Lightweight Verification via Specialized Typecheckers

Martin Kellogg
University of Washington
Seattle, USA
kelloggm@cs.washington.edu

ABSTRACT

Testing and other unsound analyses are developer-friendly but cannot give guarantees that programs are free of bugs. Verification and other extant sound approaches can give guarantees but often require too much effort for everyday developers. In this work, we describe our efforts to make verification more accessible for developers by using specialized pluggable typecheckers—a relatively accessible verification technology—to solve complex problems that previously required more complex and harder-to-use verification approaches.

CCS CONCEPTS

• **Software and its engineering → Software verification**.

KEYWORDS

Pluggable types, accumulation analysis, lightweight verification

ACM Reference Format:
Martin Kellogg. 2021. Lightweight Verification via Specialized Typecheckers. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3468264.3473105

1 INTRODUCTION

Software is everywhere, but *correct* software is not. Heavily-used projects include known bugs [7]. 90% or more of Java applications that use cryptography misuse it [16]. Despite decades of research, buffer overflows remain a common causes of security issues [45].

Researchers and practitioners have developed two broad categories of techniques to help developers write correct software: high-precision techniques such as testing that detect some bugs with few false alarms, but miss many others; and, high-recall techniques such as full formal verification by proof that find all bugs, but have many false-alarms or require herculean effort to deploy.

An ideal bug-prevention technique would be:

- *sound*: it would never certify a program that contains bugs.
- *precise*: it would always certify a program with no bugs[1].
- *usable*: it would be easy for developers to use. Ideally, it would be fully automated.
- *comprehensible*: it would be easy for developers to understand why it rejects a program, so fixing the problem is easy.
- *efficient*: it would require no time or space overhead at run time, and would be fast to run at compile time.
- *applicable*: it would run on existing code as-is.

Neither testing nor verification meet these criteria. Testing (and unsound static analyses), while easy for developers to deploy, cannot prove that any kind of bug—even those a given approach might purport to catch—is absent: as Dijkstra quipped: "tests can only show the presence of bugs, not their absence" [44]. Verification is sound—it provides a mathematical proof of correctness. But, extant verification approaches fail one or more of the other criteria: proof assistants require expertise developers do not have; SMT-backed analysis engines produce output most developers do not understand and are often imprecise, too slow, or both; type systems require too much manual effort to write type annotations, produce too many false positives, or both; abstract interpretations are either imprecise, too slow, or both. All of these approaches have been adopted in niches where their weaknesses are less relevant, but the only verification most software engineers encounter remains the static type systems of languages like Java.

We propose a middle ground: specialized typecheckers that can prove that a particular bug definitely does not occur in a given program. A specialized typechecker focuses on a narrow (but important) property, and therefore requires few annotations and can maintain high precision—while still guaranteeing the absence of the targeted error. The typechecker of a statically-typed languages like Java is *not* an example of such a typechecker: it deals with a broad class of errors at the cost of a significant annotation burden—every type in a Java program is a cost to the developer of the safety the type system provides. Like other verification technologies, the choice of what typechecker or typecheckers to deploy on a given program remains a major challenge for the developer, akin to deciding what specification to verify or what tests to write.

Our typecheckers improve on testing by being sound and nearly as easy for developers to use. The typecheckers improve on extant verification techniques by retaining their soundness, but improving the ease-of-use and practicality. Our work builds on previous work in pluggable typecheckers [23, 26, 46] by focusing on the virtues of simpler analyses. Simple analyses are easier for programmers to understand, require fewer annotations, have high precision, and can also cooperate to solve complex problems by each solving one sub-problem. They are not a panacea, however: to be effective, developers must still choose the right typecheckers to run for the most serious problems for their programs—and a collection of simple analyses must be developed to prevent those problems.

Our key insight is that specializing typecheckers to narrow problem domains makes them precise and usable enough to become

[1] No practical analysis can be perfectly sound and precise, because any non-trivial semantic property of a program is undecidable [49].

