must be the same as in the *<heartbeat-request>*.”

**Debugging.** System invariants allow narrowing down causes of software behavior (“The X.509 public key certificate is not recognized if *<subject-name>* contains a zero byte”). Generated tests and oracles ensure reliable automated repair.

**Monitoring.** System invariants enable detecting abnormal behavior at runtime (“In log4j, logging a *<user-agent>* containing “*{jndi: <url>}*” opens *<url>*”). Problematic queries can be isolated and investigated until the problem is fixed.

In the future, testing, debugging, and monitoring would thus be taken over by *software bots* who would autonomously explore software behavior, report issues, and suggest actions to their human co-workers, boosting developer productivity and software reliability.

## A Extended Synopsis of the Scientific Proposal

### A.1 Objectives

With modern software development practices, humans can assemble feature-rich software systems from powerful components. But the larger these systems become, the harder it is to test them—and when they fail, developers are faced with millions of code lines that all could contribute to the failure.

The aim of S3 is to **fully automate testing, debugging, and monitoring of complex software systems**. The idea is to have *software bots* that autonomously write and execute software tests, determine correct behavior, monitor systems in operation, diagnose failures and abnormal behavior, and even repair code to reliably fix failures—reporting and explaining their findings to developers and operators, who may customize and guide their activities (Figure B1-1). Such software bots would **substantially reduce the cost of developing software and considerably improve its reliability**, while humans would focus on the *creative* parts of developing software.

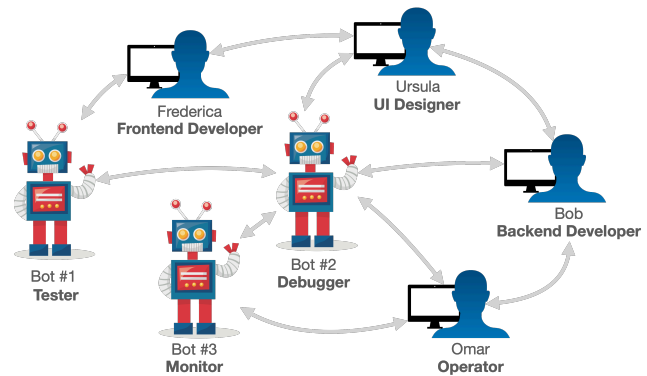


Figure B1-1: Software development and IT operations with software bots

### A.2 Challenges and State of the Art

Today, we *do* have software bots to run tests and monitor software. But these bots are *dumb*—for each bot and each new software, a *human* must specify what to test and what to observe. In order to autonomously create tests and debug and monitor some new software, bots first have to be able to *interact* with it—that is, produce *inputs*, examine *outputs*, relate these to the inputs, and thus explore and check its functionality. Such an autonomy, however, faces long-standing challenges:

First, there is the **input generation problem**: *How can a bot create inputs that reliably cover functionality?* Random system input generators (“fuzzers” [31]) easily detect issues and vulnerabilities in input processing of arbitrary programs. However, creating *valid* inputs and interactions that reliably reach code beyond input processing is still a challenge, for which test generators still require *human assistance*—either through input specifications [9, 8, 14, 19, 2, 34] or a comprehensive population of diverse input samples that cover behavior [29, 44, 26, 6]. Without such assistance, a software bot is lost.

The biggest challenge, however, is the long-standing **test oracle problem**: *How can a bot check the outputs of a system?* All test generators and fuzzers assume some *oracle* that checks for correctness—by default, a *generic* oracle that detects illegal or unresponsive states. If, however, a bot is to run *specific* checks, it needs a *test oracle* that retrieves and evaluates the relevant information from the output. Creating oracles manually [38, 9, 25, 7, 32] is nontrivial and time-consuming: “Compared to many aspects of test automation, the problem of automating the test oracle has received significantly less attention, and remains comparatively less well-solved.” [4].

*Specification mining* and *program synthesis* techniques *mine* oracle candidates from a program [13, 1, 42, 17, 3] that match a set of given executions. Despite the progress made by these techniques, their fundamental problem is that they risk *overfitting* to the given executions. If a  $\text{sqrt}(x)$  function, for instance, is only called with  $x < 100$ , then a specification miner like DAIKON [13] would infer that  $x < 100$  is a precondition of  $\text{sqrt}(x)$  (and  $\text{sqrt}(x) < 10$  a postcondition). Using a generator to test  $\text{sqrt}(x)$  with more arguments would be no help, as it might invoke the function with generated arguments that violate the implicit precondition (say, with  $x = -1$ ), and thus obtain executions with arbitrary results (“garbage in, garbage out”). We thus have a *chicken-and-egg* problem here: To mine a specification from generated inputs, we need the test generator to respect the very preconditions we want to mine. As with test generation, mining oracles thus needs a comprehensive population of diverse input samples that cover behavior.

The oracle problem not only affects the final result of a computation. If the bot is to *interact* with the software and produce new inputs in reaction to output properties, we have the oracle problem at *every step* of the interaction. This is why so much time goes into manual testing and debugging—and still, software catastrophes like *Heartbleed* [10] or *log4shell*, the “largest vulnerability ever” [21], keep on haunting us.

### A.3 Approach: Learning and Leveraging System Invariants

In recent work, my team and I have pioneered groundbreaking techniques to automatically infer the *input language* of a system—that is (1) *input syntax*, learning concise and well-structured *grammars* from how the system decomposes and processes its inputs [20, 16], generating input samples as needed [28]; and (2) *input semantics* as *constraints* detailing how grammar elements relate to each other [41]. From such grammars and constraints, a testing bot can produce myriads of valid inputs, addressing the **input generation problem**. These inputs are all produced at the *system level*, where it is the duty of the program under test to reject any invalid input. Hence, we avoid the “garbage in, garbage out” problem and have only valid executions to learn from, including input syntax and semantics.

We had an eureka moment when we realized that we can create similar techniques to also *infer syntax and semantics of system outputs*, notably *output grammars* from how the system composes outputs; and again *constraints* over grammar elements. A testing bot can thus *decompose* and *check system outputs*, even in relation to earlier (given or generated) inputs—that is, it addresses the **test oracle problem**.

This leads us to the central novel concept of S3. *System invariants* express the semantics of software systems as interleavings of grammars and constraints that characterize inputs, outputs, and their relations. This includes *file and network data* (bytes and characters); *system and API interactions* (calls and results); and even *graphical user interface interactions*. Having once *learned* the invariants of a system, autonomous bots can *leverage* them to test, monitor, and debug the system—without ever getting tired.

#### A.3.1 Learning Input Languages

Let us introduce the concept of system invariants with an example from the networking domain. The *Heartbeat* extension [43] of a TLS server allows clients to check whether a server is still active, by sending a 0x1 byte followed by a *payload*, which the server is expected to include verbatim in its response.

