Lightweight Verification via Specialized Typecheckers

Martin Kellogg

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2022

Reading Committee:

Michael D. Ernst, Chair

René Just

Zachary Tatlock

Program Authorized to Offer Degree:

Computer Science & Engineering

©Copyright 2022
Martin Kellogg

University of Washington

Abstract

Lightweight Verification via Specialized Typecheckers

Martin Kellogg

Chair of the Supervisory Committee:
Professor Michael D. Ernst
Paul G. Allen School of Computer Science & Engineering

Software defects can cause severe damage, because software is ubiquitous in the modern world. Software testing cannot find all defects. Full formal verification, though powerful, remains too difficult and expensive for most software engineering projects.

Lightweight verification is a promising middle ground between testing and full formal verification that permits developers to prove the absense of particular kinds of defects with low overhead. Proving the absense of such defects improves the reliability, correctness, and security of software. Lightweight verification enables working software engineers to begin to use verification tools, paving the way toward a future in which verification is a standard part of every developer's toolkit.

In this dissertation, we describe novel contributions to lightweight verification via the use of specialized pluggable typecheckers. Our contributions are in two categories: (1) new techniques that increase the *expressiveness* of lightweight verification by making verification simpler or cheaper for particular classes of problems— thus making verification of the absence of those problems *more lightweight*—and (2) *impact* on real developers by applying specialized typecheckers to new domains.

Our first contribution is the theory of accumulation analysis, which demonstrates that alias analysis—the key bottleneck in a traditional typestate analysis—is not necessary for 41% of typestate specifications in a literature survey, meaning that those 41% of specifications can be checked using a lightweight accumulation analysis instead of an expensive traditional typestate analysis. We have implemented several accumulation analyses, including for two specific classes of problems traditionally addressed with typestate—initialization and resource leaks. We have shown that these specialized typecheckers implementing accumulation analyses are effective tools for lightweight verification: they are sound (that is, doing verification rather than bug-finding), fast (running in minutes on commodity hardware), and as precise as the unsound, heuristic-based static analyses commonly employed by developers.

Our second contribution is a collection of specialized typecheckers for proving the absense of out-of-bounds array accesses. Our typecheckers achieve similar results as an expensive

SMT-backed analysis in an order of magnitude less time, increasing the practicality of array-bounds verification.

Our third contribution is a collection of specialized typecheckers for proving the absence of certain violations of compliance rules. Lightweight verification is a novel technique in the domain of compliance certification which achieves significant impact: developers prefer lightweight verification to state-of-the-practice manual audits.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Figure Number Page

LIST OF TABLES

Table Number Page

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

This dissertation covers the work I've done throughout my PhD, none of which would have been possible without my excellent collaborators: Vlastimil Dort, Suzanne Millstein, Manli Ran, Manu Sridharan, Martin Schäf, Serdar Tasiran, Narges Shadab, and others. A special thanks to my advisor, Michael D. Ernst, whose guidance and advice has been invaluable through my PhD journey.

Thanks to my committee: Michael D. Ernst, René Just, Zachary Tatlock, Martin Schäf, and Nic Weber.

Thanks to my partner, Bethany Connor, for putting up with my antics at home. Thanks to my friends, family, and colleagues—especially the PLSE lab—for putting up with my antics everywhere else.

Chapter 1

INTRODUCTION

Software is everywhere, but *correct* software is not. Heavily-used projects include known bugs [16]. Most (90% or more) Java applications that use cryptography misuse it [63]. Despite decades of research, buffer overflow vulnerabilities remain one of the most common causes of security issues [234].

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

The platonic ideal bug-prevention technique would have the following characteristics:

- *soundness*: it would never certify a program that contains bugs.
- *precision*: it would always certify a program that does not contain bugs[1].
- *usability*: it would be easy for developers to use. Ideally, it would be fully automated.
- *comprehensibility*: it would be easy for developers to understand why it rejects a program, so that they can fix the problem easily.
- *efficiency*: it would require no time or space overhead at run time, and would be fast to run at development time.
- *applicability*: it would run on existing code without requiring that code to be modified.

Achieving all of these criteria in practice is impossible, so extant techniques trade-off between them.

Testing and unsound static analyses give up on soundness, even with respect to the bugs they are attempting to catch, to achieve most of the other criteria. Testing is popular due to its strengths in the other categories:

[1] No practical analysis can be perfectly sound and precise, because any non-trivial semantic property of a program is undecidable [261]. Nevertheless, both soundness and precision are important goals for any bug-prevention technique.

- tests often have high precision, though in practice it is common for tests to fail even when no defect is present (e.g., because of flaky tests [209]);

- tests are relatively easy for developers to write (though developers often do not write enough tests, motivating work in test input generation, e.g., [235, 64]);

- tests are understandable when they fail (though developers prefer to debug from smaller test cases, which motivates academic work on test case minimization, e.g., [201]); and

- tests do not require run time overhead (even if testing time is sometimes a problem [105]).

The fundamental lack of soundness makes it impossible, however, for a testing technique to ever achieve the platonic ideal described above: no testing technique will ever guarantee that it finds *every* defect in a program. For example, even if a developer writes a test that a feature behaves correctly, in reality only the feature's behavior on the chosen inputs is tested, and any other input could always be buggy. While it is theoretically possible for a sound technique to do as well as testing along the other axes of the platonic ideal, it is theoretically *impossible* for any testing technique to be sound. This theoretical gap—and the need, in safety-critical domains, to be very sure that a program is free of bugs—motivates research on sound techniques such as verification.

Verification approaches provide mathematical proofs of correctness: thus, they are sound. Any verification tool is only as good as its definition of "correctness," however: a human must choose the correct specification to verify, and practical concerns mean that verification tools are usually sound with some caveats [206]. Every extant verification approach also fails at least one of the other criteria: proof assistants require expertise that most developers do not have; SMT-backed analysis engines produce output that is hard to understand, and they are often imprecise, slow, or both; type systems require much manual effort to write type annotations, produce many false positives, or both; abstract interpretations are either imprecise, slow, or both. All of these approaches have seen some adoption in niches where their weaknesses are less relevant: for example, abstract interpretation is used to verify properties of spacecraft control software [49]. Nevertheless, *lightweight verification* remains out of reach. A lightweight verification tool is any technique that does verification by mathematical proof and approaches the platonic ideal along its other dimensions: does not produce too many false positives, is easy for developers to use and understand, runs quickly, and requires few modifications to legacy code. Lightweight verification is itself something of a platonic ideal: a goal towards which verification research can be oriented. For typical software engineers, the static type systems of languages like Java or Rust are the only verification tools they encounter—and even these relatively popular verification tools do not fully achieve all the criteria of the platonic ideal of lightweight verification: they require a great deal of user effort (in the form of type annotations, restrictions on aliasing, etc.) and are not applicable legacy code written in other languages.

In this dissertation, we propose an approach to building lightweight verification tools—*specialized pluggable typecheckers*—that produces sound verifiers with distinct advantages over prior work in verification on the other criteria of the platonic ideal. A specialized typechecker focuses on a narrow (but important) property, and therefore requires few type annotations and can maintain high precision—while still guaranteeing the absence of the targeted error[2]. 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.

The key insight behind our approach to designing specialized pluggable typecheckers is to design explicitly for *simplicity*: the other desirable properties of verifiers are often *consequences* of the simplicity of the right abstraction. Simple analyses run quickly. Developers can easily understand their results, which makes them scalable and predictable. However, a simple analysis is not necessarily simple to design. In fact, it is often more difficult to find elegant solutions to real, complex problems: a core theme in this dissertation is decomposing complex problems into cooperating sets of simple analyses.

Simple specialized 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 [128, 237, 96] by focusing on the virtues of simpler analyses. Simple analyses 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 contributions improve the state-of-the-art in lightweight verification in two important ways:

- *Expressiveness.* We have developed new specialized typecheckers that enable lightweight verification for problems that traditionally require heavier-weight analyses.

- *Impact and Access.* We have developed specialized typecheckers for new domains where they replace other techniques for ensuring correctness, demonstrating to developers that lightweight verification is not just an ideal but actually practical.

In particular, this dissertation contributes the following:

Accumulation analysis (Chapter 3). Typestate protocols [284] are finite-state machines that describe the operations that are legal and illegal in an object's various states. For example, a file in an Open state might have legal Read and Close operations, but a file in a Closed state might only have a legal Open operation. In general, sound typestate checking is computationally expensive due to the need to reason about aliasing. In practice, this

[2] 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.

cost has been a barrier to the adoption of typestate verification. However, some typestate properties are monotonic: the set of legal operations only grows as the object transitions through typestates. We discovered and proved that these monotonic typestates can be checked soundly and modularly, without needing to reason about aliasing, using a family of simple analyses that we call *accumulation analyses*. An accumulation analysis conservatively under-approximates the set of typestate transitions that have definitely occurred and forbids goal transitions until the analysis can prove that all of their enabling transitions must have occurred. The next two paragraphs give two examples of monotonic typestate properties that we have verified soundly and modularly using accumulation analyses; in a literature survey, we found that 41% of typestate problems in the literature could be checked by accumulation analyses.

Malformed object construction (Section 3.5). When an object is constructed, some set of logical parameters must be provided. For example, a geometric point object might require both x and y values, but its color might be optional. The popular builder design pattern [137]—where each logical argument has its own method on a "builder" object, and the final object is only created when the builder's `build` method is called—enables programmers to avoid defining exponentially-many constructors. The builder pattern is convenient for programmers, but using it does cost some compile-time safety. Without the builder pattern, the programmer would not have written a constructor that took no `x` value; with the builder pattern, the programmer might forget to call a logically-required setter method, such as `setX()` in the point example, before calling `build`. We designed an accumulation analysis whose goal transition is `build` and whose enabling transitions are exactly the methods that set the required logical parameters. This restores compile-time safety when using the builder pattern. In a user study of AWS developers, those using our approach were about 50% faster and about 50% more likely to correctly update all call-sites. Security vulnerabilities can also result if the missing parameter was necessary for safety. In 9.2 million lines of code, our tool found 16 real security bugs with just 3 false positives (84% precision), but needed just 34 manually-written annotations (1 per 250,000 lines of code).

Resource leaks (Section 3.6). After a program allocates a programmer-managed resource, such as a file descriptor, a network socket, or a database connection, the program must release the resource on all paths. Failing to do so causes a resource leak, which can cause resource starvation or denial-of-service, especially in long-running applications. Our key insight is the monotonicity of the property: all resources must be closed at least once. Our approach to solving this problem combines three simple analyses: (1) a taint analysis that tracks which expressions might contain objects that need to be closed, (2) an accumulation analysis whose goal transition is "a resource goes out of scope" and whose enabling transition set is `close()` , and (3) an analysis that compares the previous two when an expression may go out of scope. Our approach is sound, fast, and precise. It outspeeds traditional approaches that track all aliasing by orders of magnitude and is competitive with unsound bug-finders on precision. On the benchmarks we tested, our analysis improved slightly upon the precision of the analysis built into the Eclipse IDE (29% vs 25%) while dominating on recall (100% vs 13%). Our approach is also usable: it required only 286 manually-written annotations in over 400,000 lines of code (about 1 for every 1,500 lines) in distributed-systems infrastructure that made heavy use of resources that must be manually closed.

Array bounds (Chapter 4). We designed a set of seven cooperating verifiers to prove that array accesses are in-bounds. This shows how expressive simple analyses can be when combined in the right way. Prior monolithic approaches attempted to reason about all the complexity of bounds-checking using one complex abstraction, such as systems of arbitrary linear inequalities over program variables. Our abstractions decompose the complexity into multiple novel dependent type systems, simple linear inequalities, and more. The way they interact was also novel: the analyses are carefully staged to avoid mutual dependence except where necessary for precision, which we limited to one case in the rely-guarantee style. Our system is usable: it has a lower annotation burden than Java's generics. It is precise: similar to the best extant monolithic approaches based on abstract interpretation. And it is fast: it analyzes large programs in minutes rather than hours.

Compliance (Chapter 5). Another way to make verification more attractive to developers is to automate a manual task that developers already have to do. An example is compliance, a process common in industry whereby an external auditor affirms that a company's systems properly handle sensitive data. For example, credit card companies require that companies holding credit card data must follow the PCI DSS (Payment Card Industry Data Security Standard), which has requirements like "credit card data be encrypted while it is stored." In practice, these audits involve manual examination of code by the auditor. We realized that many of these properties could be expressed as simple refinement type systems, and we designed and deployed them. Auditors at an industrial partner accepted the output of our verifiers, obviating the need for manual audits of those properties, and they presented the results at a developer conference. Developers preferred our approach—using the typechecker was less work for them than a single manual audit—as did the auditors, because our sound checks eliminated the possibility of human error. We ran our analyses on 76 million lines of code and found 173 true violations with only 1 false positive (99% precision) while requiring only 23 manually-written annotations (1 per 3.3 million LoC). We also compared our analyses to extant unsound bug finders on an existing benchmark: our tools found all the errors (i.e. had 100% recall, vs. 88% for the next best tool) with comparable precision (our tools had 97% precision vs. 100% for the best unsound bug-finder).

Chapter 2

RELATED WORK AND BACKGROUND

2.1 Related work

This section gives an overview of existing approaches to preventing bugs in software. Section 2.1.1 focuses on unsound approaches: testing, dynamic analyses, and static analyses that aim for high precision without offering guarantees about preventing all bugs of a certain class. Section 2.1.2 focuses on sound verification technologies, especially focusing on those that are relatively scalable.

Both categories of techniques approach the platonic ideal of bug prevention we described in chapter 1 from different directions. Sound tools *do* provide such a guarantee, but usually fall short on one or more of the other criteria (precision and ease of use are the most common). Unsound tools usually excel at one or more of the other criteria (precision and ease of use are the most common), but an unsound technique can never provide a guarantee that all bugs have been resolved.

Lightweight verification aims for useful qualities from both: the usability and precision of unsound approaches and the soundness guarantees of sound approaches. The key insight in our approach is that simple, specialized typecheckers can be similarly precise and easy-to-use as common unsound techniques, but still provide a soundness guarantee. That does not mean that our lightweight verifiers perfectly match the platonic ideal: the specifications that a lightweight verifier like the ones we describe in this dissertation can verify are limited. One of our goals throughout this work has been to expand the space of what specifications can be verified in a lightweight way.

Sections 3.5.5, 3.6.8, 3.8, 4.6, and 5.11 give related work for typecheckers described in this document; this section overviews the larger research area and places our work within it.

2.1.1 Testing and other unsound analyses

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

The key advantage of these techniques is that developers actually do use them. They therefore give us a target: if a lightweight verification tool is as precise and as easy to use as some unsound technique that developers do already use, we can be confident that eventually the lightweight verifier will see adoption: a sound tool or technique is strictly better than an unsound one if it is no worse along the other dimensions of the platonic ideal.

2.1.1.1 Testing

Developers often write test cases by hand. A test case consists of an input and an expected output (the *oracle*). Testing is relatively easy to use, and writing tests is a basic requirement for high-quality software engineering. Tests are also usually high-precision: good tests are deterministic and only fail when there is actually a defect, though *flaky tests*—tests that non-deterministically fail—can cause imprecision (and identification of flaky tests is an active research area [209, 29, 190]). However, tests only provide guarantees that the software under test behaves correctly *on the inputs that are actually tested.* Exhaustive testing (i.e., testing on every possible input) is impractical for realistic systems, so it is *impossible* for testing to provide guarantees about all executions. One goal of our lightweight verification work is to supplant testing: if it is as easy to verify a property as it is to test for it, using a verification tool is always the superior choice.

Researchers have proposed many techniques for improving the efficacy of testing. A test is only useful if it is actually run when a defect is introduced; continuous testing [265] and continuous integration [131] are approaches to automated testing that run regression tests after every code change (defined as keystrokes and commits, respectively). Continuous integration is ubiquitous in industrial development. These techniques attempt to (partially) mitigate the unsoundness of testing: they improve the probability that an actual defect is detected when it is introduced.

Test cases can be generated randomly and used to build regression test suites [262] or to *fuzz* programs: provide random input to the program and monitor it for violations of oracles that should apply to all programs, such as “this program should not crash.” Researchers and practitioners have developed general security fuzzers (e.g. AFL [322]) as well as specialized fuzzers for particular domains (e.g., Zest for highly-structured input [236]). A related technique is property-based testing [68], which generates inputs that match some specification written by the programmer.

The techniques described in the previous paragraph all generate test inputs and rely on programmer-specified or generally-applicable oracles; the problem that they address is that despite tests being relatively easy to write, developers still do not write enough of them. There is also research on generating oracles. For example, oracles can be generated from documentation (e.g. Javadoc comments in Java code [144, 38]).