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3473105

practical for developers to use in their everyday work. Our contributions are the design, implementation, and evaluation of such specialized typecheckers for a variety of problem domains: array indexing (section 3.1), compliance (section 3.2), object construction (section 3.3), and resource leaks (section 3.4). The key insight behind the design of the checkers for the latter two domains—that a certain class of typestate analyses [53] which we call *accumulation analyses* can be efficiently implemented without a whole-program alias analysis, as was previously thought necessary for soundness. In our proposed work, we will use this insight to develop analyses for other domains traditionally analyzed using typestate analysis.

2 RELATED WORK AND BACKGROUND

We survey the most relevant techniques for detecting and preventing software bugs to our work, which we divide into two categories: unsound techniques such as testing and sound verification. Our approach aims for useful qualities from both: the usability and precision of unsound approaches and the soundness guarantees of sound approaches, albeit in limited scenarios. Our key insight is to start with the most usable form of verification we could identify—specialized typecheckers—and extend its capabilities to detect new kinds of defects while retaining or improving its usability.

Unsound techniques. Developers often validate that their programs work correctly by testing a fixed set of inputs or deploying high-precision, unsound bug-finding tools. These approaches are popular because they are easy for developers to use and usually find some bugs. However, they cannot provide a guarantee that a program is free of bugs, even bugs of a certain class.

Developers often write tests by hand. A test case consists of an input and an expected output (the *oracle*). Researchers have proposed many techniques for improving the efficacy of testing, including continuous testing [50] and continuous integration [28], fuzzing (e.g. [57]), and oracle generation from other development artifacts (e.g. [12]). Other dynamic analyses besides testing help programmers achieve correctness, including exceptions, invariant detection [24], and "hybrid" analyses that combine static and dynamic analyses, such as concolic testing [51]. Static analyses with high precision but no soundness guarantee have a similar trade-off. Examples include analyses based on manually-curated heuristics ("bug patterns") [8], on symbolic execution [39], or on sound analyses with relaxed restrictions to avoid common false positives [9]. Despite the effectiveness and popularity of testing and other unsound analyses, they offer no guarantees, unlike our work.

Verification and other sound techniques. Verification refers to any approach that allows a developer to prove that their code is consistent with some specification. We call such approaches *sound*, meaning that guarantee that successfully-analyzed code will not violate the specification when run (though in practice most implementations admit some unsoundness [41]). Approaches in this space include proof assistants such as Coq [10], which can verify arbitrarily-complex specifications but require expert users to guide the tool; translation of verification conditions to well-studied problems like satisfiability-modulo-theories (SMT) [21, 40] or graph reachability [48]; or dataflow analyses based on abstract interpretation [20] or type systems, which are equivalently expressive [19]. While different techniques have been successful in niches where their strengths outweigh their costs—e.g. translation to graph theory for taint analysis [13] or abstract interpretation for industrial control code [11]—they have not seen broad adoption among most developers. We believe that this lack of adoption stems from these approaches' usability and scalability: they require specialized knowledge to operate or debug failures, and often are not applicable to legacy code, requiring systems to be (re-)written to use them. Our goal is to maintain the soundness of these approaches, while improving their usability and scalability to make them more attractive to developers, and thus increase their impact on software quality in general.

Pluggable type systems. The most closely-related work to ours is on pluggable types—our work builds and expands upon prior work in this domain, and the specialized type systems described herein are a kind of pluggable type system. The notion of pluggable type qualifiers was first formalized in [26], who also prototyped a pluggable type system for C that enforced that `const` annotations were used correctly. The infrastructure for practical pluggable types was then developed over the next few years by the community [6, 15, 17, 27, 33, 42, 46]. The de-facto standard for Java is now the Checker Framework (checkerframework.org), on which we implemented our typecheckers. Other researchers have built many type systems with this framework to address general programming problems, including: nullness [23, 46], interning [23, 46], signature strings [23], compiler message keys [23], immutability [18, 23, 46], format strings [56], regular expressions [52], GUI effects [32], and others. Our work builds upon and is inspired by these and other pluggable type systems—it addresses problems that other pluggable type systems do not and focuses on the benefits of small, specialized checkers and how to deploy them effectively.