The syntax of such requests is easy to specify. The *grammar* in Figure B1-2 identifies the individual elements in the request; we can use it to *parse* given requests (and thus access and check its constituents), but also to *produce* requests (and thus test the server).

Using a grammar as a producer ensures *syntactic validity*.

This makes test generation far more efficient than generating and mutating random bytes, as valid inputs reliably exercise functionality beyond input processing. (To test input processing and handling of invalid inputs, we can still mutate valid inputs.)

In recent work [16, 20], my team and I have shown how to automatically extract *input grammars* from existing programs. Our MIMID prototype takes a program and a set of diverse sample inputs (which we can even generate from scratch [28]) and tracks where and how individual bytes in the input are processed. Bytes that follow the same path form tokens; subroutines induce hierarchies. After refining grammars through active learning, MIMID thus produces a context-free grammar that accurately represents the input language of a program. As MIMID grammars reuse code identifiers and reflect code structure, no other technique produces grammars that are even nearly as concise, structured, and well-readable.

So far, MIMID and similar techniques have assumed mostly *text-based* parsers with one input source. However, its principles are also applicable to binary inputs and multiple I/O streams. Hence, with S3, the grammar in Figure B1-2 could actually be extracted from a given TLS server and a set of traces, allowing for massive testing of the Heartbeat extension.

#### A.3.2 Learning Interactions

To really *test* a system, we also must check its *output*. In our *Heartbeat* example, the server responds with a 0x2 byte, followed by the payload in the request and some padding. To specify the response, we could again use a grammar. However, we also want to express that the response is the *result* of the request. For this, we *interleave* request and response into a single *I/O grammar* [23] characterizing the interaction (Figure B1-3).

With such an I/O grammar, we can *parse* an entire client/server interaction to *check* whether a 0x1 client request gets a proper 0x2 server response. However, we can also *produce* inputs for the server, expanding  $\langle \text{heartbeat\_request} \rangle$ , and then *parse* and check the server response. Alternatively, we can *mock* a server by

```

 $\langle \text{heartbeat\_request} \rangle ::= 0x1 \langle \text{payload\_length} \rangle \langle \text{payload} \rangle \langle \text{padding} \rangle$ 
 $\langle \text{payload\_length} \rangle ::= \langle \text{uint16} \rangle$ 
 $\langle \text{payload} \rangle ::= \epsilon \mid \langle \text{byte} \rangle \langle \text{payload} \rangle$ 
 $\langle \text{padding} \rangle ::= \epsilon \mid \langle \text{byte} \rangle \langle \text{padding} \rangle$ 

```

Figure B1-2: An input grammar for the TLS Heartbeat Extension

*parsing* its input and then *producing* its output, expanding  $\langle heartbeat\_response \rangle$ . By interleaving multiple input and output sources in a single representation, we obtain a declarative specification of interactions that embeds all the expressiveness of finite-state protocol specifications, yet is detailed enough to produce valid inputs and check concrete outputs alike. Besides bytes and characters exchanged, I/O grammars can also encode file and system I/O, resource accesses, and even GUI interactions [48].

Our techniques for learning input grammars can be equally applied for *learning output grammars*, by tracking output bytes back to the methods that produced them. As the S3 techniques easily produce a wide diversity of inputs, we have a wide variety of interactions to learn from.

```

⟨exchange⟩ ::= ⟨heartbeat_request⟩ ⟨heartbeat_response⟩
⟨heartbeat_request⟩ ::= 0x1 ⟨payload_length⟩ ⟨payload⟩ ⟨padding⟩
⟨heartbeat_response⟩ ::= 0x2 ⟨payload_length⟩ ⟨payload⟩ ⟨padding⟩
⟨payload_length⟩ ::= ⟨uint16⟩
⟨payload⟩ ::= ε | ⟨byte⟩ ⟨payload⟩
⟨padding⟩ ::= ε | ⟨byte⟩ ⟨padding⟩

```

**Figure B1-3:** TLS Heartbeat I/O grammar. After a client sends a  $\langle heartbeat\_request \rangle$ , the *server* responds with  $\langle heartbeat\_response \rangle$ .

### A.3.3 Learning Constraints

Our grammar in Figure B1-3 still is not sufficient for testing, as we miss essential properties—for instance, that the payload in request and response should be identical. Such equalities cannot be expressed in a context-free grammar. However, we can represent these as additional *constraints* that express the relationship between elements in the exchange. Such constraints are similar to function pre-/postconditions, except that the role of function variables is taken by grammar *nonterminals*. They can thus make use of arbitrary formalisms, expressing e.g. arithmetic, strings, sets, or temporal logic.

Figure B1-4 lists such constraints for the grammar in Figure B1-3. We see that  $\langle payload\_length \rangle$  is the length of the  $\langle payload \rangle$  element (for both request and response), and that the payload in request and response must be identical.

```

len(⟨payload⟩) = uint16(⟨payload_length⟩) ≤ 16357
⟨heartbeat_response⟩.⟨payload⟩ = ⟨heartbeat_request⟩.⟨payload⟩

```

**Figure B1-4:** Constraints for the grammar in Figure B1-3: The payload in  $\langle heartbeat\_request \rangle$  and  $\langle heartbeat\_response \rangle$  must be identical.

I call the combination of an I/O *grammar* (Figure B1-3) and I/O *constraints* (Figure B1-4) a **system invariant**, as it specifies the possible interactions of systems as well as their pre- and postconditions. By separating the formalisms for grammars and constraints, we can make use of efficient grammar-based parsers and producers, offloading only the semantic aspects to checkers and solvers. The alternative, universally using string constraints for lexical, syntactical, *and* semantic properties, would quickly overload a solver like Z3 [5, “The solver isn’t particularly good at handling all cases around these”].

Manually specified invariants can easily check concrete interactions. A *Heartbleed* attack, in which  $\langle payload\_length \rangle$  would be manipulated such that  $\langle heartbeat\_response \rangle.\langle payload \rangle$  would include private server memory contents, would be immediately detected by violating the given constraints.

Such manual specification is tiresome, though. Hence, a key idea of this proposal is to *extract constraints automatically* from given traces and implementations. Notably, *program synthesis techniques* could systematically find predicates (constraints) that match the observed values of variables (nonterminals). But while program synthesis may overfit to a small set of observed values, we can use the input grammar to generate as many valid and diverse inputs as we see fit, and hence can have a myriad of interactions to learn from.

Defining such constraints is feasible. Our ISLa *input specification language* [41] allows expressing and checking constraints like the ones in Figure B1-4. ISLa also serves as a *producer*, using the Z3 constraint solver to produce semantically valid inputs. To the best of our knowledge, ISLa is the first and only declarative language that can capture the full semantics of complex inputs for parsing and producing.