2.1.1.2 Dynamic analyses

A dynamic analysis runs the program at least once. Testing is the most common kind of dynamic analysis, but there are others. Because dynamic analyses (including testing) generalize from information gathered from specific executions with fixed inputs, they are usually unsound but precise: they produce information that is true about the inputs they have observed but may not generalize to other inputs.

Raising exceptions in the runtime is a good example of a dynamic analysis that *is* sound: on any erroneous execution, the runtime causes the program to crash rather than continue with an execution that is demonstrably incorrect. Java, for example, raises exceptions for many common errors, including null pointer dereferences, division by zero, and out-of-bounds array accesses [148]. While these exceptions prevent some problems, such as buffer overflow attacks due to out-of-bounds array accesses, they still cause the program to crash—possibly causing other problems such as denial of service. Our typecheckers catch defects before the program is ever executed, preventing the problem before it occurs.

Other unsound dynamic analyses can also aid in achieving correctness. Invariant detectors such as Daikon [115] observe execution to establish a set of facts that are true on all inputs ("invariants"); if a future execution violates an invariant, it may indicate a bug [154]. Other approaches mix static and dynamic approaches. For example, concolic testing combines dynamic test executions ("concrete executions") with symbolic execution (see section 2.1.1.3) [270]. Like testing, these approaches provide no guarantees.

2.1.1.3 Unsound static analysis

Static analyses with high precision but no soundness guarantee are widely used, demonstrating the value of precise, fast, and easy-to-use static analyses. Unsound static analyses are therefore a useful benchmark against which to compare lightweight verifiers: if a lightweight verifier can match (or come close to matching) an unsound static analysis on the other criteria for the ideal bug-prevention tool, the lightweight verifier is clearly superior.

One category of unsound static analyses is based on manually-curated heuristics ("bug patterns"). Examples of these techniques include FindBugs [19] and its successors, as well as commercial tools like Fortify [215]. Another category of unsound static analyses are analyses that are guaranteed to be *complete*: that is, they (provably) have no false positives [232].

Most other unsound tools are derived from sound analyses, but with some restrictions relaxed to avoid common false positives. Some analysis tools are based on symbolic execution [182], which derives path constraints for expressions in the program that may lead to errors, and then checks if those constraints are feasible using a solver. Though symbolic execution could theoretically be done soundly, extant tools are unsound. The most popular commercial analyzer of this type is Coverity [31]. NullAway [21] is another example of an unsound tool derived from a sound one: it relaxes the restrictions of a pluggable type system for nullity (section 2.1.2.5). Another example is CryptoGuard [258], which relaxes an IFDS-backed analysis (section 2.1.2.2) using a slicing algorithm. Another example is AWS's RAPID [108], which is an intentionally-unsound typestate analysis (section 2.1.2.4) that does not track all possible aliasing in the program being analyzed. Like testing, these analyses provide no guarantees. We compare our lightweight verifiers to relevant tools from this category in sections 3.6.6.6, 4.4, and 5.7.

2.1.2 Verification

Full formal verification usually requires human intervention, often via a proof assistant such as Coq [24], Isabelle/HOL [231], or Lean [79]. Using such systems, building and verifiying a realistic system is a research project in and of itself; nevertheless, researchers have built and verified a C compiler [202], an operating system [184], and more. Though these tools hold great promise and can construct proofs with ironclad guarantees, they correspondingly require specialized expertise in writing proofs and are rarely applicable to legacy code, making them unsuitable for use by working programmers. The remainder of this section therefore focuses on verification approaches that construct proofs automatically (even if they require user-written specifications, such as type annotations) and are applicable to legacy code: automated theorem proving via translation to SMT solvers and graph reachability, abstract interpretation, typestate systems, and pluggable type systems.

2.1.2.1 Automated theorem provers

Automated theorem provers reduce analysis problems derived from code to standard problems for which the research community has developed specialized solvers.

One class of such automated theorem provers is based on reduction to satisfiability-modulo-theories (SMT). The community has developed powerful SMT solvers, such as Z3 [78], which are often the backend for these kind of analyses. Extended Static Checking [199, 83, 126], KeY [9], Dafny [198], and other similar tools are examples of analysis tools based on this paradigm. This is the dominant paradigm in bounds verification and in some other types of program analysis. These tools suffer brittleness or instability: a small, meaning-preserving change within a method implementation may change the tool's output from "verified" to "failed" or "timeout", or might lead to different diagnostics in unrelated parts of the program [200, 156]. Scalability and usability are also challenges. We compare our typecheckers for array bounds directly to a tool from this category in section 4.4. More generally, these approaches are superior for problems where the code being analyzed is unlikely to change—meaning that their brittleness is less of a concern—and is relatively small, but the property to be proved is both complex and easy to express in the language of an SMT solver—that is, has many boolean conditions. These requirements make it difficult (but not impossible!) to develop lightweight verifiers based on this paradigm.

2.1.2.2 Dataflow analysis

Dataflow analyses such as IFDS [260] and IDE [266] are the dominant paradigm in taint analysis, especially of Java programs [42]. These analyses convert dataflow problems of a particular class—the set of dataflow facts must be a finite set, and the dataflow functions must distribute over the confluence operator (i.e. joins)—into graph reachability problems, allowing them to be solved efficiently. Many classic dataflow problems fall into this class, including constant propagation, live variables, and variable initialization. Arbitrary typestate [284, 223]

```
public static final String DEFAULT_CRYPTO = "RC2";
private static char[] CRYPTO;
private static char[] crypto;
public void go() {
    KeyGenerator keyGen = KeyGenerator.getInstance(String.valueOf(crypto));
    SecretKey key = keyGen.generateKey();
    Cipher cipher = Cipher.getInstance(String.valueOf(crypto));
    cipher.init(Cipher.ENCRYPT_MODE, key);
}
private static void go2(){ CRYPTO = DEFAULT_CRYPTO.toCharArray(); }
private static void go3(){ crypto = CRYPTO; }
public static void main (String [] args) {
    BrokenCryptoABICase8 bc = new BrokenCryptoABICase8();
    go2(); go3(); bc.go();
}
```

Figure 2.1: An example of a false negative for CogniCrypt/CrySL [188] from CryptoAPIBench [8] that shows an unsoundness of modern IFDS/IDE-backed analyses; note the flow through static character arrays. “RC2” is an unsafe cryptographic algorithm, so it should not be used as the argument for `Cipher.getInstance`. Our type-based analysis for compliance (chapter 5) is sound, including on this example.

and information flow [43] problems are also expressible as IFDS or IDE problems. Mature analysis tools that implement IFDS and/or IDE for Java include Soot/Heros [299, 42] and WALA [95].

Examples of notable, recent IFDS- or IDE-backed analyses in the security domain include FlowDroid [18], which analyzes Android applications for information leaks, and CogniCrypt/CrySL [188], which analyzes Java code for misuses of cryptography. IFDS and IDE analyses can, in theory, be sound [260], but these analyses do suffer false negatives in practice. For example, section 5.7 demonstrates that CogniCrypt/CrySL is unsound even on relatively simple microbenchmarks; see fig. 2.1 for an example.

In addition to the direct unsoundness in the figure, this example shows another weakness of these analyses: they require the analysis user to specify the entry-points of the program. In simple cases like the program in the figure, the entry point (`main`) can be inferred. For libraries with many possible entry points, however, specifying or inferring entry points can require human intervention and is another possible source of unsoundness, because code that is not reachable from an entry point is not analyzed. By contrast, our typecheckers analyze and verify all source code, regardless of whether it is obviously reachable from an entry point.

These kind of dataflow analyses are most effective when the entry points are fixed and the property to be proved commonly involves reasoning across module boundaries (as they are whole-program analyses by nature). Taint-tracking or information flow problems are notable examples of such domains. In those domains lightweight verifiers based on this paradigm might be possible, but even in those domains extant tools are unsound [14, 256].

2.1.2.3 Abstract interpretation

Abstract interpretation [73] is a general framework for designing and formalizing program analyses. Abstract interpretation-based systems have been successfully deployed to prove the absence of array bounds errors, overflow, and division by zero in an industrial setting on a restricted subset of C [36, 37]. Extensions to the abstract domains used therein form the backbone of Clousot [120, 192], a verifier for arbitrary C# code; we compare our cooperating type systems for array bounds checking to Clousot in section 4.4, and section 4.6 contains a more extended discussion of Clousot.

In general, abstract interpretation is a promising approach for building lightweight verifiers. Our choice of pluggable typecheckers versus abstract interpretation is mostly one of convenience; the type systems defined in this work all have equivalent abstract interpretations, because abstract interpretation and type systems are equally expressive [72]. The advantage of our approach is our focus on designing for simplicity and composability.

2.1.2.4 Typestate

A typestate [284] system permits the type of an object to change as a result of operations in the program. For example, in a typestate system, a chess piece's type might change from `Pawn` to `Queen`, or a file's type might change from `UnopenedFile` to `OpenedFile` to `ClosedFile`. File operations like `read()` are permitted only on an `OpenedFile`. Fully typestate-oriented languages have been proposed [11]. Typestate-like properties can be translated to IFDS/IDE and enforced by a dedicated solver [223].

Arbitrary typestate analysis requires a precise alias analysis for soundness; because whole-program alias analyses with sufficient precision scale poorly—taking on the order of hours for practical programs [289], despite significant research investment in the problem—typestate analysis is usually considered impractical. To see why aliasing information is necessary, consider a file object of type `OpenedFile` with two aliases: `f` and `g`. If `f.close()` is called, the type of `f` changes to `ClosedFile`. The type of `g` also must change in tandem, or the type system might permit a later call to `g.close()` that would result in an error. Our work on accumulation analyses (chapter 3) shows that a subset of traditional typestate properties can be soundly checked without a precise alias analysis using a simple typechecker, making them dramatically more practical: the key bottleneck (the whole-program alias analysis) is avoided.

2.1.2.5 Pluggable types

The most closely-related work to that described herein is the prior work on pluggable types—our work builds upon 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 [128], 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 [129, 130, 149, 66, 67, 54, 15, 213, 237]. The de-facto standard for Java is now the Checker Framework [88]. Researchers have built many type systems, including: nullness [237, 96], interning [237, 96], signature strings [96], compiler message keys [96], immutable types [96, 237, 70], locking discipline [114], format strings [309], regular expressions [277], GUI effects [147], Android taint-tracking [113], and many 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 address and focuses on the benefits of small, specialized checkers and how to deploy them effectively.

2.2 Background on Pluggable Type Systems

This section defines common terms used in the rest of the work, and outlines some common properties of the analyses described later, to avoid repetition.

A *type* is a set of run-time values: an expression's compile-time type is an overestimate of all its possible run-time values. Typechecking is a dataflow analysis that produces sound estimates of what a program may compute. Every variable has one or more types from each running type system at every point in a program. Every analysis described in further chapters is implemented as a type system, unless stated otherwise.

A *type qualifier* [130] refines a type by restricting the set of values it represents, meaning a qualified type is a subtype of the same unqualified type. Essentially, it is a separate type system that can be mixed into a base type system. The implementations of our typecheckers use Java's type annotations to represent type qualifiers in Java source code. For instance, in the variable declaration `@Positive int i`, the type is `@Positive int`, which contains fewer values than `int` and is therefore a subtype of `int`.

Each type system is modular and runs on one method at a time. Programmers write type qualifiers on fields and method signatures (formal parameter and return types). Our type systems infer almost all types within method bodies (though they still require programmer intervention to e.g., add a type qualifier to the component type of a list).

All of our type systems are flow-sensitive. For example, after a test `x.f > 0`, the type of `x.f` is `@Positive` until a possible side effect or a control flow join. Effectively, this means that each specialized typechecker can be viewed as an abstract interpretation [72]; each type qualifier is equivalent to an abstract value.

All our typecheckers are implemented using the Checker Framework [237] (`https://checkerframework.org/`), an industrial-strength, open-source tool for building Java type systems that is used at companies such as Amazon, Facebook, Google, and Uber. The framework abstracts some details of the analysis implementation (e.g., by modeling the heap) and automatically supports features such as flow-sensitive local type inference, Java generics, and qualifier polymorphism.

Each typechecker contains a definition of the type hierarchy, type rules, and inference rules. The type and inference rules are implemented directly, without calling an external solver. This keeps performance fast and predictable, and can be more expressive, at the cost of an increase in implementation size. Because our typecheckers are specialized to narrow properties, the increase in implementation size is acceptable.

Chapter 3

LIGHTWEIGHT VERIFICATION VIA ACCUMULATION ANALYSIS

3.1 Motivation

A typestate specification [284] associates a finite-state machine (FSM) with program values of a given type. As a value transitions through the states of the FSM, different operations are enabled or disabled; that is, the FSM encodes a behavioral specification for the type.

A typestate analysis checks that a program follows a typestate specification—that is, the program does not attempt to perform a disabled operation. Typestate analyses are well-studied in the literature, and have been deployed for many purposes, including enforcing a locking discipline [138, 82], verification of Windows device drivers [62], and preventing security vulnerabilities [246]. However, *sound* typestate analyses—those with no false negatives—are rarely deployed in practice; for example, a recent paper [108] describing how AWS has deployed a typestate-based analysis at cloud-scale explicitly omits soundness as a goal. However, building a sound analysis is an important goal: without a soundness guarantee, an analysis might find some bugs, but could not guarantee that no more bugs remain.

A key barrier to sound typestate analyses is the need to reason about aliasing. Consider the classic example [138, 325, 287, 130, 139, 302, 320, 275, 324, 318, 7, 243, 81, 185, 11, 76, 328, 102, 103] of a `File` object, whose typestate is specified in fig. 3.1, and the following program in a Java-like imperative language:

```
File f = new File(...);
f.open();
File g = f;  // f and g are aliases after this line is executed
g.close();
f.read();    // an error occurs when this line is executed
```

On line 3, the shared object—which both aliases `f` and `g` refer to—is in the `open` typestate. When `g.close()` is called on line 4, the state of the underlying object transitions to the `closed` state. It is therefore an error when `f.read()` is called on line 5. However, if a static typestate analysis analyzing this program does not consider that `f` and `g` are aliased, then the analysis's estimate of `f`'s typestate does not transition to the `closed` state, and the analysis unsoundly concludes that the call on line 5 is safe—that is, the analysis suffers from a false negative.

For a sound typestate analysis, there are two high-level approaches to handling aliasing: restrict how the programmer creates aliases (e.g., via ownership types [69, 294] or access

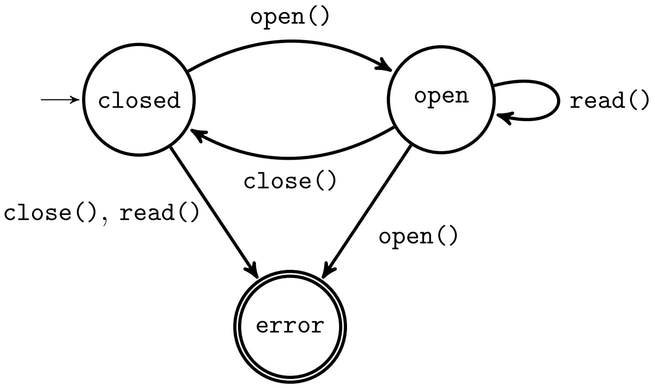


Figure 3.1: The typestate automaton for a `File` object that can be re-opened after being closed. This typestate specification is not an accumulation typestate system: soundly enforcing it statically requires an alias analysis.

permissions [35]), or use a sound inter-procedural may-alias analysis that conservatively overestimates which program variables might be aliases. In practical imperative programming languages with unrestricted aliasing, inter-procedural may-alias analysis is NP-hard [191], and scaling alias analysis to real programs while maintaining acceptable precision remains an open research problem. State-of-the-art analyses often run for an hour or more on practical programs [289].

In this chapter, we describe *accumulation analysis*, which can soundly and modularly solve some (but not all!) typestate problems. An accumulation analysis collects operations—corresponding to typestate transitions—that have definitely occurred on a given program expression. For example, an accumulation analysis could check the property "before calling `read()` on a `File`, call `open()`." The accumulation analysis would record on which expressions `open()` had definitely been called, and forbid calls to `read()` that did not occur via such expressions. Note that this is a weaker property than the full specification in fig. 3.1—it does not forbid "read after close" defects.

Unlike a traditional typestate analysis, an accumulation analysis is sound without any aliasing information. This means that checking a specification with an accumulation analysis is cheaper—often by an order of magnitude or more—than checking that same specification with a general-purpose typestate analysis. Further, effective incremental analysis—i.e., modularity—is possible for an accumulation analysis, because no whole-program alias analysis is needed. Practical accumulation analyses do use limited, cheap, local aliasing information to improve precision. A practical accumulation analysis using limited aliasing information is sound because no aliasing information at all is required for soundness.