Pluggable typecheckers can be divided by generality. A *general* pluggable typechecker is widely applicable throughout a program—it prevents a class of bugs that could occur in many parts of the program. The Nullness Checker of the Checker Framework, which guarantees that every pointer dereference is non-null, is one example of a general typechecker: Java is an object-oriented language, so pointer dereferences are common. By contrast, a *specialized* typechecker handles a class of bugs that is restricted to a small part of the program text. Specialized typecheckers can be smaller and simpler while retaining high precision, because they need not reason about most of the program—though they verify that the whole program is well-typed, the subset of the program that could be mistyped is small. An extant example of a specialized typechecker is the Signature Checker of the Checker Framework, which verifies that the various kinds of names for Java classes used by different parts of the Java Virtual Machine and the Java compiler are not mistaken for each other. Most programs do not even use such names, and those that do typically use them sparingly. Generality is a spectrum: given two general typecheckers, we could argue that one is more general than the other. Generality is also subjective—we cannot yet precisely measure it. Nevertheless, it is a critical design principle of the work described in the next section.

3 COMPLETED WORK

3.1 Array indexing

An array access `a[i]` is in-bounds if $0 \leq$ `i` and `i` $<$ *length*(`a`). Unsafe array accesses are a common source of bugs. Their effects

include denial of service (via crashes or otherwise), exfiltration of sensitive data, and code injection. They are the single most important cause of security vulnerabilities [45]: buffer overflows enabled the Morris Worm, SQL Slammer, Code Red, and Heartbleed, among many others. A run-time system can prevent out-of-bounds accesses, but at the cost of halting the program, which is undesirable. Despite decades of research, preventing out-of-bounds accesses remains an urgent, difficult, open problem.

We have developed a system to prove safety of bounds checks—equivalently, to detect all possible erroneous array accesses—via a collection of type systems. Typechecking is a nonstandard choice for this problem. In previous attempts, types were too weak to capture the rich arithmetic properties required to prove facts about array indexing, could be hard to understand, and cluttered the code. The contribution of this section is to show that a carefully-designed collection of type systems—each specialized to a simple property—is an excellent fit to this general programming problem.

We have designed a set of seven lightweight, easy-to-understand specialized type systems. We implemented them in a tool, called the Index Checker, which provides the strong guarantee that a program is free of out-of-bounds array accesses, without the large human effort typically required for such guarantees. The Index Checker scales to and finds serious bugs in well-tested, industrial-size codebases: it found 89 bugs in three case studies (totaling 119,503 non-comment, non-blank lines of Java code), with 507 false positives; the bugs found included 5 priority-one[2] bugs in Google Guava. We directly compared the Index Checker to three other tools for detecting array-indexing bugs: the heuristic bug-finder FindBugs [8], the SMT-based theorem prover KeY [4], and the abstract-interpretation-based tool Clousot [25]. Our experiments showed that the Index Checker finds many more bugs than FindBugs and is more scalable than KeY or Clousot.

This work is described in greater detail in [35].

3.2 Verified compliance

Lightweight verification can support other tasks besides programming. One such task is *compliance*: the process of certifying that code follows a set of rules—a *compliance regime*—that encode industry-standard best practices. Large organizations—corporations, governments, etc.— require software to be compliant: processors of credit card transactions must comply with the PCI DSS [47], cloud vendors for the U.S. government must comply with FedRAMP [34], and SOC reports are used by many companies to evaluate the security posture of vendors [2]. Verification need not be an added task for developers: it can automate a task—preparing for compliance audits—that developers already perform.

Manual compliance audits of source code (the industry standard) are expensive, error-prone, partial, and prone to regressions. We have shown that many compliance controls—verifying properties including encryption key length, cryptographic algorithm selection, cloud data store initialization, and the absence of hard-coded credentials—can be audited using simple, specialized typecheckers, avoiding the weaknesses of manual audits.