We can even *learn* constraints from given inputs. Our ISLearn prototype instantiates patterns from a catalog to obtain invariant candidates [13]; to mine such constraints at the system level is also a first. Our learner can thus easily detect that in all *observed* inputs,  $\langle payload\_length \rangle$  is indeed the length of the payload (Figure B1-4). While validating this invariant candidate, however, the learner would test the server with counterexamples *violating* the payload length assumption, and thus immediately trigger the *Heartbleed* vulnerability (Figure B1-5).

But how would a bot distinguish correct from incorrect behavior? Our learners can *partition* behavior into syntactic and semantic *equivalence classes*. For *Heartbeat*, these classes would be: (1) a request where the payload returned is identical; and (2) one where it is not (i.e., *Heartbleed*). With one invariant and example for each, these two classes would be easy for humans to check; such differentiation would also allow focusing on the most relevant invariants. If *earlier runs or versions* exist, though, our bots can focus on *new and unseen behavior* [12], comparing behavior against invariants learned from *existing* runs and/or

previous program versions. Output differences can be measured at *syntactic* and *semantic* levels, avoiding spurious alarms due to minor differences such as time stamps or session keys.

As a result, bots can produce concise and actionable reports in which grammar elements *abstract* and *generalize* problem circumstances [15, 24]: “If the  $\langle payload\_length \rangle$  field in the client request  $\langle heartbeat\_request \rangle$  exceeds the request length, then in the server  $\langle heartbeat\_response \rangle$ , the  $\langle payload \rangle$  is different (Figure B1-5), in contrast to all past runs. Are you aware of this?”

```
 $\langle heartbeat\_request \rangle ::= 0x1\ 0x3fff\ \text{"hello"}\ 0x0\ \dots$ 
 $\langle heartbeat\_response \rangle ::= 0x2\ 0x3fff\ \text{"hello"}\ 0secret1\ 0secret2\ 0"$ 
```

Figure B1-5: A concrete interchange leaking secrets

#### A.4 Risks and Gains

Let me be clear: What I have sketched above is a “happy path,” which assumes that lots of postulated techniques would work as intended. Real-world system invariants will be far, far more complex than the *Heartbeat* extension—the full TLS 1.3 specification [37], for instance, encompasses 160 pages. Even though we will be able to infer most or even all of a system’s I/O *syntax*, there is no way that S3 would be able to obtain such full and precise semantics of systems fully automatically. That is because all synthesis and learning techniques break down complex behavior into simple functions—and thus may produce approximations even if given an infinite supply of executions to learn from. However, even *approximate* and *partial* invariants already suffice to: (1) obtain high-level, declarative specifications of system behavior, to be read, refined, and extended by humans; (2) have oracles that precisely check for these invariants; (3) generate myriads of valid test inputs; (4) use executions from these test inputs to further refine invariants and oracles; (5) have precise debugging and repair as our generated tests avoid overfitting; (6) can discover anomalies, subtle bugs, and vulnerabilities in critical infrastructures; and (7) thus still reach our objective—software bots that autonomously test, debug, and monitor given software.

#### A.5 Organization

The S3 project comes in six *work packages*: WP1–WP3 cover *obtaining* system invariants, notably foundations (WP1), mining *input* languages (WP2), and synthesizing invariants (WP3). WP4–WP6 *apply* system invariants in testing (WP4), debugging (WP5), and monitoring (WP6).

Six *Ph.D. students*, each working on a WP for 3.5 years, will be funded by the present proposal. Two *postdocs* who will combine their own independent research agenda with S3 techniques (one for WP1–WP3, one for WP4–WP6) and I (devoting 50% of working time to S3) will be funded through my resources at CISPA.

Figure B1-6 shows the information flow between packages. While work packages are designed to *benefit* from each other and create synergies, each package can also be realized independently.

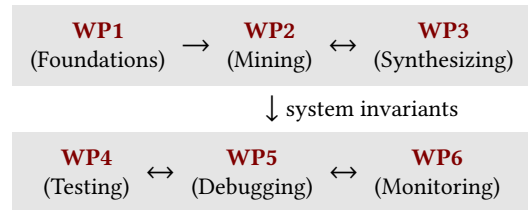


Figure B1-6: S3 work packages

**WP1: Foundations of System Invariants.** In WP1, my team and I define *syntax and semantics* of system invariants, notably for stateful and reactive systems, building on our experience in specifying complex input languages [41]. This includes research to: (1) specify *vocabularies* and *libraries* for common system invariants; (2) track access to *system resources*, notably reading and writing to multiple channels; (3) track *nonfunctional* properties such as resource usage, flows, or dependencies [27], timing, frequency, or thread schedules; and (4) tie multiple layers of encoding and encryption. We will run *case studies*, creating invariants for protocols such as FTP [36], SMTP [35], or TLS [37].

**WP2: Mining Input Languages.** In WP2, we mine *input languages* from existing systems to express their set of valid inputs; these form *system preconditions*. We build on our experience in mining input grammars [16, 20] and track input elements throughout program executions to infer input structure as well as semantic constraints. Challenges include: (1) systematically *refining* input languages by checking whether inputs generated from them are accepted by the system under test; (2) detecting *processing layers* (say, scanning XML tokens, parsing XML syntax, processing an XML tree); and (3) detecting *constraints* such as length fields or checksums.

**WP3: Synthesizing System Invariants.** In WP3, we mine the syntax and semantics of *system outputs*, expressed as grammars and relationships between input and output elements, effectively acting as

system *postconditions*. This includes: (1) learning *output grammars* from given systems; (2) *synthesizing* semantic properties using pattern catalogs and program synthesis techniques; (3) developing *differentiation techniques* to determine the properties most relevant for some outcome; and (4) *refining* properties through generated test cases.

**WP4: Testing with System Invariants.** WP4 makes use of the system invariants specified in WP1 (or mined in WP2 and WP3) to thoroughly *test* existing systems. This builds on our experience in systematically generating highly complex inputs [41, 18, 11]. We will research means to: (1) use system invariants as *oracles*; (2) use invariants as *mock objects* during testing; (3) achieve systematic *coverage* over input and output elements, constraint conditions, and their combinations; and (4) develop a methodology for *invariant-driven development*, refining hand-specified invariants into implementations. As a result, we obtain fully automated testing techniques that effectively and efficiently explore program behavior and flag unexpected outcomes.

**WP5: Debugging with System Invariants.** In WP5, my team and I generate *diagnoses* and *repairs* for failures. Based on our experience in leveraging input grammars to determine failure circumstances [15, 24], my team and I will research how to derive invariants that characterize failing runs (in contrast to passing runs). Such invariants not only *explain* failure conditions, but also *produce* further test inputs that trigger the failure; generated tests and oracles then guide and validate *automated software repair*.