We have proven that accumulation analysis does not require aliasing information, demarcated exactly those typestate specifications that can be soundly checked via an accumulation analysis, and explored how common such specifications are. Further, we have implemented multiple practical accumulation analyses, including analyses for common problems like resource leaks and initialization. Our hope is that analysis designers facing typestate-like problems in the future can use our work to determine whether the property they are interested in is an accumulation property: if it is, an accumulation analyses would permit them to verify the property without resorting to an expensive, whole-program alias analysis.

This chapter makes the following contributions:

- a theory of accumulation analysis and its relationship to traditional typestate analysis, including a proof of soundness (section 3.2);
- a literature survey of work on typestate analysis, from which we collected 1,355 typestate specifications and determined that 41% of them are accumulation typestate specifications (section 3.3);
- a description of a general, practical accumulation analysis framework (section 3.4);
- the design of an accumulation analysis that is an effective tool for preventing malformed object construction, and accompanying experimental evidence (section 3.5); and
- the design of a collection of analyses, including an accumulation analysis, that is an effective tool for preventing resource leaks, and accompanying experimental evidence (section 3.6).

The work presented in this chapter includes material from three related publications [177, 180, 181], which I have here attempted to synthesize into a coherent whole. Much of the good in this chapter is due to the work of my collaborators Manu Sridharan, Michael D. Ernst, Narges Shadab, Manli Ran, and Martin Schäf on those papers; all errors are mine.

3.2 A Theory of Accumulation Analysis

3.2.1 Background: What Is Typestate?

In a standard type system, the type of an expression is immutable throughout the program and the set of operations available on the expression is correspondingly immutable. However, type systems fail to capture the behavioral specifications of many real-world objects that change over time. For example, a chess pawn might become a queen and gain new movement operations, a caterpillar might become a chrysalis and lose the ability to crawl before eventually becoming a butterfly and gaining the ability to fly, or a `File` might be opened and gain the ability to be read. In each of these examples, the logical identity of the object stays the same, but its state—and what that state enables it to do—changes. Typestate [284] extends types to account for possible state changes by encoding the various states and behaviors of a type as a finite-state machine—the typestate automaton for that type. Formally:

Definition 1. *A* ***typestate automaton*** $A = (\Sigma, S, s_0, \delta, e)$ *for type* τ *is a finite-state machine. The language* Σ *is the set of operations, such as method calls, that can be performed on* τ*. The states* S *are called* typestates*;* $s_0 \in S$ *is the initial state. The edges defined by the transition table* δ *are called transitions and correspond to the effect of operations. There is a distinguished error state* $e \in S$*. Each typestate has* $k = |\Sigma|$ *outgoing transitions; none, some, or all of these transitions may be to the error state* e *or may be self-loops. The error state* e *has only self-loops—that is, the error state is a trap state.*

At every step during the execution of a program, each value/object of type τ is in one of the typestates of the typestate system.

Definition 2. *An* ***operation*** *is an event that may cause an object to change state. Every type has a set of operations that can be performed on it, but not all operations are necessarily legal in all states. Traditionally, operations are method calls. However, they can be generalized to include any other event, such as assigning a field or a reference going out of scope.*

Without loss of generality, we represent typestate automata as having no accepting states (or, equivalently, all non-error states are accepting). If a typestate automaton were to have one or more accepting states, we could transform it to have no accepting states but encode the same behavioral specification in the following way: add a "go out of scope" transition to each typestate; in accepting states (and the error state), this is a self-loop transition, but in non-accepting states, this is a transition to the error state.

Definition 3. *A* ***typestate system*** *is the pair of a typestate automaton and the corresponding type* τ *whose safe usage it encodes.*

As an example of a typestate system, fig. 3.1 shows the automaton, and the type is `File`. Note how each edge is labeled with the corresponding operation. A double circle around the state represents the distinguished error state e. We always draw all transitions, with the exception of those from the error state (which are, by definition, always self-loops).

We consider only static typestate analyses. Dynamic run-time monitoring to detect typestate violations exists, but a run-time monitor—like any dynamic analysis—cannot prevent errors before they happen. See section 3.8 for more details on related techniques that are outside the scope of the present work.

3.2.2 Definitions and Proofs

This section has three goals. First, section 3.2.2.1 formally defines accumulation analysis. Second, section 3.2.2.2 defines an *accumulation typestate system* and shows that every accumulation analysis has a corresponding accumulation typestate system. Finally, section 3.2.2.3 proves that accumulation typestate systems are exactly those typestate systems that can be soundly checked by a static typestate analysis that does not use aliasing information.

3.2.2.1 Accumulation Analysis

First, we formalize the notion of an accumulation analysis:

Definition 4. *An* ***accumulation analysis*** *is a static program analysis that approximates, for each in-scope expression x of type τ at each program point, a set of operations S that have definitely occurred on the value to which x refers.*

An accumulation analysis has one or more ***goals****. A goal is a pair $\langle g, E \rangle$ where g is the* ***goal operation*** *and E is a set of* ***enabling operations****.*

Informally, an accumulation analysis enforces that a goal operation g does not occur until after every enabling operation $e \in E$ for g has already occurred.

An operation in an accumulation analysis is defined identically to an operation in a typestate automaton (definition 2).

Definition 5. *A* ***sound accumulation analysis*** *must issue an error if some goal operation may occur before its enabling operations. More formally, it must issue an error if, for some expression x of type τ and some operation g, both of the following are true:*

1. *There exists at least one goal $\langle g, \_ \rangle$—that is, g is a goal operation.*
2. *There exists an execution of the program where the set of operations S that have actually occurred on the value of x before an occurrence of g on x is not a superset of one of the enabling sets for g. That is, where there does not exist some goal $\langle g, E \rangle$ such that $S \supseteq E$.*

Intuitively, a sound accumulation analysis is "accumulating" enabling operations, and once everything in the enabling set is accumulated, there is no way to "disable" the goal operation. For example, if g is a goal operation for some goal $\langle g, E \rangle$, an object must first perform some set of operations to make g legal (i.e., the operations in E), and once g becomes legal, it *stays* legal.

Note that soundness, as in definition 5, only precludes false negative warnings. It says nothing about whether the accumulation analysis might issue a false positive, and a trivially-sound "accumulation analysis" could simply issue an error any time a goal operation might be executed. In practice, a useful accumulation analysis tracks whether the transitions in an enabling set have occurred, and it permits the goal operation if they have.

Note that if an accumulation analysis has multiple goals, their goal operations may or may not be the same. Multiple goals with the same goal operation are useful to express disjunctive specifications. For example, section 3.5 uses the disjunctive specification "call either `withOwners()` or `withImageIds()` before calling `describeImages()`."

3.2.2.2 Relationship Between Typestate and Accumulation

Next, we need to describe the relationship between a typestate system and an accumulation analysis. As an aid to doing so, we introduce the following:

Definition 6. *An* ***error-inducing sequence*** *in a typestate automaton T is a sequence of transitions $S = t_1, \ldots, t_i$ such that T is in the error state after all transitions in S are applied (and not before).*

Definition 7. *An* ***accumulation typestate system*** *is a typestate system such that for any error-inducing sequence $S = t_1, \ldots, t_i$, all subsequences (including both contiguous and non-contiguous subsequences) of S that end in t_i also result in the typestate automaton being in the error typestate (i.e., all subsequences of S that end in t_i are also error-inducing).*

Intuitively, an accumulation typestate system is any typestate system whose error-inducing paths are closed under subsequence so long as the final error-inducing operation is held constant. That is, removing operations from the beginning or middle of an error-inducing sequence always produces another error-inducing sequence.

Note that a vacuous sound typestate analysis such as "issue an error at every program statement" is trivially enforcing an accumulation typestate system. The typestate automaton that such an analysis enforces only has transitions to the error state, so all sequences are error-inducing.

This definition leads to a decision procedure (algorithm 1) for determining whether a given typestate system T is an accumulation typestate system. Consider all error-inducing operations $U = \{u_1, \ldots, u_n\}$. The elements of U are the final transitions for every error-inducing sequence in the automaton of T. For any $u_i \in U$, let E_i be the language[1] of the error-inducing sequences of operations in T that end in u_i, with the last transition removed (i.e., the u_i transition that leads to the `error` typestate). Let $E_{subseq(i)}$ be the language of subsequences of E_i. Let $E = \bigcup_{i=1}^{n} E_i * u_i$ and $E_{subseq} = \bigcup_{i=1}^{n} E_{subseq(i)} * u_i$. That is, E is the union of all error-inducing paths in T, and E_{subseq} is the union of all subsequences of error-inducing paths in T that end in the same transition as the corresponding error-inducing path from which they were derived. If and only if E and E_{subseq} recognize the same language, T is an accumulation typestate system.

It is easy to check whether E and E_{subseq} recognize the same language, because both are regular. E is regular, because it can be recognized by T's automaton, if the error typestate is converted to an accepting state. Since there are finitely-many operations, any E_i and $E_{subseq(i)}$ have a finite alphabet. Higman's theorem [157] says that the language of the subsequences of any language over a finite-alphabet is regular. Therefore, any $E_{subseq(i)}$ is also regular. E_{subseq} is regular because regular languages are closed under both union and concatenation. So, the procedure for checking whether a typestate automaton is an accumulation typestate

[1] Throughout, we will abuse notation and refer to both languages and their corresponding language-recognizers by the same name.

Algorithm 1: A decision procedure for checking whether or not a given typestate automaton T is an accumulation typestate automaton. The complexity of the algorithm is the larger of $O(nlogn)$ where n is the number of states, or $O(en)$, where n is the number of states and e is the number of edges.

```
input  : A typestate automaton $T$
output : True iff $T$ is an accumulation typestate automaton
/* FindErrorInducingTransitions returns all transitions to error.            */
$U \leftarrow$ FindErrorInducingTransitions($T$)
/* $E$ and $E_{subseq}$ are finite-state automata. $\forall$ $X$, Union($\emptyset, X) = X$.   */
$E \leftarrow \emptyset$
$E_{subseq} \leftarrow \emptyset$
foreach $u_i \in U$ do
    /* ErrorInducingAutomatonVia is a two-step process: (1) modify T so that
       states from which $u_i$ is error-inducing are accepting, and then (2) minimize and
       return the result, which is an automaton that accepts a sequence of transitions
       $S$ iff $S$ followed by $u_i$ causes an error in the original automaton $T$.      */
    /* Concat modifies an input automaton to accept iff it receives a sequence that
       the input automaton accepts followed by the concatenated transition.      */
    $E_i \leftarrow$ Concat(ErrorInducingAutomatonVia($u_i, T$), $u_i$)
    /* Subsequences produces the automaton that accepts the subsequence language
       for the input automaton, which Higman's theorem guarantees exists.        */
    $E_{subseq(i)} \leftarrow$ Concat(Subsequences($E_i$), $u_i$)
    $E \leftarrow$ Union($E, E_i$)
    $E_{subseq} \leftarrow$ Union($E_{subseq}, E_{subseq(i)}$)
return AcceptSameLanguage($E, E_{subseq}$)
```

automaton is as easy as checking whether the two finite state machines for E and E_{subseq} recognize the same language.

Theorem 1. *Every accumulation analysis has a corresponding accumulation typestate system.*

Proof. Consider some accumulation analysis *acc* with goals $(g_1, E_1), \ldots, (g_n, E_n)$ over type τ. The corresponding accumulation typestate system is the pair of the type τ and the accumulation typestate automaton constructed by the following procedure:

1. Create an error state `error` with a self-loop transition for each operation on τ.

2. Let $\mathcal{P}_E$ be the powerset of E, where $E = \bigcup_{i=1}^{n} E_i$ is the union of the enabling sets $E_1, \ldots, E_n$. For each element S of $\mathcal{P}_E$, create a corresponding state and label it with S. Note that S refers to both the member of $\mathcal{P}_E$ and the corresponding state.

3. Make the state that is labeled by the empty set be the start state of the automaton.

4. For each state $S \in \mathcal{P}_E$ and for each transition $t_e \in E$, add a transition from state S to state $S \cup \{t_e\}$ labeled t_e. (This transition might be a self-loop.)

5. Let $G = \{g_1, \ldots, g_n\}$ be the set of goal transitions. For each element g_i of G and for each state $S \in \mathcal{P}_E$:
   If there exists a goal $\langle g_i, E_i \rangle$ such that $E_i \subseteq S$,
   then add a self-loop transition to S labeled g_i if it does not already have a transition labeled g_i. (It might have such a transition if g_i is both an enabling transition and a goal transition.)
   Else if such a goal does not exist,
   add a transition from S to the `error` state labeled g_i, removing a transition labeled g_i if one already exists.

6. For each operation t on τ such that $t \notin G$ and $t \notin E$—that is, for each operation that is neither a goal operation nor an enabling operation—add a self-loop transition labeled t to each non-error state. (Recall that the error state already has self-loop transitions for each operation, added in step 1.)

The resulting accumulation typestate automaton encodes the same behavior as the original accumulation analysis. □

Note that this construction is a existence proof, not an efficient translation: it does induce an exponential blowup in the number of states. A practical accumulation analysis does not track states directly—rather, it tracks only the enabling sets—so state explosion is not a problem in practice.

3.2.2.3 Soundness Without Aliasing

This section proves that accumulation typestate systems are exactly the typestate systems that are soundly checkable without reasoning about aliasing:

Theorem 2. *A typestate system $T = (A, \tau)$ is an accumulation typestate system if and only if there exists a typestate analysis that can soundly check T with no aliasing information.*

The high-level intuition behind the proof of theorem 2 is the consequence of two facts:

- without using aliasing information, a typestate analysis observes only a subsequence of the actual operations that are applied to the object to which some expression refers, and

- accumulation typestate automata are exactly those that are error-closed under subsequence, when the last transition is held constant.

The formal proof is split into lemmas 2 and 3 (which are the forward and backward directions of the bi-implication respectively). Before we proceed to the proof of lemmas 2 and 3, we define the supporting machinery of the proof: the language, relevant definitions, etc.

Accumulation analyses as defined in section 3.2.2.1 are sound without access to aliasing information:

Corollary 1. *An accumulation analysis, even without aliasing information, is sound.*

Proof. Convert the accumulation analysis to an accumulation typestate system via the procedure in the proof of theorem 1. By theorem 2, the accumulation typestate system can be soundly checked. □

An important consequence of the ability to soundly check an accumulation typestate system with *no* aliasing information is that approaches that utilize *limited* aliasing information are also sound. In practice, analyses can compute inexpensive, typically local, alias information to improve precision (i.e., to avoid issuing false positive warnings); see section 3.4.

Preliminaries This section introduces the machinery used to prove theorem 2.

Language We will prove theorem 2 over a core calculus that represents a simple imperative programming language. This language contains the essential parts of a programming language related to typestate checking and aliasing—method calls, fields, and assignments.

A program P in this language is a statement s of one of the following kinds:

- an assignment: $x_i \texttt{ := } x_j$.
- a field load: $x_i \texttt{ := } x_j\texttt{.}f_k$.
- a field store: $x_i\texttt{.}f_j \texttt{ := } x_k$.
- a method call: $x_i\texttt{.}m_j\texttt{()}$.
- a statement sequence: $s_i \texttt{ ; } s_j$.

Source code variables range from $\texttt{x\_}_1$ to $\texttt{x\_}_n$, where n is some positive integer. Statements may only refer to variables in that range. There is a single type T. Each variable contains a reference to a *value*—that is, a particular object instance—of type T. We use $x_i, x_j, \ldots$ as metavariables for arbitrary variables in the range $\texttt{x\_}_1,\ldots,\texttt{x\_}_n$. T has methods $\texttt{m\_}_1$ to $\texttt{m\_}_k$ and a corresponding typestate automaton A whose k operations are exactly the methods $\texttt{m\_}_1$ to $\texttt{m\_}_k$. A method call statement can only refer to methods in T. We use $m_i, m_j, \ldots$ as metavariables for arbitrary methods in T. Each object of type T has fields $\texttt{f\_}_1$ to $\texttt{f\_}_m$, where m is some positive integer. Load and store statements may only refer to fields in this range. Each field contains a reference to some value of type T. We use $f_i, f_j, \ldots$ as metavariables for arbitrary fields in T.

To simplify the presentation and proofs, this language lacks conditionals, loops, method bodies, return values, etc.—which makes precise alias and typestate analysis trivial. However, our algorithms are general (they do not take advantage of the straight-line nature of the code) and can be extended to a richer language without changing the essence of the proof. Section 3.4 discusses practical concerns when implementing an accumulation analysis for a real programming language.