[2]The highest priority is zero, but at the time the Guava team had never acknowledged a priority zero bug in their public repository

Auditors for AWS have accepted the output of our typecheckers as audit evidence, showing that our approach is practical. A security team ran two of our typecheckers as part of a large-scale security audit (which covers both compliance-relevant and non-compliance-relevant services) covering over 68 million lines of non-comment, non-blank Java code, finding 173 true positive warnings, 1 false positive warning, and requiring only 23 human-written type annotations. We also performed two open-source experiments using these typecheckers. First, we open-sourced our tools and ran them on 654 projects (about 5.7 million lines of non-comment, non-blank Java code) randomly selected from GitHub that used relevant APIs, finding that 24% of projects were verifiable, 39% of projects contained true positive warnings and 25% of projects contained false positive warnings (12% contained both, and 25% of projects were not compatible with our infrastructure or timed out). In the second, we compared our tools to other publicly-available tools for finding and preventing mis-uses of cryptographic APIs, using a benchmark from previous work [3]. We found that only our tools were sound (i.e. did not miss errors), and that our tools' precision was only slightly worse (97% vs 100%) than the best-in-class among the other tools. Since the cost of a false negative (i.e. missed alarm) in the compliance domain is high—a failed audit—only our tools are suitable replacements for manual compliance audits.

A full description of this work is available in [37].

3.3 Object construction via accumulation

The standard API for Java object construction contains one constructor for each combination of possible values that results in a well-formed object. Alternate patterns for object construction have been devised, such as the *builder pattern* [29]. To use the builder pattern, the programmer creates a separate "builder" class, which has two kinds of methods: *setters*, each of which provides a *logical argument*—a value that ordinarily would be a constructor argument; and a *finalizer* (often named `build`), which actually constructs the object and initializes its fields appropriately. The builder pattern improves readability and flexibility in client code, but it loses compile-time verification that logical arguments are provided. Popular frameworks like Lombok [54] and AutoValue [14] ease creation of builders by automatically generating a builder class from the class definition of the object to be constructed.

To verify that all required arguments are provided to builders, we have devised a modular typestate analysis [53], for the special case of an *accumulation analysis*. An accumulation analysis is a program analysis where the analysis abstraction is a monotonically increasing set, and some operation is legal only when the set is large enough—that is, the estimate has accumulated sufficiently many items. Accumulation analysis is a special case of typestate analysis in which (1) the order in which operations are performed does not affect what is subsequently legal, and (2) the accumulation does not add restrictions; that is, as more operations are performed, more operations become legal. The key advantage of our modular accumulation analysis over a traditional typestate analysis is that no expensive, whole-program alias analysis is required: aliasing information can improve precision, but is not required for soundness.

Our typechecker for the builder pattern accumulates calls to setter methods, and checks that all required setters have been invoked

when the finalizer is called. Our implementation automatically derives specifications for Lombok and AutoValue builders, which makes its annotation burden small. We evaluated it with case studies and a small user study. In a case study on a builder in the AWS API that, if mis-used by a client, can expose the client to a security vulnerability [43], we found 16 real vulnerabilities with just 3 false positive warnings (84% precision) and 34 human-written annotations in over 9.1 million lines of non-comment, non-blank Java code. In the small user study, we found that industrial developers who regularly use Lombok found it easier and faster to fix bugs resulting from changes in a Lombok-generated builder when they used our tool than when they used their usual debugging tools.

The full version of this work appears in [36].

3.4 Resource leaks

A resource leak occurs when a program allocates a resource, such as a socket or file handle, but fails to deallocate it. Resource leaks cause severe bugs, even in modern, heavily-used Java applications [30]. This state-of-the-practice differs little from two decades ago [55]. That resource leaks remain such a serious problem despite decades of research and improvements in languages and tooling shows that preventing them is an urgent, difficult, open problem.

We observe that detecting a resource leak for a variable involves three parts: 1) tracking its *must-call obligations*, 2) tracking which methods have been called on it, and 3) comparing the results of these to check if its obligations have been fulfilled. Our key insight is that (2) can be reduced to an accumulation problem (section 3.3), and that (1) and (3) can be solved using standard taint-tracking and gen-kill dataflow analyses, respectively. We developed a baseline leak checker via this approach.