**WP6: Monitoring with System Invariants.** In WP6, we want to raise anomaly detection to a higher level. We first *learn* system invariants from (generated) tests and/or in production. Again, the resulting invariants would express high-level syntactic and semantic I/O properties, but also *resource accesses* dependent on earlier inputs—allowing a far higher expressiveness and abstraction level than the generic checks in common intrusion detection systems. Our bots can then continuously and efficiently *check* system invariants; in case of a violation, they would diagnose the precise circumstances of an attack and isolate matching inputs. Untrusted systems and components could be constrained to extracted and assessed invariants, preventing latent malicious behavior [22].

To implement and evaluate our techniques, my team and I will first build prototypes in and for PYTHON programs. Our experience with our interactive textbooks on test generation [47] and debugging [46] has shown us that in PYTHON, novel techniques for dynamic instrumentation and analysis can be implemented orders of magnitude faster than with traditional compiled languages such as C or JAVA.

Having thus refined our algorithms and techniques, we will *instantiate* them for C programs and integrate them into bots for testing, debugging, and monitoring real-world servers and interactive systems. Our evaluation will focus on traditional *accuracy* measures. For WP4, the question will be how well mined system invariants approximate input and output languages; for WP5, how well they capture failure conditions; and for WP6, how well they identify attacks and abnormal behavior. The goal is to have a high recall (= missing few bugs / failure conditions / attacks) and a high precision (= reporting few false alarms).

The S3 techniques will become available as: (1) *open source textbooks* that demonstrate all concepts with interactive, well-documented code and examples, allowing for easy replication and extension; and (2) *open source software bots* for autonomous testing, debugging, and monitoring of real-world software.

## A.6 Impact

S3 will not only produce autonomous software bots for testing, debugging, and monitoring, but also detect (and fix) significant bugs in day-to-day systems. Catching just one vulnerability before it goes into production might be worth more than the entire cost of this research.

But S3 also unlocks synergies with other research fields, as its system invariants provide a *de facto* infinite supply of tests, with hundreds of input, output, and execution features available for analysis and learning. *Mining function specifications* [13] can finally become precise, providing pre- and postconditions to boost symbolic reasoning and verification; these invariants could even refer to input and configuration properties. Network and security protocols would be supplied with *formal specifications* for inspection and verification [30]. Inferring system invariants from *opaque systems* such as trained machine learning models would *reverse engineer* them [33], explaining and predicting their behavior based on myriads of experiments.

Developers today spend 50% of their time on testing and debugging, and still cannot eradicate all software bugs and vulnerabilities. The gains in productivity and reliability by S3 software bots could be considerable—up to even being measurable in the gross national product of Europe and the world.

## B Curriculum Vitae: Andreas Zeller

Born October 28, 1965, in Hanau, Germany

CISPA Helmholtz Center for Information Security  and Saarland University , Saarbrücken, Germany

ACM Fellow · Six 10-Year Impact Paper Awards · 21,000+ citations · h-index  $\geq$  60

+49 681 87083-2372  · zeller@cispa.de  · <https://andreas-zeller.info>  · ORCID 0000-0003-4719-8803 

Andreas Zeller is one of Europe's most influential researchers in Software Engineering. He has made fundamental contributions to automated debugging and mining software repositories, including:

- Zeller's *Delta Debugging* algorithm [45, 49], which automatically reduces failure-inducing changes and inputs to a minimum, is at the core of most approaches of automated debugging and repair of software.
- *Mining Software Histories to Guide Software Changes* [50] pioneered the field of mining software repositories and is one of the most-cited papers in Software Engineering.
- The *SZZ* algorithm [39], linking bug archives and version histories, is at the heart of most approaches mining past fixes and predicting defects.

All these contributions have their origin in Zeller's 1997 Ph.D. work on software versioning, have been cited thousands of times, and won multiple test of time awards by the community. Zeller's recent research focuses on inferring input formats [20, 16] and leveraging these for testing [19, 40] and debugging [15, 24].

### B.1 Education

1997 Ph.D. in Computer Science, TU Braunschweig (summa cum laude)

1991 Diploma in Computer Science, TU Darmstadt (passed with distinction)

### B.2 Positions

2019–present Faculty, CISPA Helmholtz Center for Information Security, Germany

2003–present Full Professor, Computer Science, Saarland University, Germany

2001–2003 Associate Professor (C3), Computer Science, Saarland University, Germany

1999–2001 Post-Doc Researcher (C1), Computer Science, Passau University, Germany

1997–1999 Post-Doc Researcher (C1), Computer Science, TU Braunschweig, Germany

1991–1997 Scientific Assistant, Computer Science, TU Braunschweig, Germany

Guest researcher/lecturer at Microsoft Research, Redmond, USA (2011, 2009, 2005); ETH Zürich, Switzerland (2007); and University of Washington, USA (2005).


### B.3 Career-Long Contribution Awards


2019 **IFIP Fellow** (IFIP's recognition of substantial and enduring contributions to the ICT industry)


2018 **ACM SIGSOFT Outstanding Research Award** (The highest research award by ACM SIGSOFT, the ACM special interest group on software engineering)


2010 **ACM Fellow** (ACM's most prestigious member grade recognizing the top 1% of members for their outstanding accomplishments) for “contributions to automated debugging and mining software archives,” two fields I helped to shape.

### B.4 Test of Time Awards

2021 **ICST 2021 10-Year Most Influential Paper Award** for “Assessing Oracle Quality with Checked Coverage”  (ICST 2011; with David Schuler)

2020 **ISSTA 2020 10-Year Impact Paper Award** for “Mutation-Driven Generation of Unit Tests and Oracles”  (ISSTA 2010; with Gordon Fraser)

2017 **MSR 10-Year Most Influential Paper Award** for “How Long Will It Take to Fix This Bug?”  (MSR 2007; with Cathrin Weiß, Rahul Premraj, and Thomas Zimmermann)

2015 **MSR 10-Year Most Influential Paper Award** for “When do Changes induce Fixes?”  (MSR 2005; with Jacek Śliwerski and Thomas Zimmermann)

- 2015 **ICSE 10-Year Most Influential Paper Award, Official Runner-Up** (one of two runner-up papers) for “[Locating Causes of Program Failures](#)” (ICSE 2005; with Holger Cleve)
- 2014 **ICSE 10-Year Most Influential Paper Award** for “[Mining Software Histories to Guide Software Changes](#)” (ICSE 2004; with Thomas Zimmermann, Peter Weißgerber, and Stephan Diehl)
- 2009 **ACM SIGSOFT 10-Year Impact Award** for “[Yesterday, my program worked. Today, it does not. Why?](#)” (ESEC 1999; single author).