Dynamic Semantics To execute a program, we maintain a *machine state* $\langle \rho, \sigma, \tau \rangle$ composed of an environment (ρ) mapping each variable to a value of type T, a store (σ) mapping each value–field pair to a value, and a typestate store (τ) mapping each value to a typestate in A. The initial environment maps each x_i to a distinct value v_j. The initial store maps each value–field pair $\langle v_i, f_j \rangle$ to a distinct value v_k. The initial typestate store maps each value v_i to the start typestate s_0 of A.[2] Executing a statement in machine state $\langle \rho, \sigma, \tau \rangle$ either produces an updated machine state $\langle \rho', \sigma', \tau' \rangle$, or it terminates the program in an error if any value's entry in the typestate store would be A's `error` typestate. The dynamic semantics (fig. 3.2) are as follows:

- For an assignment x_i `:=` x_j, produce a new machine state with an updated environment: $\rho'(x_i) = \rho(x_j)$ (rule ASSIGN).

- For a field load x_i `:=` x_j`.`f_k, produce a new machine state with an updated environment: $\rho'(x_i) = \sigma(\rho(x_j), f_k)$ (rule LOAD).

- For a field store x_i`.`f_j `:=` x_k, produce a new machine state with an updated store: $\sigma'(\rho(x_i), f_j) = \rho(x_k)$ (rule STORE).

- For a call x_i`.`m_j`()`, let $t' = \mathit{succ}(\tau(\rho(x_i)), m_j, A)$. That is, t' is the successor typestate in A when transition m_j occurs in the current typestate of the value that x_i is a reference to. If t' is not the `error` typestate, produce a new machine state with an updated typestate store: $\tau'(\rho'(x_i)) = t'$ (rule CALL). If t' is the `error` typestate, the semantics "get stuck" and the program terminates in an error.

- For a sequence s_i `;` s_j, first execute s_i. If the program terminates in an error while executing s_i, the semantics for the sequence statement "get stuck". Otherwise, let $\langle \rho', \sigma', \tau' \rangle$ be the machine state after executing s_i. Execute s_j in $\langle \rho', \sigma', \tau' \rangle$ (rule SEQ).

[2] Initializing all variables before a program starts simplifies the language by removing the need for a `new` expression.

$$\frac{}{\langle \rho, \sigma, \tau \rangle \vdash x_i := x_j \Downarrow \langle \rho[x_i \mapsto \rho(x_j)], \sigma, \tau \rangle}\ \text{ASSIGN}$$

$$\frac{}{\langle \rho, \sigma, \tau \rangle \vdash x_i := x_j.f_k \Downarrow \langle \rho[x_i \mapsto \sigma(\langle \rho(x_j), f_k \rangle)], \sigma, \tau \rangle}\ \text{LOAD}$$

$$\frac{}{\langle \rho, \sigma, \tau \rangle \vdash x_i.f_j := x_k \Downarrow \langle \rho, \sigma[\langle \rho(x_i), f_j \rangle \mapsto \rho(x_k)], \tau \rangle}\ \text{STORE}$$

$$\frac{\langle \rho, \sigma, \tau \rangle \vdash t' = \mathit{succ}(\tau(\rho(x_i)), m_j, A) \qquad t' \neq \mathit{error}}{\langle \rho, \sigma, \tau \rangle \vdash x_i.m_j() \Downarrow \langle \rho, \sigma, \tau[\rho(x_i) \mapsto t'] \rangle}\ \text{CALL}$$

$$\frac{\langle \rho, \sigma, \tau \rangle \vdash s_i \Downarrow \langle \rho', \sigma', \tau' \rangle \qquad \langle \rho', \sigma', \tau' \rangle \vdash s_j \Downarrow \langle \rho'', \sigma'', \tau'' \rangle}{\langle \rho, \sigma, \tau \rangle \vdash s_i; s_j \Downarrow \langle \rho'', \sigma'', \tau'' \rangle}\ \text{SEQ}$$

Figure 3.2: The big-step dynamic semantics of the language expressed as inference rules. The notation $\mu[x \mapsto y]$ means that the map μ is updated so that x maps to y. $M \vdash s \Downarrow M'$ means that executing statement s in machine-state M results in machine-state M'.

Sound Typestate Analysis

Definition 8. *A **typestate analysis** is a static program analysis. Its inputs are a program P and a typestate system $T = (A, \tau)$. It reports call statements within P that may cause the program to terminate in an error when running P.*

Definition 9. *A typestate analysis is **sound** if it reports each call statement that causes the program to terminate in an error at run time in any execution of the program.*

Representation of Aliasing Suppose that a typestate analysis has access to two oracle functions $\mathit{MustOracle}(x_i, s)$ and $\mathit{MayOracle}(x_i, s)$ for aliasing information. Each oracle takes a variable x_i and a program statement s and returns a list of *names*—variables or arbitrarily-nested field load expressions—that the input variable must (respectively, may) alias before the given statement.

MustOracle returns a list of names that definitely do alias x_i at s. More formally, for a sound oracle, if the list returned by $\mathit{MustOracle}(x_i, s)$ contains x_j, then x_i and x_j are definitely aliased before statement s on all executions. If the list does not contain x_j, then x_i and x_j may or may not be aliased before s. A trivial *MustOracle* that always returns an empty list is sound.

MayOracle returns a list of names that *might or might not* alias x_i at s. More formally, for a sound oracle, if the list returned by $\mathit{MayOracle}(x_i, s)$ does not contain x_j, then x_i and

x_j are definitely not aliased before statement s on all executions. If the list does contain x_j, then x_i and x_j may or may not be aliased before s. A trivial *MayOracle* that always returns every in-scope name in the program is sound.

These oracles can represent an external alias analysis, an on-demand alias analysis, aliasing tracking built into the typestate analysis, etc. If the oracles are sound, then for all x_i and s, $\mathit{MustOracle}(x_i, s) \subseteq \mathit{MayOracle}(x_i, s)$. For a traditional typestate analysis (as defined by fig. 3.3 below) to be sound for an arbitrary typestate system such as the `File` example in fig. 3.1, both oracles must be sound.[3]

Definition of Typestate Analysis A typestate analysis is a fixpoint analysis that can be viewed as a dataflow analysis or an abstract interpretation. It operates by maintaining a set of *abstract stores*, one for each program point. An abstract store is a map from names to sets of estimated typestates. We write $\phi_s(x_i)$ for the estimated typestates of name x_i before program statement s, and $\phi'_s(x_i)$ for those after. For any sequencing statement $r\texttt{;}s$, for all x_i, $\phi'_r(x_i) = \phi_s(x_i)$. The notation $\hat{\phi}_s(x_i.*)$ means all names in ϕ_s that begin with x_i.

At the beginning of the analysis, at every program point, the abstract store maps all names[4] to the set containing only the start state s_0 of the typestate automaton A. Then, the analysis processes each statement s using the following rules (which also appear in fig. 3.3) until the set of abstract stores reaches a fixpoint:

- For an assignment x_i `:=` x_j, for each $n \in \hat{\phi}_s(x_i.*)$, let $n' = n[x_j/x_i]$—that is, n' is n with its x_i replaced by x_j—and let $T'_{n'} = \phi_s(n')$, the abstract value of n' in the pre-state. The analysis updates the abstract store after s so that n is mapped to $T'_{n'}$: $\phi'_s(n) := T'_{n'}$ (rule TS-ASSIGN). For all other names m in ϕ_s where $m \notin \hat{\phi}_s(x_i.*)$, the analysis copies the entry from the previous abstract store: $\phi'_s(m) := \phi_s(m)$.

- For a load statement x_i `:=` x_j`.`f_k, for each $n \in \hat{\phi}_s(x_i.*)$, let $n' = n[x_j.f_k/x_i]$ and let $T'_{n'} = \phi_s(n')$. The analysis updates the abstract store after s so that n is mapped to $T'_{n'}$: $\phi'_s(n) := T'_{n'}$ (rule TS-LOAD). For all other names m in ϕ_s where $m \notin \hat{\phi}_s(x_i.*)$, the analysis copies the entry from the previous abstract store: $\phi'_s(m) := \phi_s(m)$.

- For a store statement x_i`.`f_j `:=` x_k, for each $n \in \hat{\phi}_s(x_i.f_j.*)$, let $n' = n[x_k/x_i.f_j]$ and let $T'_{n'} = \phi_s(n')$. Then, for each n and its n' and $T'_{n'}$, the analysis performs the following steps (rule TS-STORE):

[3] For the language of fig. 3.2, it is trivial to construct a sound alias analysis that never includes a name in the result of a *MayOracle* query unless the corresponding *MustOracle* query would also include that name. In a richer programming language, the *MayOracle* is necessary to handle analysis imprecision and control flow joins.

[4] An analysis may use widening, abstraction, or iterative expansion of maps to handle the fact that the set of names is infinite.

$$\frac{\begin{array}{c}\phi_s \vdash \forall n \in \hat{\phi}_s(x_i.*), n' = n[x_j/x_i] \wedge T'_{n'} = \phi_s(n') \\ \phi'_s = \phi_s[\forall n \in \hat{\phi}_s(x_i.*), n \mapsto T'_{n'}]\end{array}}{\phi_s \vdash x_i := x_j \Downarrow \phi'_s}\ \text{TS-ASSIGN}$$

$$\frac{\begin{array}{c}\phi_s \vdash \forall n \in \hat{\phi}_s(x_i.*), n' = n[x_j.f_k/x_i] \wedge T'_{n'} = \phi_s(n') \\ \phi'_s = \phi_s[\forall n \in \hat{\phi}_s(x_i.*), n \mapsto T'_{n'}]\end{array}}{\phi_s \vdash x_i := x_j.f_k \Downarrow \phi'_s}\ \text{TS-LOAD}$$

$$\frac{\begin{array}{c}\phi_s \vdash \forall n \in \hat{\phi}_s(x_i.f_j.*), n' = n[x_k/x_i.f_j] \wedge T'_{n'} = \phi_s(n') \wedge \\ A_n^{must} = \mathit{MustOracle}(n, s) \wedge A_n^{may} = \mathit{MayOracle}(n, s) \\ \phi'_s = \phi_s[\forall n \in \hat{\phi}_s(x_i.f_j.*), n \mapsto T'_{n'}][\forall a_n \in A_n^{must}, a_n \mapsto T'_{n'}] \\ [\forall b_n \in A_n^{may} - A_n^{must}, b_n \mapsto T'_{n'} \cup \phi_s(b_n)]\end{array}}{\phi_s \vdash x_i.f_j := x_k \Downarrow \phi'_s}\ \text{TS-STORE}$$

$$\frac{\begin{array}{c}\phi_s \vdash T = \phi_s(x_i) \qquad T' = \bigcup_{t \in T} succ(t, m_j, A) \\ A^{must} = \mathit{MustOracle}(x_i, s) \qquad A^{may} = \mathit{MayOracle}(x_i, s) \\ \phi'_s = \phi_s[x_i \mapsto T'][\forall a \in A^{must}, a \mapsto T'][\forall b \in A^{may} - A^{must}, b \mapsto T' \cup \phi_s(b)]\end{array}}{\phi_s \vdash x_i.m_j() \Downarrow \phi'_s}\ \text{TS-CALL}$$

$$\frac{\phi_s \vdash s_i \Downarrow \phi'_{s_i} \qquad \phi'_{s_i} = \phi_{s_j} \qquad \phi_{s_j} \vdash s_j \Downarrow \phi'_s}{\phi_s \vdash s_i; s_j \Downarrow \phi'_s}\ \text{TS-SEQ}$$

Figure 3.3: Inference rules for a traditional, sound typestate analysis. Each rule applies to some statement s, which appears in the consequent. The notation $x[y/z]$ means "x with each z replaced by y." The notation $\hat{\phi}_s(x_i.*)$ means all names in ϕ_s that begin with x_i.

1. The analysis updates the abstract store after s so that n is mapped to $T'_{n'}$: $\phi'_s(n) := T'_{n'}$.
2. The analysis queries $\mathit{MustOracle}(n, s)$ (call the result A_n^{must}). For each $a_n \in A_n^{must}$, the analysis performs a *strong update* to the abstract store: $\phi'_s(a_n) := T'_{n'}$.
3. The analysis queries $\mathit{MayOracle}(n, s)$ (call the result A_n^{may}). For each element b_n in $A_n^{may} - A_n^{must}$—that is, variables that may be aliases but are not guaranteed to be aliases—the analysis performs a *weak update* to the abstract store so that it maps b_n to $T'_{n'} \cup \phi_s(b_n)$: $\forall b_n \in A_n^{may} - A_n^{must}, \phi'_s(b_n) := T'_{n'} \cup \phi_s(b_n)$.

For all other names m in ϕ_s where $m \notin \hat{\phi}_s(x_i.f_j.*) \wedge \forall A_n^{may}, m \notin A_n^{may}$, the analysis copies the entry from the previous abstract store: $\phi'_s(m) := \phi_s(m)$.

- For a call statement x_i.m_j(), let $T' = \bigcup_{t \in \phi_s(x_i)}$. The analysis performs the following steps (rule TS-CALL):

  1. If any $t' \in T'$ is `error`, the analysis reports an error for the statement.
  2. The analysis updates the abstract store so that $\phi'_s(x_i) := T'$.
  3. The analysis queries $MustOracle(x_i, s)$ (call the result A^{must}). For each $a \in A^{must}$, the analysis performs a strong update to the abstract store: $\phi'_s(a) := T'$.
  4. The analysis queries $MayOracle(x_i, s)$ (call the result A^{may}). For each $b \in A^{may} - A^{must}$, the analysis performs a weak update to the abstract store: $\phi'_s(b) := T' \cup \phi_s(b)$.

- For a sequence $s = s_i$; s_j, the analysis first analyzes s_i, and then analyzes s_j with the resulting abstract store (rule TS-SEQ)). (Note that the analysis does not terminate in the case of an error, but keeps reporting errors on subsequent statements.)

This standard formulation of a traditional typestate analysis is sound for any arbitrary typestate system, as long as its aliasing oracles are sound:

Theorem 3. *A traditional typestate analysis is sound if its MustOracle and MayOracle functions return sound results.*

Proof. By co-induction on the dynamic semantics (fig. 3.2) and the rules for a traditional typestate analysis (fig. 3.3). The key invariant is that the actual typestate to which a name refers on any particular execution at some statement is always in the abstract store. □

Typestate Analysis with No Aliasing Information

Definition 10. *A* ***typestate analysis with no alias information*** *is a typestate analysis whose MustOracle and MayOracle functions return empty lists for all arguments.*

Intuitively, a typestate analysis "with no alias information" assumes that no aliasing occurs in the program—even when making such an assumption is unsound.

A typestate analysis with no alias information has a simpler method call rule: it never updates its abstract store in response to an aliasing query, so steps 3 and 4 may be omitted. Similarly, there is a simpler store rule: only the $n \in \hat{\phi}_s(x_i.f_j.*)$ need to be updated, because all *MayOracle* queries (unsoundly) return false.

Informally, having no aliasing information means that the analysis might not be aware that one or more transitions have occurred on the value to which some expression refers, because those operations occurred via an alias. That is, the analysis's estimate of the typestate of an expression that actually refers (at run time) to a value v in typestate t is a typestate reachable by a subsequence of the sequence of transitions that results in $\tau(v)$ being t. Stated more formally:

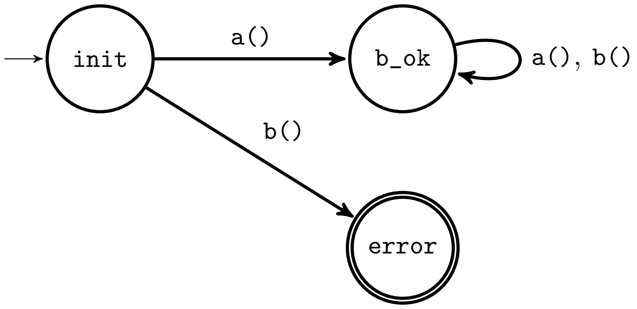


Figure 3.4: An accumulation typestate automaton for the property "call a() before calling b()".

Lemma 1. *Let $R = \phi_s(x_i)$ be the set of estimated typestates produced by a typestate analysis with no aliasing information for a variable x_i before a statement s. Let S be the trace of an arbitrary execution leading up to some occurrence of s, and let $t = \tau(\rho(x_i))$ be the typestate of the actual value to which x_i refers before that occurrence of s. Applying S to the automaton leads to typestate t. There exists a typestate $r \in R$ such that applying some subsequence of S leads to r. That is, there is some estimated typestate r that is reachable by a subsequence of the transitions that lead to t.*

Stated another way, lemma 1 says that for every possible trace S through the program that reaches s, there is at least one $r \in R$ that "corresponds to" S, in the sense that r is reachable by a subsequence of S.