The precision of any accumulation analysis can be improved by computing targeted aliasing information, and we devised three novel techniques that use this capability to achieve precision in practice: a lightweight ownership transfer system; a specialized resource alias analysis; and a system to create a fresh obligation when a non-final resource field is updated.

Our approach occupies a unique slice of the design space when compared to prior approaches: it is sound and runs quickly compared to heavier-weight approaches (running in minutes on programs that a state-of-the-art approach took hours to analyze). We implemented our techniques for Java in an open-source tool, which revealed 49 real bugs in widely-deployed software. It scales well, has a manageable false positive rate (similar to the high-confidence resource leak analysis built into the Eclipse IDE), and imposes only a small annotation burden (1/1500 LoC) for developers.

The full version of this work appears in [38].

4 PLANNED WORK

Sections 3.3 and 3.4 show that accumulation analysis is a powerful tool that allows specialized typecheckers to address problems that, in prior work, had required more expensive whole-program analyses. This section outlines our planned work on accumulation analysis. The key research question we want to answer is "**(RQ1)** what problems can accumulation analysis let us solve using specialized typecheckers?" This research question also invites some interesting other questions, including "**(RQ2)** are there other problems that, like resource leak analysis, are traditionally solved with a typestate analysis, but can be solved with an accumulation analysis?" and "**(RQ3)** are there bespoke analyses for important problems that, when examined, turn out to be accumulation analyses?"

One way to answer RQ1 is to examine what properties an accumulation analysis can express. For example, we know that an accumulation can express: "call method A before method B" properties (section 3.3), and "always call method A before deallocation" properties (section 3.4). But these properties only scratch the surface of what accumulation analysis can express. For example, both of these properties accumulate method calls, but in general an accumulation analysis could accumulate any kind of property. An example of accumulating something other than method calls is checking that a constructor actually initializes all non-null fields to non-null values: fields are accumulated. Initialization is a well-studied problem, and we suspect that existing initialization checkers might be accumulation analyses (i.e., might be part of the answer to RQ3).

Another example of accumulation is "def-use" properties of APIs. A *def-use property* has the form "if method A is called with argument P, then method B must be called with argument Q before method C is called (or deallocation)." One such property that can lead to subtle bugs when using AWS' DynamoDB API [22] is that the client must supply consistent *filter expressions* and *expression attribute names and values*. When querying the database, the client supplies both as strings; an accumulation analysis for each can accumulate what the client supplies, and the results of these two accumulation analyses can be compared before the query method is called. Other NoSQL database query languages might suffer from similar issues.

Dependency injection frameworks like Guice [31] provide logical arguments in other ways than calling methods directly (for example, by attaching `@Provider` annotations to classes), which can lead to mal-formed objects in the same way that mis-uses of builders can (section 3.3). Because we know that the underlying problem is accumulation of logical arguments, we could build a checker for a particular dependency injection framework by examining how it provides logical arguments.

To help answer RQ2, we plan to examine how typestate analysis is used in practice by examining programs written in typestate-oriented languages like Plaid [5] to determine which of the typestates used could be expressed as accumulation. We hypothesize that many (or even most) of the typestate systems used in real programs in such languages will be expressible as accumulation.

REFERENCES

[1] 2018. *ISSTA 2018, Proceedings of the 2018 International Symposium on Software Testing and Analysis*. Amsterdam, Netherlands.

[2] Bhargav Acharya. 2016. Why Cloud Providers Need a SOC Report. https://www.schellman.com/blog/why-cloud-providers-need-a-soc-report. Accessed 28 March 2019.

[3] Sharmin Afrose, Sazzadur Rahaman, and Danfeng Yao. 2019. CryptoAPI-Bench: A Comprehensive Benchmark on Java Cryptographic API Misuses. In *2019 IEEE Cybersecurity Development (SecDev)*. IEEE, 49–61.

[4] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. 2014. The KeY platform for verification and analysis of Java programs. In *VSTTE 2014: 6th Working Conference on Verified Software: Theories, Tools, Experiments*. Vienna, Austria, 55–71.

[5] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-oriented programming. In *OOPSLA Companion: Object-Oriented Programming Systems, Languages, and Applications*. Orlando, FL, USA, 1015–1022.