### B.5 Supervision of Graduate Students and Postdoctoral Fellows

- 2001–present 17 Ph.D.s completed, 2 Ph.D.s to complete in 2022, 5 Ph.D.s currently supervised  
10 Post-Docs completed, 2 Post-Docs currently associated, all CISPA and Saarland U

### B.6 Teaching Activities

- 2021 Interactive textbook “[The Debugging Book](#)”
- 2019 Interactive textbook “[The Fuzzing Book](#)” (with Gopinath, Böhme, Fraser, Holler)
- 2018–present Course of studies “Entrepreneurial Cybersecurity” (Saarland Teaching Award 2020)
- 2012 Udacity Online course CM259 “Software Debugging” (commissioned by Udacity)
- 2005 Textbook “Why Programs Fail”, Morgan Kaufmann Editors
- 1997–present Courses on Software Engineering, Introduction to Programming, Program Analysis, Software Testing, Security Testing, Automated Debugging.

### B.7 Organization of Scientific Meetings

- 2022 **Program Chair** ACM/IEEE International Conference on Software Engineering (ICSE 2022; co-chaired with Daniela Damian)
- 2020 **Program Chair** IEEE International Conference on Software Testing and Verification (ICST 2020; co-chaired with Corina Păsăreanu)
- 2016 **General Chair** IEEE European Symposium on Security and Privacy (EuroS&P 2016)
- 2016 **General Chair** ACM International Symposium on Software Testing and Analysis (ISSTA 2016)
- 2013 **Program Chair** ACM/IEEE International Conference on Automated Software Engineering (ASE 2013; co-chaired with Tevfik Bultan)

Co-Organizer of three Dagstuhl Seminars in the past ten years (2015, 2017, 2023).

### B.8 Institutional Responsibilities

- 2016–2018 Elected member, Saarland University Senate
- 2008–2018 Scientific Vice-Coordinator, Saarbrücken Graduate School for Computer Science
- 2001–2018 Head of the Software Engineering Chair at Saarland University

### B.9 Commissions of Trust

- 2018–present Steering Committee, International Conference on Software Engineering (ICSE)
- 2016–present Research Highlights Editorial Board, Communications of the ACM
- 2016–present Editorial Board, Journal of Software Testing, Verification and Reliability
- 2008–present Steering Committee, Int. Symposium on Software Testing and Analysis (ISSTA)
- 2011–2020 National DFG Proposal Review Panel for Computer Science, Elected Member
- 2013–2017 Editorial Board, IEEE Transactions on Software Engineering (TSE)
- 2009–2017 Editorial Board, Springer Journal on Empirical Software Engineering (ESEM)
- 2011–2016 Steering Committee, European Software Engineering Conference (ESEC)



## Appendix: All ongoing grants and submitted grant applications of the PI (Funding ID)

### Ongoing Grants

Project Title	Funding Source	Amount	Period	Role of the PI	Relation to S3 Proposal
EMPEROR: Learning Causes of Program Behavior	DFG grant	220,000 €	2021–2024	Co-PI <sup>1</sup>	no overlap; explores using ML classifiers for ALHAZEN [24]
CPSec: Effective Testing of Cyber-Physical Systems	BMBF grant	3,200,000 €	2022–2024	Co-PI <sup>2</sup>	no overlap; explores differential fuzzing

### Grant Applications











(none)

<sup>1</sup>Project with Co-PI Lars Grunske, Berlin, totaling one Ph.D. student for three years under supervision of the PI


<sup>2</sup>Project with Co-PI Thorsten Holz (CISPA), Robert Bosch GmbH, and let's dev GmbH & Co. KG, NXP Semiconductors Germany GmbH, totaling two Ph.D. students for three years under supervision of the PI

## C Ten-Year Track Record: Andreas Zeller

### C.1 Ten Representative Publications

1. G. Fraser and **A. Zeller**. **Mutation-driven generation of unit tests and oracles** . *IEEE Transactions on Software Engineering (TSE)*, 2011, 38(2), pp. 278–292 and ISSTA 2010. <https://doi.org/10.1109/TSE.2011.93>  
▷ This paper introduced test oracle inference at the unit level, using mutation analysis
2. V. Dallmeier, N. Knopp, C. Mallon, G. Fraser, S. Hack, and **A. Zeller**. **Automatically Generating Test Cases for Specification Mining** . *IEEE Transactions on Software Engineering (TSE)*, 2011, 38(2), pp. 243–257. <http://dx.doi.org/10.1109/TSE.2011.105>  
▷ This paper pioneered the use of generating test cases for mining (state machine) specifications.
3. C. Holler, K. Herzig, and **A. Zeller**. **Fuzzing with Code Fragments** . In *USENIX Security Symposium*, 2012, pp. 38–48. <https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler>  
▷ The LANGFUZZ tool introduced grammar-based test generation to the security community.
4. A. Gorla, I. Tavecchia, F. Gross, and **A. Zeller**. **Checking App Behavior against App Descriptions** . In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2014, pp. 1025–1035. <http://dx.doi.org/10.1145/2568225.2568276>  
▷ CHABADA introduced checking program behavior (notably the API usage of apps) against natural language descriptions (app descriptions).
5. K. Jamrozik, P. von Styp-Rekowsky, and **A. Zeller**. **Mining Sandboxes** . In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2016, pp. 37–48. <http://dx.doi.org/10.1145/2884781.2884782>  
▷ This paper introduced the idea of mining behavior constraints to later enforce them.
6. B. Mathis, R. Gopinath, M. Mera, A. Kampmann, M. Hörschele, and **A. Zeller**. **Parser-Directed Fuzzing** . In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* 2019, pp. 548–560. <https://doi.org/10.1145/3314221.3314651>  
▷ This paper showed how to leverage dynamic analysis to systematically explore all branches of a parser and hence obtain a set of samples for grammar mining.
7. R. Gopinath, M. Mathis, and **A. Zeller**. **Mining Input Grammars from Dynamic Control Flow** . In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020. <https://doi.org/10.1145/3368089.3409679>  
▷ The MIMD tool is our latest input grammar miner, automatically extracting a concise, well-structured, and well-readable grammar from a program and a set of inputs.
8. R. Gopinath, A. Kampmann, N. Havrikov, E. Soremekun, and **A. Zeller**. **Abstracting Failure-Inducing Inputs** . In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2020. <https://doi.org/10.1145/3395363.3397349>  
▷ This paper shows how to leverage grammars to abstract failure-inducing inputs into general patterns that explain failures and produce further failing test inputs.
9. A. Kampmann, N. Havrikov, E. Soremekun, and **A. Zeller**. **When does my Program do this? Learning Circumstances of Software Behavior** . In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020. <https://doi.org/10.1145/3368089.3409687>  
▷ The ALHAZEN tool systematically explores program behavior, refining and validating hypothesis to determine failure circumstances.
10. E. Soremekun, E. Pavese, N. Havrikov, L. Grunske, and **A. Zeller**. **“Inputs from Hell: Learning Input Distributions for Grammar-Based Test Generation.”**  In *IEEE Transactions on Software Engineering (TSE)*, 2020. <https://doi.org/10.1109/TSE.2020.3013716>  
▷ This paper showed how to learn distributions of input elements to direct test generation towards common and uncommon inputs.