Lemma 1 is not quite true of a typestate analysis as defined in fig. 3.3: field loads do not necessarily preserve it. Because the store rule is unsound due to the unsoundness of the aliasing oracles, the entry in the abstract store for a given field may not actually be related to the value to which that name refers, due to possible aliasing. For example, consider the following program, being analyzed with respect to the "only call b() after a()" typestate automaton in fig. 3.4 (note that "Estimated state" and "Actual state" columns only show entries for names that are relevant to the problem):

Program	Estimated state (ϕ_s)[5]	Actual state (τ)[6]
`x2 = x1`	{x1.f↦init, x2.f↦init}	{x1.f↦init, x2.f↦init}
`x3.a()`	{x1.f↦init, x2.f↦init, x3↦b_ok}	{x1.f↦init, x2.f↦init, x3↦b_ok}
`x1.f = x3`	{x1.f↦b_ok, x2.f↦init, x3↦b_ok}	{x1.f↦b_ok, x2.f↦init, x3↦b_ok}
`x2.f = x4`	{x1.f↦b_ok, x2.f↦init}	{x1.f↦init, x2.f↦init}
`x5 = x1.f`	{x1.f↦b_ok, x2.f↦init, x5↦b_ok}	{x1.f↦init, x2.f↦init, x5↦init }
`x5.b()`	{x1.f↦b_ok, x2.f↦init, x5↦b_ok}	{x1.f↦init, x2.f↦init, x5↦init }

$$\frac{\phi_s \vdash \phi'_s = \phi_s[x_i \mapsto s_0]}{\phi_s \vdash x_i := x_j.f_k \Downarrow \phi'_s} \text{ TS-LOAD-FIX}$$

Figure 3.5: A modified load rule for a typestate analysis with no aliasing information, which preserves lemma 1. s_0 is the start state of the automaton A being checked.

This program (left side of the table above) leads to lemma 1 being untrue at the final statement, because the actual state of `x5` (`init`) is not reachable from the estimated state (`b_ok`). The key issue is aliasing: `x1` and `x2` are aliases, so `x1.f` and `x2.f` actually refer to the same value. When `x2.f` is re-assigned to `x4`, the actual value to which `x1.f` refers changes—but with no aliasing information, the typestate analysis is unaware, leading to the problem.

Note that this problem applies to arbitrary typestate systems: both accumulation typestate systems and non-accumulation typestate systems. Lemma 1 discusses both.

There is a simple solution to this problem that makes lemma 1 hold for a typestate analysis with no aliasing information: update the load rule so that the analysis assumes that all loads return a value whose typestate is the start state of the automaton (rule TS-LOAD-FIX in fig. 3.5).

This rule trivially preserves lemma 1 for field loads, and corresponds with how accumulation analyses handle field loads in practice (see section 3.4). Our proof assumes this simpler load rule for the typestate analysis with no aliasing information. However, note that this rule would make a traditional typestate analysis unsound (i.e., this rule makes theorem 3 untrue): in an arbitrary typestate analysis, the start state is not necessarily a safe default assumption. A useful property of accumulation typestate automata, however, is that every operation which might ever lead to an error on any path must necessarily lead to an error from the start state—otherwise, the definition of accumulation typestate automaton could not be met when considering the empty subsequence.

We now prove lemma 1:

Proof. By co-induction on the dynamic semantics and the rules for a typestate analysis with no aliasing information. Here is the case analysis:

Base case: when a program begins executing, the dynamic semantics say that all names refer to values in the start state. A typestate analysis with no aliasing information estimates that at a program's entry point, all names are in the start state, as well. Trivially, the start state is reachable by the same sequence of operations as itself.

Case assignment: For an assignment s, where s is x_i `:=` x_j, the invariant is preserved

[5] Entries in ϕ_s are single-element sets. For simplicity of presentation, set notation has been elided.

[6] Keys in τ are values. For simplicity of presentation, the necessary lookups in ρ and σ have been elided.

by the inductive hypothesis. Consider that by the inductive hypothesis, the invariant is preserved for x_j. Then consider the rule used by the typestate analysis with no aliasing information for an assignment: every mention of x_i in the abstract store is replaced by x_j. Further, the dynamic semantics for an assignment require that the previous value of x_i is no longer accessible via x_i: x_i after the assignment refers only to x_j. Since x_i and x_j after the assignment are treated entirely the same, but the abstract store is otherwise unchanged by the analysis, what was true of x_j before the statement is true for x_i after.

Case load: The special load rule TS-LOAD-FIX trivially guarantees that the invariant is preserved: the start state is reachable by a subsequence of the operations that reach any other state (in particular, by the empty subsequence).

Case store: This rule trivially preserves the invariant, because the invariant must be maintained only for the estimates for variables—not for fields—and rule TS-STORE only updates estimates for fields.

Case method call: For a method call $s = x_i\texttt{.}m_j\texttt{()}$, only steps 1 and 2 of rule TS-CALL are applied, because a typestate analysis with no aliasing information never performs strong or weak updates on possible aliases. The invariant is preserved via the inductive hypothesis: for x_i itself, let r_1 be the element of R that is reachable by a subsequence of the actual sequence S in the inductive hypothesis. The analysis updates its estimate to include $r_1 + m_j$ (that is, the sequence r_1 followed by the transition m_j). After s is executed, the actual sequence is $S + m_j$, and since we know that r_1 is reachable by a subsequence of S, $r_1 + m_j$ must be reachable by a subsequence of $S + m_j$—the same subsequence used to reach r_1, with m_j added on. For any aliases of x_i, the inductive hypothesis also guarantees that the invariant holds: the estimate contains some r that is a subsequence of S, and any subsequence of S is also a subsequence of $S + m_j$.

Case sequence: For a sequence, the invariant is trivially preserved by induction. □

Proof of Theorem 2 The proof is split into two parts—the forwards and backwards direction of the bi-implication, which are lemmas 2 and 3, respectively.

Lemma 2. *T is an accumulation typestate system $\implies$ there exists a sound typestate analysis that can check T with no aliasing information.*

Proof. The proof is by contradiction. Suppose that an arbitrary typestate analysis with no aliasing information (as defined by definition 10) for an accumulation typestate system T is unsound. That is, suppose that it fails to issue an error at some method call statement $s = x_i\texttt{.}m_j\texttt{()}$, but the program terminates in an error in some execution e, because $\tau(\rho(x_i))$ after s would be `error`.

Let $v_i = \rho(x_i)$. That is, x_i actually refers to v_i at[7] s on execution e. m_j must be the transition that would lead v_i to enter the error typestate at the call $x_i\texttt{.}m_j\texttt{()}$, because the program would have already terminated if some other transition might have caused v_i to

[7] s must be a method call statement, so v_i is the same before and after s.

enter the error state before s was reached. Let $R' = \phi'_s(x_i)$ be the analysis's estimate of the possible typestates of x_i after the call statement is executed. Because the analysis did not issue an error at s, R' must not contain the `error` typestate.

Since R' does not contain the error typestate after observing m_j, then m_j must have been a legal transition on each typestate in the analysis' pre-state estimate $R = \phi_s(x_i)$. By lemma 1, there is some typestate $r \in R$ that is reachable via some subsequence of the transitions that led to the actual typestate $t = \tau(\rho(x_i))$ that v_i was in during e before transition m_j was applied.

The typestate r is reachable by a subsequence of the sequence of transitions that actually occurred on v_i that led it to reach t, but m_j is a legal transition in r. This is a contradiction: m_j must be both an error-inducing and a legal transition in r. m_j must be an error-inducing transition in r by the definition of an accumulation typestate system (definition 7): m_j must be an error-inducing transition in typestates reachable via subsequences of the transitions that lead to t, including r. But, m_j must also be a legal transition in r because the analysis did not issue an error when its estimate included r. Since one transition cannot be both error-inducing and legal, by contradiction, the analysis must have been sound. □

Lemma 3. *T is an accumulation typestate system $\Longleftarrow$ there exists a sound typestate analysis that can check T with no aliasing information.*

Proof. The proof is by contradiction. Suppose that there is a typestate analysis with no aliasing information that can soundly check a typestate system T that is not an accumulation typestate system. Since T is not an accumulation typestate system, there exists some sequence of transitions $S = t_1, \ldots, t_i$ that ends in an error typestate that has a subsequence S' that ends in t_i that does not end in an error typestate. Let D be the difference between S' and S: the sequence of transitions that appear in S but do not appear in S'.

Construct a program P with two variables $x_{S'}$ and x_D. The first statement in P is x_D `:=` $x_{S'}$, which aliases these expressions. Then augment the program in the following manner: for each transition $t \in S$, if t is an element of S', then add the statement $x_{S'}$`.`t`()` to P. Otherwise, add the statement x_D`.`t`()` to P.

Because $x_{S'}$ and x_D were aliased by P's first statement, we know that they both point to a single value v to which every transition in S has been applied by the end of P; thus, P terminates in an error when the final transition t_i is applied. However, no error is issued: the analysis will not issue an error for $x_{S'}$`.`t_i`()`, which is the program statement that causes the error, because the sequence R that was applied to $x_{S'}$ is a legal sequence of transitions (and the error-inducing transition t_i is guaranteed to be in S', not in D, by definition). This is a contradiction of our original premise that a typestate analysis with no aliasing information could soundly check T: an error-inducing transition (t_i) occurs, but the analysis with no aliasing information fails to issue an error. Thus, T must have been an accumulation typestate system. □

3.3 How Common is Accumulation? A Literature Survey

This section aims to answer the research question: **RQ1: What fraction of typestate problems can be solved modularly with an accumulation analysis?**

We will approximate the answer by using the population of typestate problems that appear in the scientific literature. Note that this is likely to be an under-approximation of incidence in practice, because scientific papers usually address the most complex problems.

We performed a literature survey of papers in the research literature since 2000 that contain typestate specifications. We chose the year 2000 because a similar survey [101], which we discuss in section 3.3.2.2, was published in 1999. For each typestate specification that we discovered, we used the decision procedure in algorithm 1 to determine whether the specification was an accumulation typestate system—and therefore soundly analyzable without any aliasing information by theorem 2. The vast majority of the papers that we analyzed use typestate for some small number of examples. We report on these papers in aggregate and describe specific, common examples (section 3.3.2.1). There are two outliers [101, 28] that reported on categories containing hundreds of specifications, which we discuss in detail (section 3.3.2.2).

The remainder of this section details our methodology, discusses the results, and gives examples of specifications that can and cannot be checked via accumulation.

3.3.1 Methodology

We searched Google Scholar for papers since 2000 whose full-text includes "typestate", resulting in 1,760 hits. (We originally included "type-state" and "type state" as search terms, but discovered no computer science results in the first 100 hits for each that "typestate" did not also return.) We discarded any paper that was not published in the research track of a reputable computer science conference or journal or was duplicative with another paper in the dataset (e.g., for work with both a conference paper and a journal extension, we only included the journal extension), resulting in a set of 187 papers. The authors are familiar with the relevant conferences and journals in programming languages and software engineering, and we used our judgment for these, erring on the side of inclusivity. For conferences or journals outside PL and SE, we included papers in any venue with a CORE ranking of A or A*.

We then examined each of the remaining papers in detail and recorded how many typestate specifications they contained, which specifications those were, and which of the specifications were accumulation typestate systems. When recording which specifications occurred in each paper we examined, we also recorded whether the specifications were duplicates of specifications that appeared in other papers. Among the papers we examined, 102 ($\approx 55\%$ of those examined closely, and $\approx 6\%$ of all Google Scholar hits) contained one or more typestate specifications. The venues that contributed papers with one or more typestate specifications to this study are: ECOOP (12), ESEC/FSE (12), ICSE (12), OOPSLA (10), PLDI (8), ISSTA (7), ASE (6), POPL (5), CCS (4), SAS (4), TOSEM (4), TSE (4), CC (2), ASPLOS (1), CAV (1), EuroSys (1), ICPC (1), IWACO (1), SAC (1), SOSP (1), TOPLAS (1), VMCAI

Table 3.1: The results of the literature survey. "TSA" stands for "TypeState Automata"; "ATSA" stands for "Accumulation TypeState Automata". All specification counts are without de-duplication.

Dataset	**Source**	**TSA**	**ATSA**	**ATSA%**
Papers since 2000 with <20 TSAs	101 scientific papers	302	67	22%
Dwyer et al. (1999) [101]	34 papers, tools, students	511	306	60%
Beckman et al. (2011) [28]	4 real Java projects	542	182	34%
Total	All of the above	1355	555	41%

(1), WWW (1).

3.3.2 Results

Table 3.1 summarizes the results. An artifact [179] contains our analysis of each relevant paper. That artifact also contains a finite-state machine for each typestate problem (as defined in section 3.3.2.1 below) we saw and the list of the papers we saw it in.

3.3.2.1 Papers Containing Examples

The 101 papers in this category contain 302 specifications, with a mean of 3 and a median of 2.

22% of these specifications are accumulation typestate systems. However, there is a significant amount of duplication between the papers in this dataset—many papers use the same few examples of typestate automata to motivate their general work on typestate.

We combined the typestate automata in these papers into categories for each *typestate problem*: for example, we counted every one of the 19 papers that we observed using the classic `File` example (fig. 3.1) into a single instance of the `File` typestate problem. Considering problems rather than specifications, we found that these 101 papers only contain 114 problems. Of those 114, 31 are accumulation typestate problems (27%), indicating that there is slightly more duplication among the non-accumulation typestate specifications. Perhaps this is because papers dealing with general typestate analysis want to motivate their use of an alias analysis—which requires at least one non-accumulation typestate example. We discuss this discrepancy further in section 3.3.3.

Next, we give the three most common examples of typestate problems that are accumulation and are not accumulation typestate systems.

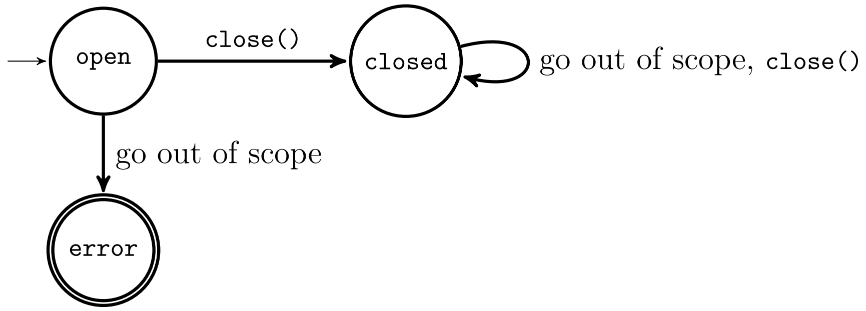


Figure 3.6: The typestate automaton for a resource leak, which is an accumulation typestate problem.

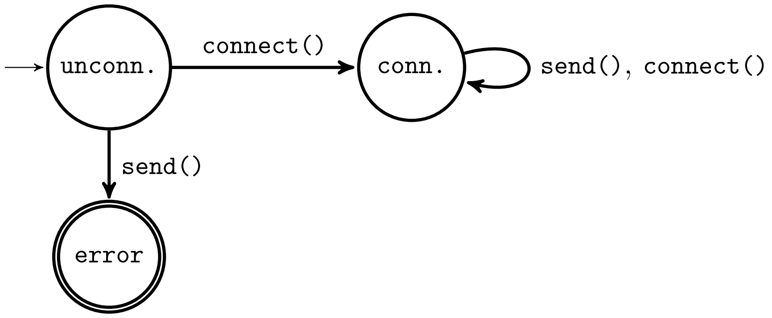


Figure 3.7: The typestate automaton for connecting a socket before sending data using it, which is an accumulation typestate problem.

Examples of Typestate Problems That Are Accumulation The problem of detecting resource leaks (fig. 3.6) appears 16 times across 14 papers[8] [82, 187, 328, 180, 315, 194, 197, 65, 108, 102, 17, 7, 313, 249].

The need to call a distinguished initialization method on an object after its constructor finishes but before using it appears 7 times across 4 papers [124, 82, 275, 324]. For example, when using a `Socket` object, one must call `connect()` before using it to send data (fig. 3.7).

A third common accumulation problem is that of object initialization: before an object is fully constructed, all of its logically-required fields must be set to reasonable values (fig. 3.8). This pattern appears 6 times across 6 papers [177, 178, 254, 108, 136, 152]. However, our literature survey has shown that bespoke analyses for other kinds of object initialization are

[8] We tried to stay as true as possible to the story each paper presented, which is why some automata appear multiple times in the same paper. The paper treated them differently, but we believe them to be the same example. For instance, [82] discusses memory leaks and leaked sockets, which are both resource leaks.