[6] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. 2006. A framework for implementing pluggable type systems. In *OOPSLA 2006, Object-Oriented Programming Systems, Languages, and Applications*. Portland, OR, USA, 57–74.

[7] John Anvik, Lyndon Hiew, and Gail C Murphy. 2006. Who should fix this bug?. In *International Conference on Software Engineering (ICSE)*. 361–370.

[8] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. 2008. Using static analysis to find bugs. *IEEE Softw.* 25, 5 (Sep. 2008), 22–29.

[9] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. 2019. NullAway: Practical type-based null safety for Java. In *ESEC/FSE 2019: The ACM 27th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Tallinn, Estonia, 740–750.

[10] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. 1997. The Coq proof assistant reference manual: Version 6.1. (1997).

[11] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A Static Analyzer for Large Safety-Critical Software. *SIGPLAN Not.* 38, 5 (May 2003), 196–207. https://doi.org/10.1145/780822.781153

[12] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications, See [1], 242–253.

[13] Eric Bodden. 2012. Inter-Procedural Data-Flow Analysis with IFDS/IDE and Soot. In *International Workshop on State of the Art in Java Program Analysis (SOAP '12)*. Association for Computing Machinery, New York, NY, USA, 3–8. https://doi.org/10.1145/2259051.2259052

[14] Kevin Bourrillion and Éamonn McManus. 2019. AutoValue with Builders. https://github.com/google/auto/blob/master/value/userguide/builders.md. Accessed 14 August 2019.

[15] Gilad Bracha. 2004. Pluggable type systems. In *RDL 2004: Workshop on Revival of Dynamic Languages*. Vancouver, BC, Canada.

[16] Alexia Chatzikonstantinou, Christoforos Ntantogian, Georgios Karopoulos, and Christos Xenakis. 2016. Evaluation of cryptography usage in android applications. In *International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS)*. 83–90.

[17] Brian Chin, Shane Markstrum, Todd Millstein, and Jens Palsberg. 2006. Inference of user-defined type qualifiers and qualifier rules. In *ESOP 2006: 15th European Symposium on Programming*. Vienna, Austria, 264–278.

[18] Michael Coblenz, Whitney Nelson, Jonathan Aldrich, Brad Myers, and Joshua Sunshine. 2017. Glacier: Transitive class immutability for Java. In *International Conference on Software Engineering (ICSE)*. IEEE, 496–506.

[19] Patrick Cousot. 1997. Types as abstract interpretations. In *POPL '97: Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Paris, France, 316–331.

[20] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*. Los Angeles, CA, 238–252.

[21] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *TACAS 2008: Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Budapest, Hungary, 337–340.

[22] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[23] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kıvanç Muşlu, and Todd Schiller. 2011. Building and using pluggable type-checkers. In *ICSE 2011, Proceedings of the 33rd International Conference on Software Engineering*. Waikiki, Hawaii, USA, 681–690.

[24] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1–3 (Dec. 2007), 35–45.

[25] Manuel Fähndrich and Francesco Logozzo. 2010. Static contract checking with abstract interpretation. In *International Conference on Formal Verification of Object-Oriented Software*. Paris, France, 10–30.

[26] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. 1999. A theory of type qualifiers. In *PLDI '99: Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*. Atlanta, GA, USA, 192–203.

[27] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. 2002. Flow-sensitive type qualifiers. In *PLDI 2002: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. Berlin, Germany, 1–12.

[28] Martin Fowler and Matthew Foemmel. 2006. Continuous integration.

[29] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. 1995. *Design Patterns*. Addison-Wesley, Reading, MA.

[30] Mohammadreza Ghanavati, Diego Costa, Janos Seboek, David Lo, and Artur Andrzejak. 2020. Memory and resource leak defects and their repairs in Java projects. *Empirical Software Engineering* 25, 1 (2020), 678–718.

[31] Google. 2006. Guice. https://github.com/google/guice. Accessed 23 August 2019.

[32] Colin S. Gordon, Werner Dietl, Michael D. Ernst, and Dan Grossman. 2013. JavaUI: Effects for controlling UI object access. In *ECOOP 2013 — Object-Oriented Programming, 27th European Conference*. Montpellier, France, 179–204.