## C.2 Patents

- **A. Zeller** and K. Jamrozik. 2016. “Mining Sandboxes”, European Patent [WO 2016/131830 A1](#) .
- N. Nagappan, T. Zimmermann, B. Murphy, and **A. Zeller**. 2016. “Predicting Defects in Code”, US Patent 9378015.


## C.3 Keynotes at International Conferences, Summer Schools

**FUZZING** (2022) · “Cyber in Nancy” Summer School, Nancy, France (2022) · **ASE** (2021) · **ISSTA** (2020) · **MOBILEsoft** (2020) · **ICSE** (2018) · **RV** (2017) · **MSR** (2017) · **ICST** (2017) · Halmstad Summer School on Software Testing, Halmstad, Sweden (2017) · **ICPC** (2015) · **ICSE** (2014) · **ICMT** (2013) · **SPIN** (2012)

## C.4 Organization of International Conferences


- **ACM SIGSOFT/IEEE TCSE International Conference on Software Engineering (ICSE)** — *Program Co-Chair* (2022) · Program Board Member (2018, 2016) · Doctoral Symposium Chair (2012) · PC Member (2023, 2021, 2020, 2017, 2015, 2014, 2013) · Steering Committee Member (2018–)
- **ACM SIGSAC Computer and Communications Security (CCS)** — PC Member (2021, 2022)
- **ACM SIGPLAN Programming Language Design and Implementation (PLDI)** — PC Member (2017, 2021)
- **IEEE International Conference on Software Testing and Verification (ICST)** — *Program Co-Chair* (2020) · Steering Committee Member (2020–)
- **European Symposium on Security and Privacy (EuroS&P)** — *General Chair* (2016)
- **ACM International Conference on Software Testing and Analysis (ISSTA)** — *General Chair* (2016) · *Steering Committee Chair* (2016–2017) · PC Member (2022, 2020, 2018, 2013)
- **IEEE/ACM Conference on Automated Software Engineering (ASE)** — *Program Co-Chair* (2013) · PC Member (2017, 2014)
- **ACM SIGSOFT Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)** — Doctoral Symposium Chair (2017) · PC Member (2019, 2017, 2015, 2013) · Steering Committee Member (2009–2016)
- **Dagstuhl Seminars** — *Co-Organizer* (“Software Bug Detection: Challenges and Synergies”, 2023 · “Testing and Verification of Compilers”, 2017 · “Artifact Evaluation for Publications”, 2015)

## C.5 Major Contributions to the Early Careers of Excellent Researchers

**Thomas Zimmermann**  (Senior Principal Researcher at Microsoft Research; Associate Professor at University of Calgary) did his Master thesis (2001) and Ph.D. thesis (2006) with me, shaping and pioneering the new research field of Mining Software Repositories. As ACM Fellow and elected president of ACM SIGSOFT, Thomas is now one of today’s most influential Software Engineering researchers.

**Gordon Fraser**  · **Marcel Böhme**  · **Alessandra Gorla**  · **Juan Pablo Galeotti**  · **Rahul Gopinath**  are former Post-Docs now occupying excellent positions—**Gordon Fraser** is full professor at Passau University; **Marcel Böhme** is junior faculty at the Max Planck Institute for Security and Privacy in Bochum; **Alessandra Gorla** is assistant researcher professor at the IMDEA Software Institute in Madrid; **Juan Pablo Galeotti** is professor at the University of Buenos Aires; and **Rahul Gopinath** is assistant professor in Sydney.

## C.6 Examples of Leadership in Industrial Innovation or Design

**Testfabrik AG** , founded in 2011, generates and runs tests for Web applications and mobile apps. After attracting more than 1 million € in startup funding and awards as co-founder, Testfabrik currently has 20+ full-time employees; I serve as chair of the supervisory board.

## References

- [1] G. Ammons, R. Bodík, and J. R. Larus. [Mining specifications](#). In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 4–16, Portland, Oregon, 2002. ACM. <http://doi.acm.org/10.1145/503272.503275> (Page B1-1)
- [2] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert. [NAUTILUS: Fishing for deep bugs with grammars](#). In *Network and Distributed System Security Symposium (NDSS)*, 2019. <https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/> (Page B1-1)
- [3] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. [DeepCoder: Learning to write programs](#). 2017. <https://arxiv.org/abs/1611.01989> (Page B1-1)
- [4] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. [The oracle problem in software testing: A survey](#). *IEEE Transactions on Software Engineering (TSE)*, 41(5):507–525, 2014. <https://doi.org/10.1109/TSE.2014.2372785> (Page B1-1)
- [5] N. Bjørner. [Z3 fails to terminate despite set timeout](#). <https://github.com/Z3Prover/z3/issues/5891#issuecomment-1063110825>, March 2022. (Page B1-3)
- [6] M. Böhme, V.-T. Pham, and A. Roychoudhury. [Coverage-based greybox fuzzing as Markov chain](#). In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16*, pages 1032–1043, New York, NY, USA, 2016. Association for Computing Machinery. <https://doi.org/10.1145/2976749.2978428> (Page B1-1)
- [7] G. Booch. *The Unified Modeling Language user guide*. Addison Wesley, 2005. (Page B1-1)
- [8] L. C. Briand and Y. Labiche. [A UML-based approach to system testing](#). In *International Conference on the Unified Modeling Language, Modeling Languages, Concepts, and Tools (UML)*, pages 194–208. Springer, 2001. <https://dl.acm.org/doi/10.5555/647245.719446> (Page B1-1)
- [9] K. Claessen and J. Hughes. [Quickcheck: A lightweight tool for random testing of haskell programs](#). In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*, ICFP '00, pages 268–279, New York, NY, USA, 2000. Association for Computing Machinery. <https://doi.org/10.1145/351240.351266> (Page B1-1)
- [10] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman. [The matter of Heartbleed](#). In *Proceedings of the 2014 Conference on Internet Measurement Conference, IMC '14*, pages 475–488, New York, NY, USA, 2014. Association for Computing Machinery. <https://doi.org/10.1145/2663716.2663755> (Page B1-1)
- [11] R. Dutra, R. Gopinath, and A. Zeller. [FormatFuzzer: Effective fuzzing of binary file formats](#). CoRR, abs/2109.11277, 2021. <https://arxiv.org/abs/2109.11277> Submitted for publication at ACM TOSEM. Project home page at <https://uds-se.github.io/FormatFuzzer/>. (Page B1-5)
- [12] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. [Bugs as deviant behavior: A general approach to inferring errors in systems code](#). *SIGOPS Oper. Syst. Rev.*, 35(5):57–72, oct 2001. <https://doi.org/10.1145/502059.502041> (Page B1-3)
- [13] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. [Dynamically discovering likely program invariants to support program evolution](#). *IEEE Transactions on Software Engineering (TSE)*, 27(2):99–123, February 2001. <https://doi.org/10.1109/32.908957> (Pages B1-1, B1-3, and B1-5)
- [14] P. Godefroid, A. Kiezun, and M. Y. Levin. [Grammar-based whitebox fuzzing](#). In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 206–215. ACM, 2008. <https://doi.org/10.1145/1375581.1375607> (Page B1-1)
- [15] R. Gopinath, A. Kampmann, N. Havrikov, E. O. Soremekun, and A. Zeller. [Abstracting failure-inducing inputs](#). In *ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 237–248. ACM, 2020. <https://doi.org/10.1145/3395363.3397349> (Pages B1-4, B1-5, and B1-6)
- [16] R. Gopinath, B. Mathis, and A. Zeller. [Mining input grammars from dynamic control flow](#). In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, November 2020. <https://publications.cispa.saarland/3101/> (Pages B1-2, B1-4, and B1-6)
- [17] S. Gulwani, O. Polozov, and R. Singh. [Program synthesis](#). *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017. <http://dx.doi.org/10.1561/2500000010> (Page B1-1)
- [18] N. Havrikov and A. Zeller. [Systematically covering input structure](#). In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 189–199. IEEE, 2019. <https://doi.org/10.1109/ASE.2019.00027> (Page B1-5)