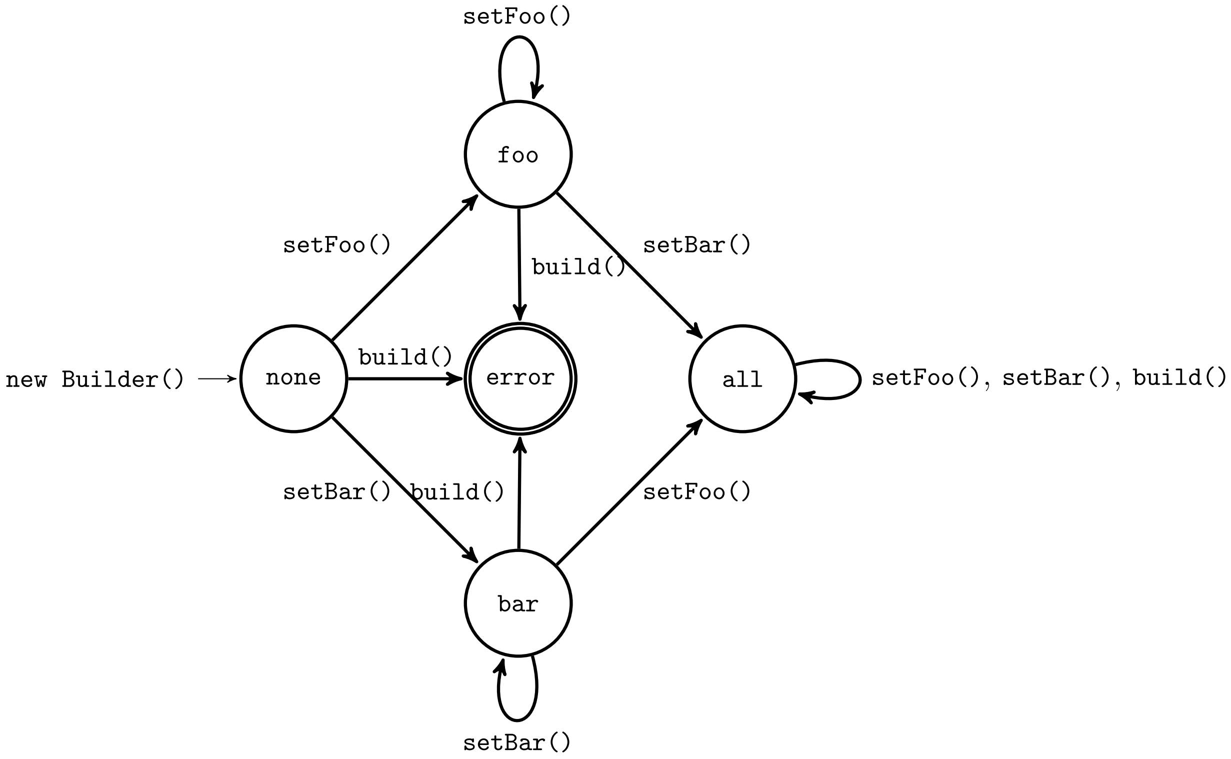


Figure 3.8: The typestate automaton for setting the required fields of an object before it is built, which is an accumulation typestate problem. This instance of the general pattern is specifically for a builder-pattern-style object construction pattern of a class with two required fields `foo` and `bar`.

also, in effect, bespoke accumulation analyses. For example, masked types [254] are a type system for ensuring that before a constructor exits, all non-null fields of the constructed class have been set to non-null values. This type system can be viewed as an accumulation analysis: the goal transition is the end of the constructor, and the enabling operations are the setting of the fields.

Examples of Typestate Problems That Are Not Accumulation The most common non-accumulation typestate problem is “don’t read or write to a stream or file after it is closed” (fig. 3.9), which appeared 31 times across 17 papers [124, 41, 45, 223, 130, 275, 33, 34, 253, 170, 203, 102, 326, 216, 324, 321, 46]. This problem is related to the file specification in fig. 3.1, but is slightly weaker—it assumes that the file is never re-opened. That this example is not accumulation demonstrates that accumulation typestate automata are a more interesting category than automata without loops other than self-loops (a category which

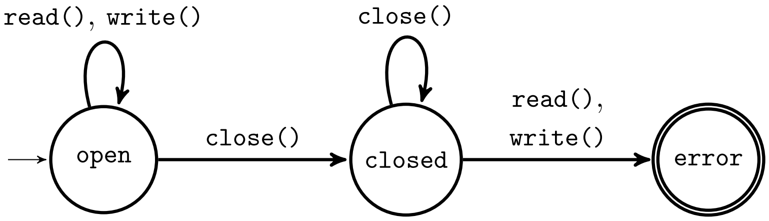


Figure 3.9: The typestate automaton for not reading or writing a stream after it has been closed, which is not an accumulation typestate problem.

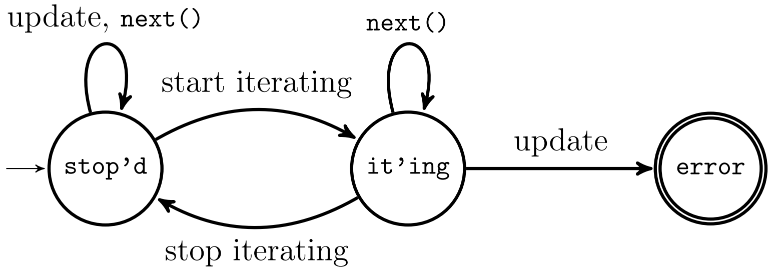


Figure 3.10: The typestate automaton for not updating a collection during iteration, which is not an accumulation typestate problem. Note that this automaton includes operations that are not method calls (e.g., "start iterating").

includes both this one and the three accumulation typestate examples above).

"Do not update a collection while iterating over it" (fig. 3.10) appeared 21 times across 14 papers [44, 317, 226, 135, 249, 321, 41, 45, 167, 161, 252, 253, 35, 223]. This property is representative of an important class of properties that are never accumulation typestate systems: "disable x after y" properties that forbid the programmer from performing operation x once operation y has been performed. The key reason that these properties cannot be checked without aliasing information—and are therefore not accumulation—is that the "disabling" operation ("start iterating" in this example) might be performed through any alias, but once it occurs, "update" must be prevented for all aliases.

The classic full file specification (fig. 3.1) appeared 20 times across 19 papers [138, 325, 287, 130, 139, 302, 320, 275, 324, 318, 7, 243, 81, 185, 11, 76, 328, 102, 103]. An interesting property of this specification is that some interesting parts of it could be enforced with an accumulation analysis if a slightly different design had been chosen for the API. In particular, if files could not be re-opened once they had been closed, enforcing "only call close after open" and "only call read after open" would become accumulation properties. Since most programmers usually create a new `File` object rather than re-using an existing one, this restriction would not be particularly burdensome, but would enable easier analysis.

3.3.2.2 Papers With Many Typestates

This section discusses two papers that report on large collections of typestate automata.

Patterns in Property Specifications for Finite-State Verification The first paper reports on 555 typestate-like specifications collected from a survey of 34 papers from the scientific literature, verification tool authors, and students in 1999 [101]. These 555 specifications were not de-duplicated. Because it precedes the start date for our survey, it is not included in the 187 papers in section 3.3.2.1. We include its data here for completeness, and to discuss the differences between their results and ours (section 3.3.3).

The primary goal of the paper was to categorize "finite-state properties"—that is, those expressible as finite-state machines—into patterns to help users of verification tools that take an FSM as input (such as typestate verifiers) create their own specifications by instantiating existing patterns. They categorized 511 of the 555 specifications into eight "patterns." Our analysis of these patterns is that instances of 5 of the 8 are always accumulation typestate systems (Existence, Precedence, Chain Precedence, Response, Chain Response), and some instances of a 6th (Bounded Existence, when the property is "at least" rather than "exactly" or "at most") are, as well. The 5 "always accumulation" patterns account for 306 of the 511 specifications that were categorized (60%).

An Empirical Study of Object Protocols in the Wild The second paper [28] studies the object protocols—that is, the behavioral specifications—of all classes in four large, open-source Java projects (one of which is the Java standard library). They also categorized these specifications based on common characteristics, much like the previous study, but they created their own set of categories.

The found 648 object protocols, which were not de-duplicated. We exclude their "type qualifier" category (106 specifications), which contains classes that behave as one of a fixed set of subtypes and can never change state. The remaining 542 protocols are typestate specifications.

Instances of their most common category, Initialization, are always accumulation typestate specifications. This category contains 182 of the 542 protocols (34%). The other 6 categories (66%) are not accumulation.

3.3.3 Discussion

It is interesting that both of the papers that reported on large sets of typestate properties included larger proportions of accumulation properties than our literature survey found otherwise. One possible explanation is that the scientific literature tends to include "exciting" or "challenging" problems—and, in the case of general typestate analysis, those problems usually involve aliasing (perhaps to justify the need for an alias analysis when analyzing an arbitrary typestate system, as we do in section 3.1 in reference to fig. 3.1). Another possible explanation is that neither of the papers that reported on large sets of specifications de-duplicated their specifications, so it could be that they contain many duplicate accumulation properties. When we de-duplicated the specifications in section 3.3.2.1, we found that non-accumulation typestate properties tended to be duplicated more often than accumulation typestate properties. This suggests that our results may be understating the prevalence of accumulation properties, which is good news: we have shown that accumulation properties are easier to check than general typestate properties.

Beckman et al. [28] is the most relevant to practical programmers interested in deploying accumulation analysis. An interesting avenue of future work would be a similar study to Beckman et al.'s on a larger corpus of software combined with a mechanization of our decision procedure for checking whether a typestate specification is accumulation, which would permit a more reliable estimate of the percentage of typestate specifications that appear in practice that are accumulation.

Another interesting observation is the relationship between different typestate specifications of the same type. For example, three of the examples we gave in section 3.3.2.1 are applicable to `File` objects: resource leaks (fig. 3.6), the classic file specification (fig. 3.1), and reading/writing a closed file (fig. 3.9). Enforcing all these properties with a single typestate analysis would necessarily require alias analysis, but enforcing just the resource leak property does not—and the same might be true of other partial specifications, such as "only call read after open"—especially if files cannot be re-opened after being closed. We suspect this may be a reason why prior work did not identify a category equivalent to accumulation: many accumulation properties are sub-properties of the full typestate specification of the relevant type. That said, accumulation properties are often interesting on their own—resource leaks, for example, are harder to detect dynamically than most other types of misuses of files—and we have shown that they are easier to enforce statically.

3.4 Building a Practical Accumulation Analysis

We implemented a general accumulation checker for Java using the Checker Framework [237] and have made it publicly available.[9] Our checker is general: the implementations of the specific accumulation analyses described in sections 3.5–3.7 all use the general accumulation checker as their basis. An accumulation analysis could be implemented modularly using any sound program analysis technique: dataflow analysis, abstract interpretation, type systems, etc. We chose a type system for convenience, and because types are naturally modular: type annotations on procedure boundaries and fields act as summaries, and local type inference infers operations that may have occurred within each procedure. Our implementation tracks enabling sets rather than enabling sequences (see section 3.4.3).

[9] `https://checkerframework.org/manual/#accumulation-checker`

3.4.1 Handling Other Features of Real Programming Languages

The core calculus in section 3.2.2.3 does not model features that are present in a practical programming language, including unanalyzed dependencies, open programs, class definitions, conditionals, inheritance, etc. Our formalism already handles some of these: for example, handling conditionals requires a may-aliasing oracle and estimated sets of typestates rather than a single typestate, both of which our formalism includes. Extending our proofs to other features is straightforward and does not require new proof techniques.

An advantage of accumulation analysis is that in practice it is possible to soundly handle code with unknown or "arbitrarily-bad" effects—including unmodeled features of the target language—by reverting to a safe default, in the same manner as an abstract interpretation might "go to top" in the presence of side effects. For example, if a call to an unanalyzed method might re-assign a field, an accumulation analysis can conservatively assume that that field's value is in the typestate automaton's start state after the call. This is sound as a consequence of lemma 1 and the definition of accumulation (in the same manner as lemma 2): the start state is necessarily a sound default assumption, because all goal transitions must be forbidden in it.

By contrast, in a non-accumulation typestate system it is not sound to fall back to the automaton's start state. For example, consider the `File` example in fig. 3.1: the start state is `closed`, where `open()` is a legal call. But treating all field reads as returning `closed` files would not be sound, because if the underlying `File` value was actually in the `open` state, a sound analysis should issue an error for a subsequent call to `open()`.

An advantage of our choice of a pluggable type system to implement our accumulation analyses is that the "start state" of a field can be changed by changing its declared type to specify a different typestate. In practice, this restricts that field to only containing values whose typestates are in the states reachable from the declared typestate—that is, the subautomaton composed of states reachable from the declared type. In practice, we have found that an accumulation analysis has sufficient precision when all field reads assume that the field is in the state corresponding to the declared type of the field.

3.4.2 Aliasing in Practical Accumulation Analyses

Practical accumulation analyses use cheap, targeted must-alias reasoning to improve the precision—that is, the false positive rate—of the analyses. For example, sections 3.5.3.3, 3.6.3, and 3.6.4 give lightweight aliasing analyses. These lightweight alias analyses compute only the aliasing information necessary to remove false positives that occurred in practice for these analyses, which makes them much cheaper than computing precise aliasing information for all variables (of types with typestate automata) in the program, as a whole-program alias analysis would.

A benefit of the accumulation analysis approach is that the core accumulation analysis (definition 4) is sound even without any alias reasoning, by corollary 1. But it is easy to utilize aliasing information that is readily available (or cheap to compute) to improve precision. In practice, using some aliasing information is necessary to achieve acceptable precision, and untracked aliasing is usually the single biggest cause of remaining false positives even after acceptable precision has been achieved. Our general accumulation checker includes both a suite of built-in cheap sound must-alias analyses (which includes those in sections 3.5.3.3, 3.6.3, and 3.6.4) and hooks for analysis developers to add further aliasing information.

3.4.3 Discussion: Accumulating Sets vs. Accumulating Subsequences

Section 3.2 uses the term "accumulation" to refer to two subtly different things. Accumulation analyses (definition 4) compute *sets* of operations. Accumulation typestate systems (definition 7) are defined by *(sub)sequences* of operations.

Definition 4 of accumulation analysis uses sets because that is how we actually implemented the accumulation analysis in this section. For an alternate definition of accumulation analysis in terms of subsequences, each goal operation would have an enabling sequence rather than an enabling set. Implementing an accumulation analysis based on this alternate definition would allow us to check "accumulation-like" properties that cannot be expressed as sets. For example, such an analysis could soundly check a property such as "call `a()` at least twice before calling `b()`" (i.e., a goal transition enabled by counting) or a property such as "call `a()` and `b()`, in that order, before calling `c()`" (i.e., a goal transition enabled by ordering).

In our literature survey (section 3.3), we found only three specifications with a goal transition enabled by ordering and none enabled by counting, which is why our implementation uses the (simpler) set abstraction. For example, in Figure 12 of [273], the authors describe a mined typestate specification for the Java KeyAgreement API. This API contains a method `generateSecret()`. Calling `generateSecret()` before `init()` and `doPhase()` is an error, so `generateSecret()` is a goal transition. However, `init()` and `doPhase()` also must be ordered: calling `doPhase()` before `init()` is also an error. The other two specifications in the literature (which appear in [273, 119]) that rely on ordering had a similar character to this example: describing some multi-stage initialization property where the initialization steps must be performed in some specific order.

3.5 A Practical Accumulation Analysis for Object Construction

3.5.1 Motivation

Objects in Java-like languages often have a combination of required and optional properties. For example, an API for a point might require `x` and `y` values, with `color` being optional. It would be legal for a client to supply $\{x, y\}$ or $\{x, y, \mathit{color}\}$, but not $\{x, \mathit{color}\}$. As another example, a bibliographic entry for a book might require `title` and either `author` or `editor`.

Ideally, an object construction API should:

- Only permit clients to supply permitted sets of values, ensuring at compile time that only well-formed objects can be created.

- Make code that constructs objects readable.
- Allow flexibility in client code, e.g., re-use of common initialization code in different scenarios.

The standard API for Java object construction contains one constructor for each combination of possible values that results in a well-formed object. This API satisfies the first requirement: if some combination is nonsensical, the API does not include the corresponding constructor. For example, every constructor for a point might require both an `x` and a `y` argument. At a constructor call site, invalid argument combinations are rejected by the compiler. However, this strategy fails the other two criteria. For readability, it is often difficult for clients to determine how an object is being constructed from the constructor invocation, particularly if multiple object properties have the same type. For complex classes, a constructor is needed for every possible combination of optional parameters, leading to a combinatorial explosion in constructor definitions. Finally, constructors provide little flexibility, as all parameters must be provided at once in a single call.