[33] David Greenfieldboyce and Jeffrey S. Foster. 2007. Type qualifier inference for Java. In *OOPSLA 2007, Object-Oriented Programming Systems, Languages, and Applications*. Montreal, Canada, 321–336.

[34] GSA. 2017. FedRAMP SECURITY ASSESSMENT FRAMEWORK, Version 2.4. https://www.fedramp.gov/assets/resources/documents/FedRAMP_Security_Assessment_Framework.pdf. Accessed 31 January 2019.

[35] Martin Kellogg, Vlastimil Dort, Suzanne Millstein, and Michael D. Ernst. 2018. Lightweight verification of array indexing, See [1], 3–14.

[36] Martin Kellogg, Manli Ran, Manu Sridharan, Martin Schäf, and Michael D. Ernst. 2020. Verifying Object Construction. In *ICSE 2020, Proceedings of the 42nd International Conference on Software Engineering*. Seoul, Korea, 1447–1458.

[37] Martin Kellogg, Martin Schäf, Serdar Tasiran, and Michael D. Ernst. 2020. Continuous compliance. In *ASE 2020: Proceedings of the 35th Annual International Conference on Automated Software Engineering*. Melbourne, Australia.

[38] Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. 2021. Lightweight and Modular Resource Leak Verification. In *Foundations of Software Engineering (FSE)*.

[39] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[40] K. Rustan M. Leino and Greg Nelson. 1998. An extended static checker for Modula-3. In *CC '98: Compiler Construction: 7th International Conference*. Lisbon, Portugal, 302–305.

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

[42] Shane Markstrum, Daniel Marino, Matthew Esquivel, Todd Millstein, Chris Andreae, and James Noble. 2010. JavaCOP: Declarative pluggable types for Java. *ACM TOPLAS* 32, 2 (Jan. 2010), 1–37.

[43] MITRE. 2018. CVE-2018-15869. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-15869.

[44] Peter Naur and Brian Randell. 1969. Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th-11th October 1968. (1969).

[45] Serkan Özkan. 2018. CVE Details. https://www.cvedetails.com/vulnerabilities-by-types.php. Summary of http://cve.mitre.org/.

[46] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. 2008. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*. Seattle, WA, USA, 201–212.

[47] PCI Security Standards Council. 2018. Payment Card Industry (PCI) Data Security Standard, v. 3.2.1. https://www.pcisecuritystandards.org/documents/PCI_DSS_v3-2-1.pdf. Accessed 26 February 2019.

[48] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Principles of programming languages (POPL)*. 49–61.

[49] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366.

[50] David Saff and Michael D. Ernst. 2003. Reducing wasted development time via continuous testing. In *ISSRE 2003: Fourteenth International Symposium on Software Reliability Engineering*. Denver, CO, 281–292.

[51] Koushik Sen. 2007. Concolic Testing. In *Automated Software Engineering (ASE) (ASE '07)*. Association for Computing Machinery, New York, NY, USA, 571–572. https://doi.org/10.1145/1321631.1321746

[52] Eric Spishak, Werner Dietl, and Michael D. Ernst. 2012. A type system for regular expressions. In *FTfJP: 14th Workshop on Formal Techniques for Java-like Programs*. Beijing, China, 20–26.

[53] Robert E. Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE* SE-12, 1 (Jan. 1986), 157–171.

[54] The Lombok Authors. 2019. @Builder. https://projectlombok.org/features/Builder. Accessed 12 February 2019.

[55] Westley Weimer and George C Necula. 2004. Finding and preventing run-time error handling mistakes. In *Object-oriented programming, systems, languages, and applications (OOPSLA)*. 419–431.

[56] Konstantin Weitz, Gene Kim, Siwakorn Srisakaokul, and Michael D. Ernst. 2014. A type system for format strings. In *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*. San Jose, CA, USA, 127–137.

[57] Michal Zalewski. 2014. American fuzzy lop. https://lcamtuf.coredump.cx/afl/. Accessed 27 May 2020.