- [19] C. Holler, K. Herzig, and **A. Zeller**. **Fuzzing with code fragments**. In *USENIX Security Symposium*, pages 38–38, Bellevue, WA, 2012. USENIX Association. <https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler> (Pages B1-1 and B1-6)
- [20] M. Hörschele and **A. Zeller**. **Mining input grammars from dynamic taints**. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 720–725, Singapore, Singapore, 2016. ACM. <http://doi.acm.org/10.1145/2970276.2970321> (Pages B1-2, B1-4, and B1-6)
- [21] T. Hunter and G. de Vynck. **The “most serious security breach ever” is unfolding right now**. *Wall Street Journal*, December 2021. <https://www.washingtonpost.com/technology/2021/12/20/log4j-hack-vulnerability-java/> (Page B1-1)
- [22] K. Jamrozik, P. von Styp-Rekowsky, and **A. Zeller**. **Mining sandboxes**. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 37–48, Austin, Texas, 2016. ACM. <http://doi.acm.org/10.1145/2884781.2884782> (Page B1-5)
- [23] B. Jones, M. Harman, and S. Danicic. **Automated construction of input and output grammars**. Technical report, University of North London, August 1999. [https://www.researchgate.net/publication/2448384\\_Automated\\_Construction\\_of\\_Input\\_and\\_Output\\_Grammars](https://www.researchgate.net/publication/2448384_Automated_Construction_of_Input_and_Output_Grammars) (Page B1-2)
- [24] A. Kampmann, N. Havrikov, E. Soremekun, and **A. Zeller**. **When does my program do this? Learning circumstances of software behavior**. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020. <https://publications.cispa.saarland/3107/> (Pages B1-4, B1-5, B1-6, and B1-8)
- [25] S. Kent. **Model driven engineering**. In *International conference on integrated formal methods*, pages 286–298. Springer, 2002. <https://dl.acm.org/doi/10.5555/647983.743552> (Page B1-1)
- [26] Libfuzzer. <https://llvm.org/docs/LibFuzzer.html>. Retrieved 2022-02-01. (Page B1-1)
- [27] B. Mathis, V. Avdiienko, E. O. Soremekun, M. Böhme, and **A. Zeller**. **Detecting information flow by mutating input data**. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 263–273. IEEE, 2017. <https://doi.org/10.1109/ASE.2017.8115639> (Page B1-4)
- [28] B. Mathis, R. Gopinath, M. Mera, A. Kampmann, M. Hörschele, and **A. Zeller**. **Parser-directed fuzzing**. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 548–560. ACM, 2019. <https://doi.org/10.1145/3314221.3314651> (Page B1-2)
- [29] P. McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163. IEEE, 2011. (Page B1-1)
- [30] S. Meier, B. Schmidt, C. Cremers, and D. Basin. **The TAMARIN prover for the symbolic analysis of security protocols**. In N. Sharygina and H. Veith, editors, *Computer Aided Verification*, pages 696–701, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. [https://doi.org/10.1007/978-3-642-39799-8\\_48](https://doi.org/10.1007/978-3-642-39799-8_48) (Page B1-5)
- [31] B. P. Miller, L. Fredriksen, and B. So. **An empirical study of the reliability of UNIX utilities**. *Communications of the ACM*, 33(12):32–44, December 1990. <http://doi.acm.org/10.1145/96267.96279> (Page B1-1)
- [32] M. Mussa, S. Ouchani, W. A. Sammane, and A. Hamou-Lhadj. **A survey of model-driven testing techniques**. In *2009 Ninth International Conference on Quality Software*, pages 167–172, 2009. <https://doi.org/10.1109/QSIC.2009.30> (Page B1-1)
- [33] S. J. Oh, B. Schiele, and M. Fritz. **Towards Reverse-Engineering Black-Box Neural Networks**. Springer International Publishing, Cham, 2019. [https://doi.org/10.1007/978-3-030-28954-6\\_7](https://doi.org/10.1007/978-3-030-28954-6_7) (Page B1-5)
- [34] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury. **Smart greybox fuzzing**. *IEEE Transactions on Software Engineering (TSE)*, 47(9):1980–1997, 2019. <https://doi.org/10.1109/TSE.2019.2941681> (Page B1-1)
- [35] J. B. Postel. **Simple mail transfer protocol (SMTP)**. RFC 821, Internet Engineering Task Force (IETF), August 1982. <https://datatracker.ietf.org/doc/rfc821/> (Page B1-4)
- [36] J. B. Postel and J. Reynolds. **File transfer protocol (FTP)**. RFC 959, Internet Engineering Task Force (IETF), October 1985. <https://datatracker.ietf.org/doc/rfc8446/> (Page B1-4)
- [37] E. Rescorla. **The transport layer security (TLS) protocol version 1.3**. RFC 8446, Internet Engineering Task Force (IETF), August 2018. <https://datatracker.ietf.org/doc/rfc8446/> (Page B1-4)
- [38] D. Rosenblum. **A practical approach to programming with assertions**. *IEEE Transactions on Software Engineering*, 21(1):19–31, 1995. <https://doi.org/10.1109/32.341844> (Page B1-1)