Due to these drawbacks of constructors, alternate patterns for object construction have been devised, such as the *builder pattern* [137]. 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 is easy for clients to use: at a client call site, the name of each setter method that is invoked indicates what is being set. The builder pattern avoids the combinatorial explosion problem of constructors, since one method exists per parameter, not per combination of parameters. Builders enable client-code flexibility, as code that calls a subset of setters can be abstracted into methods[10]. Popular frameworks like Lombok [293] and AutoValue [53] ease creation of builders by automatically generating a builder class from the class definition of the object to be constructed.

The builder pattern is important and widespread. The builder pattern is one of the original design patterns in the seminal "Gang of Four" book [137]. It was already a common design pattern in Smalltalk-80 [238]. Open-source projects that automatically generate builder classes are popular: Lombok has over 11,100 stars on GitHub, and AutoValue has over 9,700. The codebase of Amazon Web Services has over 769,000 uses of builders in non-test code [177], and both the Azure and AWS SDKs for Java provide builder-pattern-like APIs.

Unfortunately, usage of the builder pattern sacrifices some of the static safety provided by constructors. A client using a builder object can invoke any subset of the setter methods.

[10] For example, see the `setCommonFields` method in google/gapic-generator: `https://tinyurl.com/vhtyblw`

Effectively, the builder supports all 2^n possible constructors. Not all such combinations are valid, and a client can mistakenly use an illegal combination, which can lead to serious problems. Section 3.5.2.1 describes a security concern associated with improperly configured requests submitted to a public AWS API [219].

In other cases, the builder finalizer method throws an exception if a client invokes an invalid combination of setters. Programmers (and users!) find run-time crashes from builders frustrating. Hence, it would be highly desirable to have a tool that could *statically* verify builder usage, i.e., that clients only call valid combinations of setter methods. Such a static verifier for correct usage of a builder object b must perform two tasks:

1. Track which setter methods have been invoked on b at each program point.

2. When b's finalizer is invoked, ensure that all required setter methods have been invoked on b.

These tasks can be performed by an accumulation analysis as described in definition 4: the setter methods are the elements of the enabling sets, and the finalizer is the goal.

This section describes the design and implementation of that accumulation analysis as a flow-sensitive, specialized pluggable typechecker. Flow-sensitive type refinement can usually determine which setters have been invoked on a builder object automatically, without developer-written annotations. Our system can express disjunctions of required methods, crucial for handling cases like the aforementioned AWS security vulnerability (section 3.5.2.1). We present a type-based extension to our system that captures aliasing caused by the *fluent API* programming style frequently used with builders, where setter calls are chained (e.g., `b.setX().setY()....build()`). For common frameworks that generate builder classes, like Lombok and AutoValue, our tool automatically determines which logical arguments are required and which are optional, further reducing the need for manual annotation.

Our typechecker found 16 security vulnerabilities with only 3 false positives in over 9 million lines of industrial and open-source code. In open-source case studies, our typechecker found null-pointer violations and permitted the deletion of hundreds of lines of manually written, inflexible, error-prone builder code. In a small user study, users found the tool dramatically more useful and usable than the state of the practice.

The remainder of this section is organized around the following contributions:

- the identification of three real-world problems stemming from unsafe object construction (section 3.5.2),

- an accumulation analysis for reasoning about unsafe object construction (section 3.5.3),

- an evaluation of the accumulation analysis on the three problems presented in section 3.5.2 (section 3.5.4).

The section concludes with a discussion of related work on initialization (section 3.5.5).

```
DescribeImagesRequest request = new DescribeImagesRequest();
request.withFilters(new Filter("name", "RHEL-7.5_HVM_GA"));
api.describeImages(request);
```

Figure 3.11: Vulnerable client code that does not properly construct a request to the DescribeImagesRequest API, resulting in a potential “AMI sniping” concern.

```
package com.amazonaws.services.ec2.model;

public class DescribeImagesRequest {
  public DescribeImagesRequest() {...}
  public DescribeImagesRequest withOwners(String... owners) {...}
  public DescribeImagesRequest withFilters(Filter... filters) {...}
  public DescribeImagesRequest withImageIds(String... imageIds) {...}
}
```

Figure 3.12: The `DescribeImagesRequest` API. A client constructs a `DescribeImagesRequest`, modifies it via the `with*` methods, then sends it to AWS to obtain a machine image.

3.5.2 Examples of Unsafe Object Creation

This section illustrates three real-world examples of unsafe object construction: a security vulnerability caused by improper use of a builder in code that calls an AWS API, buggy usage of Lombok-generated builders, and buggy usage of AutoValue-generated builders. Our approach soundly detects all the problems described in this section.

3.5.2.1 AWS AMI Sniping

A client of a cloud services provider can create virtual computers programmatically, using the provider’s public API. An *image* is the virtual computer’s file system; it includes an operating system and additional installed software, and so it determines what code runs on the virtual computer.

For example, a client of Amazon Web Services indicates what image to use via the `DescribeImagesRequest` API (like the client in fig. 3.11). This API (fig. 3.12) requires clients to carefully create requests to avoid a potential operational security risk [219].

There are three safe ways to select which image to use when sending a request to the API:

- Use the `withImageIds` method to specify a globally unique image ID.

- Use the withFilters method to set some criteria (such as the name of the image, its operating system, etc.), *and* use the withOwners method to restrict the images searched to those owned by the requester or some other trusted party.

- Use the withFilters method to set criteria that restrict the image to one that is owned by a trusted party using the "owner", "owner-id", "owner-alias", or "image-id" filters.

The unsafe example in fig. 3.11 uses the "name" filter without an owner filter, which causes the API to return all the images that match the name. This introduces the potential for a so-called "AMI (Amazon Machine Image) sniping attack" [219], in which a malicious third party intentionally creates a new image whose name collides with the desired image, permitting the third party to surreptitiously inject their own code onto newly allocated machines. Any call that searches the public database without specifying some information that an adversary cannot fake is potentially vulnerable to a sniping attack and should be forbidden.

The vulnerability is an unsafe use of the builder pattern. DescribeImagesRequest is a builder: the with* methods are setters and the describeImages() call is the finalizer. Because the compiler permits all combinations of method calls, a client can accidentally fail to set the owner when setting the name, as in fig. 3.11.

Misuse of the API must be prevented, even though a client-side coding concern is not ordinarily eligible for a CVE [220, 228]. Revoking or changing the behavior of this widely-used API incompatibly could be a breaking change for customers, so AWS's proposed mitigation is for "customers to follow the best practice and specify an owner" [32]. An independent security researcher published instructions on how to detect if running virtual machines were impacted, but agreed that following best practices was the best available mitigation [245]. Our sound static analysis is better: it does not depend on programmers to remember to use the best practice.

3.5.2.2 Lombok builders

Lombok [330] is a widely-used Java code generation library that allows developers to avoid writing boilerplate code. Writing an @Builder annotation on class C generates a builder class for C. A client creates a builder object, incrementally adds information to it by calling setter methods corresponding to C's fields, and then calls the finalizer method build() to construct a C object. If some fields of C have types that are annotated as @NonNull, then build() throws a null-pointer exception if any such field has not been set.

A common cause of frustration for clients of such libraries is the addition of new @NonNull fields. For example, consider an application developer who depends on a library like Yubico/java-webauthn-server[11], which includes the class in fig. 3.13. Figure 3.14 is an example of such code, from java-webauthn-server's included demo. As defined, this code

[11] https://github.com/Yubico/java-webauthn-server

```
@Builder
public class UserIdentity {
    private final @NonNull String name;
    private final @NonNull String displayName;
    private final @NonNull ByteArray id;
}
```

Figure 3.13: A class that has a builder. The `@Builder` annotation causes Lombok to generate a builder at compile time. This example is simplified code from the Yubico/java-webauthn-server project.

```
UserIdentity.builder()
    .name(username)
    .displayName(displayName)
    .id(generateRandom(32))
    .build()
```

Figure 3.14: A client of the `UserIdentity` builder defined in fig. 3.13, from the same project. This builder use will not cause a run-time exception, because all fields whose type is `@NonNull` have been set.

works correctly. However, suppose that a developer of java-webauthn-server adds another field to `UserIdentity`. If this field's type is annotated as `@NonNull`, then the code in fig. 3.14 will begin to fail—at run time!—when the library dependency is updated. Even if this is caught during testing, debugging the cause can still be painful because the bug will manifest as a null-pointer exception in the unmodified client code. These sorts of bugs could be avoided by checking—at compile time—that the setter for each field whose type is non-null has been called before `build` is called.

Clients prefer compile-time checking that mandatory fields are set on builders; it is one of Lombok's most requested features [282, 171, 251, 58, 210, 225, 132, 27, 56, 61]. Reinier Zwitserloot, leader of the Lombok project, says "We get this feature request every other week: A way to have `@Builder` generate code such that things that are mandatory to set cause compile-time errors if you forget to set them" [329].

3.5.2.3 Google AutoValue

AutoValue [50] is a Java annotation processor that generates much of the boilerplate code for immutable Java classes, such as accessor methods for fields, `equals()`, `hashCode()`, and `toString()`. Like Lombok, AutoValue can also generate builder classes [53], which contain run-time checks to ensure that when `build()` is called on the builder, all required properties have been set. AutoValue generates builders as new subclasses of user-written abstract classes, whereas Lombok directly adds the builder to user-written code.

Run-time failures due to unset properties of AutoValue builders lead to pain points similar to those described for Lombok builders. Users desire a compile-time check that required properties are set, because in complex code this property can be difficult to test for [278]. Further, it can be difficult to discover which properties have default values and which need to be set by a client, complicating builder usage [221]. And, library upgrades can lead to run-time failures when properties in AutoValue types become required.[12]

3.5.3 A Specialized Pluggable Type System for Builders

This section presents our type system that guarantees required methods are always invoked on builder objects, which is an example of an accumulation analysis. Suppose there is a builder for this example `Book` class:

```
class Book {
  String title;  // required
  String author; // required
}
```

A client using the builder must call methods that set both the `title` and `author` fields, as in this example of safe code:

```
BookBuilder b = Book.builder();
b.title("Effective Java");
b.author("Joshua Bloch");
b.build();
```

To prove this code is safe, an analysis needs two kinds of facts:

- After each call to a setter s, the analysis must estimate that s has definitely been called on the receiver. Further, the analysis must also incorporate the previous estimate of called methods: after the call to `b.author()` above, the analysis must estimate that both `title` and `author` have been called on `b`.

[12] E.g., see `https://github.com/spotify/docker-client/issues/635`.

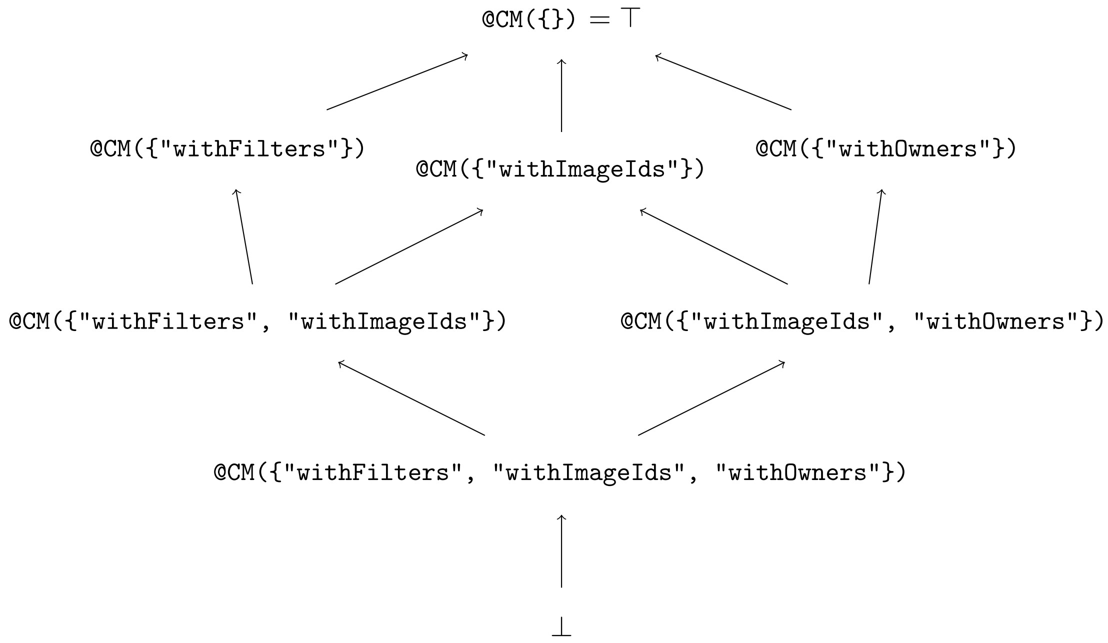

@CM({}) = ⊤
@CM({"withFilters"})
@CM({"withImageIds"})
@CM({"withOwners"})
@CM({"withFilters", "withImageIds"})
@CM({"withImageIds", "withOwners"})
@CM({"withFilters", "withImageIds", "withOwners"})
⊥

Figure 3.15: A part of the type qualifier hierarchy for the `@CalledMethods` type system; the full hierarchy is a lattice of arbitrary size. A type qualifier represents which methods have been called. "`@CM`" stands for `@CalledMethods`, for brevity. If an expression's type has qualifier `@CalledMethods({"withFilters", "withOwners"})`, then the methods `withFilters` and `withOwners` have definitely been called on the expression's value. Arrows represent subtyping relationships.

- `build` must have a specification to indicate that both `title` and `author` must have been called on its receiver.

We apply the decision procedure in algorithm 1 to determine that the typestate system corresponding to any builder is an accumulation typestate system. For any given builder, there is a typestate for each element in the powerset of the "required" logical arguments, and the finalizer method (i.e., `build()`) is only permitted from the final typestate in which all required logical arguments have been provided. An example of such a typestate system appears in fig. 3.8. The typestate automaton for the `Book` example in this section is nearly identical: `foo` is replaced by `author` and `bar` is replaced by `title` (or, without loss of generality, vice-versa). The remainder of this section details a modular, flow-sensitive, pluggable type system that implements such an accumulation analysis.

3.5.3.1 Estimating the methods called on an object

Our type system processes types of the form `@CalledMethods(`A`) T`, where `T` is a Java basetype and `@CalledMethods(`A`)` is a type qualifier. An expression with this type must evaluate to an instance of `T` (or a subclass of `T`) which has definitely had each method in A called on it. For example, after the call to `b.title()` above, the type of `b` is `@CalledMethods({"title"}) BookBuilder`. Our type system computes `@CalledMethods` types for every expression and method in the program, not just builders and setter methods.

Figure 3.15 shows part of the type qualifier hierarchy for `@CalledMethods` types. The subtyping rule for two `@CalledMethods` annotations, with sets of methods A and B, is:

$$\frac{A \supseteq B}{\texttt{@CalledMethods(A)} \sqsubseteq \texttt{@CalledMethods(B)}}$$

Our type system is flow-sensitive: a particular expression may have different types on different lines of the program, but must always be consistent with (a subtype of) the expression's declared type. Our type system relies on local type inference to compute updated expression types after method calls, e.g., updating `b`'s type qualifier to `@CalledMethods({"title"})` after the call to `b.title()`.

Though the type hierarchy has size up to 2^m where m is the number of methods in the program, the dataflow analysis (i.e., local type inference) is guaranteed to terminate: there are no unbounded ascending chains, which also means that there is no need to define widening operators (approximate $\sqcup$ operators).

In local type inference, processing of method calls is polymorphic. Say `b` has an inferred qualifier `@CalledMethods(`M`)` before a call `b.m()`. After the call, the inference computes `b`'s new qualifier as `@CalledMethods(`$M \cup \{m\}$`)`, independent of M.

Local type inference means that programmers need not write annotations within method bodies, but only on method signatures when there is inter-procedural flow of partially-completed builders. In such cases, the specifications (the type qualifiers) serve as valuable, machine-checked documentation.

As an example of a needed source-code annotation, consider this call to `describeImages()` in file `LatestImageProvider.java` in `https://github.com/iVirus/gentoo_bootstrap_java`:

```
public Optional<Image> get() {
    DescribeImagesResult result = ec2Client.describeImages(getRequest());
    ...
}
```

For each of the three overriding definitions of `getRequest()`, we added an `@CalledMethods` annotation to the return type that indicated that `withOwners()` had been called:

```
@CalledMethods("withOwners") DescribeImagesRequest getRequest() {...}
```

After adding those three annotations, our typechecker verifies the project. This also guarantees that each implementation of `getRequest()` does call `withOwners()`, since the typechecker verifies, not trusts, each annotation.