- [39] J. Śliwerski, T. Zimmermann, and **A. Zeller**. **When do changes induce fixes?** In *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR '05*, pages 1–5, New York, NY, USA, 2005. Association for Computing Machinery. <https://doi.org/10.1145/1083142.1083147> (Page B1-6)
- [40] E. Soremekun, E. Pavese, N. Havrikov, L. Grunske, and **A. Zeller**. **Inputs from hell: Learning input distributions for grammar-based test generation**. *IEEE Transactions on Software Engineering (TSE)*, 48(4):1138–1153, 2022. <https://doi.org/10.1109/TSE.2020.3013716> (Page B1-6)
- [41] D. Steinhöfel and **A. Zeller**. **Input invariants**. Technical report, CISPA Helmholtz Center for Information Security, March 2022. <https://publications.cispa.saarland/3596/> Submitted for publication at ESEC/FSE 2022. (Pages B1-2, B1-3, B1-4, and B1-5)
- [42] W. M. P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 1st edition, 2011. (Page B1-1)
- [43] M. Williams, M. Tüxen, and R. Seggelmann. **Transport layer security (TLS) and datagram transport layer security (DTLS) heartbeat extension**. RFC 6520, Internet Engineering Task Force (IETF), February 2012. <https://datatracker.ietf.org/doc/rfc6520/> (Page B1-2)
- [44] M. Załewski. American fuzzy lop. <https://lcamtuf.coredump.cx/afl/>. Retrieved 2022-02-01. (Page B1-1)
- [45] **A. Zeller**. Yesterday, my program worked. Today, it does not. Why? In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, ESEC/FSE-7, pages 253–267, Berlin, Heidelberg, 1999. Springer-Verlag. (Page B1-6)
- [46] **A. Zeller**. *The Debugging Book*. CISPA Helmholtz Center for Information Security, 2021. <https://www.debuggingbook.org/> (Page B1-5)
- [47] **A. Zeller**, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2019. <https://www.fuzzingbook.org/> (Pages B1-5 and B1-13)
- [48] **A. Zeller**, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. **Testing graphical user interfaces**. [47]. <https://www.fuzzingbook.org/html/GUIFuzzer.html> (Page B1-3)
- [49] **A. Zeller** and R. Hildebrandt. **Simplifying and isolating failure-inducing input**. *IEEE Transactions on Software Engineering (TSE)*, 28(2):183–200, February 2002. <https://doi.org/10.1109/32.988498> (Page B1-6)
- [50] T. Zimmermann, P. Weißgerber, S. Diehl, and **A. Zeller**. **Mining version histories to guide software changes**. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 563–572, USA, 2004. IEEE. <https://doi.org/10.1109/TSE.2005.72> (Page B1-6)

## F Questions and Answers

- How does s3 address important challenges?** The central problem of test generation, debugging, monitoring is the lack of *oracles*—predicates that check whether some functionality is correct or not. S3 *solves the oracle problem* by: (1) generating inputs at the system level, where invalid and nonsensical inputs would be rejected; hence, we only learn from valid inputs; (2) decomposing system outputs to learn postconditions and invariants; and (3) systematically exploring input and output space to prevent overfitting to a small set of given inputs. For details, see Section A.2 and Section D.1.3.
- How does s3 go beyond the state of the art?** Automatic test generation and fuzzing so far only check for generic issues such as crashes. With oracles extracted from system output, generated tests can finally check whether behavior is common and correct. With such tests, automated debugging, repair, and monitoring become way more accurate in their warnings, diagnoses, and fixes, because an infinite number of tests is available for guidance and validation. See also Section A.2.
- Why not generate tests at the unit level?** Generating function calls is easy, as arguments and types are all explicit. But types do not always capture the full semantics of arguments. For `int sqrt(int x)`, a test generator may invoke `sqrt(-1)`, and if this fails, produce a (false) failure report. At the system level, this problem does not arise, as programs are expected to reject invalid inputs (Section D.1.1).
- Aren't there already dynamic invariant learners at the unit level?** Yes, but these can only learn from *given* runs, as generated function arguments again may be nonsensical (“garbage in, garbage out”; Section D.1.3). Checks at the unit level come too late to block inputs and prevent damage; and if they fail, they refer to internal conditions that are difficult to characterize and debug.
- How do you know the behaviors you learn from are correct?** We assume that *most of them* are correct, allowing our bots to flag *abnormal* behavior. The abstraction in system invariants allows us to differentiate high-level *equivalence classes* of output (and thus behavior) differences, such that humans can easily assess outliers for correctness (Figure B2-5; see also WP3 and WP4).
- How do you know your test cover all behaviors?** Testing is incomplete by construction. If we know the syntactic and semantic properties of the input, though, we can make testing way more effective, especially as we infer which properties trigger which functionality (WP5).
- My server is a complete black box. How could you infer its input structure?** We can learn the input grammar from an open source alternative, or an output grammar from a client.
- How do you handle encrypted and encoded data?** We assume multiple layers connected via constraints that describe encryption and encoding (WP1–WP2).
- How do you handle passwords? Do you infer them?** This is why system invariants must be readable and maintainable by developers, so they can specify the information that bots need.
- Why not infer *(formalism)* instead of grammars and constraints?** Developers are well familiar with both grammars and conditions. *Regular expressions* and *state machines* cannot fully capture the complexity of file formats and interactions. *Universal grammars* would require encoding basic primitives such as arithmetic as grammars, which is impractical. *Programs* can either *parse* or *produce*, but not both.
- How would s3 capture the entire behavior of *(system)* in a few constraints?** We do not aim for learning (or even specifying) *all* behavior. A small set of *critical* invariants, even partial or approximate, already can bring huge benefits for testing, debugging, and monitoring (Section A.4).
- How does s3 scale to large distributed parallel systems?** One service at a time. In a network of components, we can tie output grammars of clients to the input grammars of servers and vice versa, offering even more opportunities for testing and inference.
- How about vulnerabilities that do not show up in the output?** Such vulnerabilities would manifest themselves via *resource accesses*. If a system has, say, a backdoor that opens an unrestricted shell, a monitor bot would note that it starts a shell not running any of the previously observed scripts.
- How about complex user interfaces such as the metaverse?** We would not apply s3 at the pixel level, but some layer below, where interaction takes place as operations on individual objects.
- Isn't s3 just a continuation of your earlier work?** Not at all. My team and I build on our expertise in inferring input structure, as well as our contributions in testing and debugging. But creating universal ways to decode and assess outputs, solving the oracle problem, and generally going for the largest of scale and automation is far beyond what we—or anyone else—has ever done before.