3.5.3.2 Specifying finalizer methods

Verifying correct use of a method requires a specification of that method. Consider the finalizer for the `BookBuilder` example:

```
interface BookBuilder {
  Book build(@CalledMethods({"title", "author"}) BookBuilder this);
}
```

Its specification states that the receiver for a call to `build` must be an object on which `title` and `author` have been called. This specification implements a goal from the underlying accumulation analysis: the goal operation is the annotated method (i.e., `build`), and the enabling operations are the arguments to the `@CalledMethods` annotation (i.e., `"title"` and `"author"`).

At each call to the finalizer (`build`), the typechecker checks that the builder argument passed as the receiver has an `@CalledMethods` qualifier that is a subtype of the declared receiver qualifier in the method signature. From our subtyping rule, this check ensures that at least the methods listed in the receiver qualifier have been invoked on the builder. If the check fails, the checker issues a type error, indicating possibly-defective code.

3.5.3.3 Fluent setters

Many builders are *fluent*: each setter method returns the builder again (i.e., the method returns `this`), so that calls can be chained.

Consider the following client code for the running `Book` example:

```
BookBuilder b = Book.builder();
b.title("Effective Java").author("Joshua Bloch");
Book theBook = b.build();
```

The local inference described in section 3.5.3.1 is insufficient to verify this code. After the second line, the inferred types are:

```
b : @CalledMethods({"title"}) BookBuilder
b.title("Effective Java") : @CalledMethods({"author"}) BookBuilder
```

The inferred type for `b` does not satisfy the specification of `build`. The key issue is *aliasing*: the return value of a fluent call is aliased with its receiver, but our system as described thus far is unaware of this fact (after all, by default an accumulation analysis uses no alias reasoning).

To verify this code, it is necessary to know that each fluent setter method returns its receiver. To express this specification, we introduce a new type annotation: `@This`. When written on a method's return type, it indicates that the return value of the method is always exactly the receiver object (`this` in Java). For the `Book` example, the setters should be specified as:

```
interface BookBuilder {
  @This BookBuilder title(String title);
  @This BookBuilder author(String author);
}
```

Our checker verifies `@This` annotations by ensuring the corresponding methods always return `this`.[13]

Given a call $e.m()$, the inference of section 3.5.3.1 computes an updated type for e. Given `@This` annotations, the inference performs two new types of updates. If m's return type has an `@This` qualifier, the inference also updates the `@CalledMethods` qualifier of $e.m()$ to be the same as the qualifier for e after the call. If e itself is a method call $e'.n()$ with an `@This` return type, the inference also updates the type of e' after the call, and recurses into e' as appropriate.[14] For the expression `b.title(...).author(...)`, since both `title` and `author` have `@This` annotations, the inference computes the types of `b`, `b.title(...)`, and `b.title(...).author(...)` to all be `@CalledMethods({"author","title"})`.

The analysis that verifies the `@This` annotation is an example of a cheap, local alias analysis that computes just enough aliasing information to remove a common kind of false positive when analyzing code that uses builders. A key advantage of an accumulation analysis like the one described in this section over a traditional typestate analysis is that the accumulation analysis is sound even with *no* aliasing information, but an accumulation analysis can also take advantage of aliasing information to improve precision. That means that an accumulation analysis can compute as much (or as little) aliasing information as necessary for the particular problem being solved.

3.5.3.4 Disjunctive types

Sometimes, a builder's specification requires one of two methods be called. For example, suppose that the `Book` class also has an `editor` field, and that a well-formed `Book` has either an author, an editor, or both. Then, clients like the following would be permitted:

```
Book b = Book.builder()
  .title("Advanced Topics in Types and Programming Languages")
  .editor("Benjamin Pierce")
  .build();
```

There is no corresponding `@CalledMethods` annotation that the API designer can write to specify the receiver type of the `build` method. We therefore introduce *disjunctive types*. Each of these types is a disjunction of `@CalledMethod` types. This means that, every set of `@CalledMethod` types has a perfectly precise least upper bound. (It already has a perfectly precise greatest lower bound: `@CalledMethods(`X`)` $\sqcap$ `@CalledMethods(`Y`)` $=$ `@CalledMethods(`$X \cup Y$`)`.)

For user convenience, we implement these disjunctions as a simple Boolean expression language which users write as an argument to a new type annotation called `@CalledMethodsPredicate`. The specification language uses the following grammar:

$S \rightarrow$ method name $\mid (S) \mid S \land S \mid S \lor S$

This permits the user to construct a specification like "author $\lor$ editor", expressed in Java as `@CalledMethodsPredicate("author || editor")`.

[13] Our checker also checks for valid method overriding, using standard support from the Checker Framework.

[14] Since chains of fluent calls are not overly long in practice (we did not observe any larger than about 20 methods), this recursion has negligible performance overhead.

Using @CalledMethodsPredicate to specify the AWS API As a practical example, the specification for the AMI sniping example (section 3.5.2.1) requires a disjunction. The corresponding specification is written on the parameter to the `describeImages` API in the AWS SDK (for presentation, the full specification has been shortened):

```
DescribeImageResponse describeImages(
    @CalledMethodsPredicate("withImageIds || withOwners")
    DescribeImageRequest request);
```

Given this specification for `describeImages`, the typechecker rejects any call whose receiver has not had either `withImageIds` or `withOwners` called on it. This specification is sound: it prevents all AMI sniping attacks.

Subtyping for disjunctive types We use formula implication to compute subtyping:

- `@CalledMethods(A)` $\sqsubseteq$ `@CalledMethodsPredicate(P)` if the set of methods A in the `@CalledMethods` annotation causes the predicate P to evaluate to true, then the `@CalledMethods` annotation is a subtype:

$$\frac{A \models P}{\texttt{@CalledMethods(A)} \sqsubseteq \texttt{@CalledMethodsPredicate(P)}}$$

- `@CalledMethodsPredicate(P)` $\sqsubseteq$ `@CalledMethodsPredicate(Q)` if ¬(P⇒Q) is unsatisfiable.

- `@CalledMethodsPredicate(P)` $\sqsubseteq$ `@CalledMethods(A)` if ¬(P⇒Q) is unsatisfiable, where Q is the conjunction of the methods in A.

3.5.3.5 Method effects

Sometimes programmers write methods that are wrappers for one or more calls to setters, to re-use common initialization logic. For example, suppose a programmer wrote this client code for the `Book` class:

```
  void setEjBookData(BookBuilder b) {
    b.title("Effective Java");
    b.author("Joshua Bloch");
  }

  ...
  BookBuilder b = Book.builder();
  setEjBookData(b);
  b.build();
```

The programmer needs to be able to specify the behavior of the `setEjBookData` method, which calls methods on its formal parameter. Without this specification, our checker will report an error at the `build` call, as our checker does not perform inter-procedural inference.

To specify such code, our implementation supports a method annotation `@EnsuresCalledMethods`. Its arguments are an expression and a set of methods that are called on that expression. So, `setEjBookData()` can be specified as:

```
@EnsuresCalledMethods("b", {"title", "author"})
void setEjBookData(BookBuilder b) {
  b.title("Effective Java");
  b.author("Joshua Bloch");
}
```

As with all annotations, it is checked, not trusted. The method annotated with `@EnsuresCalledMethods` typechecks only if `b`'s type at each exit point of the method is a subtype of `@CalledMethods("title", "author")`.

3.5.3.6 Implicit specifications

So far, this section has described how a programmer can specify methods. Our implementation infers most specifications for setter and finalizer methods, so programmers do not need to write them.

An `@This` type annotation is added to return types of setter methods in Lombok and AutoValue builders, as the generated code of such methods always returns `this`.

An `@CalledMethods` type annotation is added to builder finalizer methods generated by Lombok and AutoValue. For Lombok the methods in the annotation are the setters for any field whose type is `@NonNull`, except fields with an `@Singular` annotation and fields with an `@Builder.Default` annotation. For AutoValue, the methods in the annotation are the setters for each field whose type is not nullable, `Optional`, or a Guava Immutable type.

The Lombok authors are so excited by our work that Lombok now supports it directly. Lombok releases 1.18.10 and later can automatically insert `@This` and `@CalledMethods` annotations in Lombok-generated builders. This eliminates the need for our tool to add specifications in those classes.

3.5.3.7 Limitations

This type system guarantees that some methods are called before others. It does not guarantee that those methods are called with valid parameter values. For example, a programmer might pass an integer value that is out of the range required by the setter method's specification, or a programmer might pass a null value to a setter method requiring a non-null value. Existing type systems for the Checker Framework already verify these properties [237, 96, 176] and can be run together with our accumulation analysis. Or, a user could use a different analysis (e.g., NullAway [21]). A benefit of our approach is that it permits a user to use an arbitrary analysis for validating method arguments.

Other analyses can also be used to enhance reasoning about method arguments within an accumulation analysis. Consider the AMI sniping example in section 3.5.2.1. A common false positive when applying only the `@CalledMethods` type system to code that calls the `describeImages()` API is that it is also possible to specify an owner using a particular filter, without actually calling `withOwners()`. We plugged the Checker Framework's constant propagation analysis [116] into the `@CalledMethods` type system to eliminate these false positives, by treating calls that set an owner via a filter the same as direct calls to `withOwners()`.

Another limitation is that accumulation analysis does not—and cannot—guarantee that a method is *not* called, nor can it enforce a specification "either both methods are invoked or neither." Handling these cases soundly requires a sound alias analysis.

3.5.4 Evaluation

Our evaluation aims to answer these research questions:

- **RQ1:** Is our accumulation analysis sufficiently scalable and effective to find previously-unknown AMI sniping attacks in real-world programs?
- **RQ2:** Is our accumulation analysis useful to programmers when they work with frameworks that provide flexible builders at the cost of compile-time checking?

The version of the tool used in this evaluation, as well as the open-source portion of our scripts and data, is publicly available at `https://doi.org/10.5281/zenodo.3634993`. The current version of the tool, which is still being maintained, is distributed with the Checker Framework under the name "the Called Methods Checker." [87]

3.5.4.1 Finding AMI sniping bugs

We evaluated our approach to detecting AMI sniping attacks on two corpora of codebases:

- 36 open-source codebases from GitHub (about 427,000 lines of Java code). This corpus was collected by searching GitHub for projects that use the `describeImages` API, and then filtering out (for technical reasons) projects whose root directory did not contain a Gradle or Maven build file and those that did not build with a Java 8 compiler. We also discarded every copy or fork of the AWS Java SDK or a project already in the corpus.
- 509 codebases from Amazon Web Services that contain calls to the `describeImages()` API. These codebases contain about 8.7 million lines of Java source code.

The results appear in table 3.2. Our accumulation analysis found 13 AWS codebases potentially vulnerable to third-party abuse via AMI sniping. The developers fixed each potential vulnerability. Each of the 29 annotations was written on a helper method that wraps setter calls, similar to those discussed below in the open-source AutoValue case studies.

Including both sets of experiments, the tool overall achieved 84% precision, and required one annotation per 268,000 lines of code.

One true positive we discovered in the open-source evaluation was in the project Netflix/SimianArmy; the relevant code appears in fig. 3.16. If the list of image ids is null, then the code (by design) fetches every AMI available. Though the method's documentation does not say so, it is incumbent on any caller of this code to filter the result after the fact, and in fact the project's codebase contains call-sites that do not filter the results.

Both false positives in the open-source experiments (cases where our accumulation analysis could not verify safe code, even with additional annotations) were due to a single project which wraps the `describeImages` API with methods that take a list of `Filter` objects. Our type system cannot express that a list of `Filter` objects must contain the correct filters. The false positive in the closed-source code was due to a similar code pattern.

Table 3.2: Detection of AMI sniping vulnerabilities.

	Open	Closed
	source	source
Projects	36	509
Non-comment non-blank lines of Java code	427K	8.7M
Manually-written annotations	5	29
True positives	3	13
False positives	2	1

```
DescribeImagesRequest request = new DescribeImagesRequest();
if (imageIds != null) {
    request.setImageIds(Arrays.asList(imageIds));
}
DescribeImagesResult result = ec2Client.describeImages(request);
```

Figure 3.16: A true positive AMI sniping concern in Netflix's SimianArmy project.

3.5.4.2 Usefulness to programmers

There are two ways that programmers interact with our accumulation analysis:

- When a programmer begins using our tool, they need to **onboard** their project by running the checker and possibly writing annotations or changing their code.
- When a programmer makes a change to a project, the tool might issue a warning.

To evaluate the usefulness of our tools to programmers in each of these scenarios, we did two corresponding kinds of evaluation:

- Case studies: we ran our tool on existing programs. The case studies demonstrate the typical effort to find issues or to confirm the correctness of an existing project that was developed without our tools.
- A user study: we presented industrial engineers with common tasks related to modifying existing builders. The user study demonstrates that our tools ease editing existing code.

3.5.4.3 Case studies

The case studies (table 3.3) demonstrate the costs and benefits of onboarding an existing project. We sampled the projects from GitHub by searching for projects with significant Lombok or AutoValue builder usage that could compile with our infrastructure, preferring more popular projects where

Table 3.3: Verifying uses of the builder pattern. “Fr” is “framework”: either “L” (Lombok) or “AV” (AutoValue). “LoC” is lines of non-comment, non-blank Java code. “FCs” is the number of finalizer calls. “LoC+” and “LoC-” are the number of lines of code we added and removed, respectively. “Annos.” is number of manually-written annotations to specify existing methods. “TPs” is true positives. “FPs” is false positives, where our accumulation analysis could not guarantee that the call was safe, but manual analysis revealed that no run-time failure was possible.

Project	Fr	LoC	FCs	LoC+	LoC-	Annos.	TPs	FPs
Yubico/java-webauthn-server	L	7,153	42	52	426	48	0	3
javagurulv/clientManagementSystem	L	5,134	65	0	0	0	0	0
google/error-prone	AV	74,180	9	0	0	2	0	2
googleapis/gapic-generator	AV	49,054	442	2	0	58	1	1
google/nomulus	AV	71,627	95	0	0	23	0	8

possible (based on number of GitHub stars). The paper authors (who performed the case studies) were not familiar with the projects or their use of Lombok or AutoValue.

Lombok. We encountered two interesting patterns in the projects using Lombok: the mandatory stages pattern and usage of the Java `Optional` type.

The mandatory stages pattern. The java-webauthn-server project contained complex manually-written code to statically enforce that required fields are set in a specific order. This is called the *mandatory stages pattern*. If there are n mandatory fields, the code introduces $n - 1$ new builder types, each of which has a setter for only one field that returns the next builder type in the chain. The last one returns a standard builder instance that can be used to set optional fields. Figure 3.17 gives a simple example with just one required argument. When employing this pattern with multiple required arguments, the programmer must impose an order in which the arguments are to be set, or else create an exponential number of builder types. With our approach, none of these classes are necessary. In the case studies, we could safely delete them.

Initializing fields of `Optional` type Lombok permits users to manually write parts of the builder that Lombok would otherwise generate. The java-webauthn-server program used this facility extensively to permit fields with `Optional<T>` to have both a setter that takes a `T` as an argument and a setter that takes an `Optional<T>`, like the code in fig. 3.18. When writing a setter manually, the user also has to manually write the `@This` annotation. All 48 annotations in java-webauthn-server were `@This` annotations on manually-written setters for `Optional`s. The use of `Optional` is a questionable design decision [111]. The Lombok authors advocate using `null` to indicate an optional value when using Lombok builders [276], and doing so avoids the need for either manually-written setters or `@This` annotations. This pattern also required us to add some code that Lombok would normally have generated, but which the original, hand-written code elided—showing the danger of hand-writing

```
public static StartRegistrationOptionsBuilder.MandatoryStages builder() {
    return new StartRegistrationOptionsBuilder.MandatoryStages();
}

public static class StartRegistrationOptionsBuilder {
    public static class MandatoryStages {
        private final StartRegistrationOptionsBuilder builder = new StartRegistrationOpt

        public StartRegistrationOptionsBuilder user(UserIdentity user) {
            return builder.user(user);
        }
    }
}
```

Figure 3.17: Code from the project Yubico/java-webauthn-server which uses a complex Java type to force programmers to set required fields in a builder. This code is from the `StartRegistrationOptions` class. Note that this code replaces generated code, so with our approach all code in this figure can be safely deleted.

```
class StartAssertionOptions {
  private final @NonNull Optional<Long> timeout;

  static class StartAssertionOptionsBuilder {
    private @NonNull Optional<Long> timeout = Optional.empty();

    public @This StartAssertionOptionsBuilder timeout(long t) {
      return this.timeout(Optional.of(t));
    }
  }
}
```

Figure 3.18: Manually-written `timeout()` setter method from the project Yubico/java-webauthn-server which requires an `@This` annotation.

```
static @CalledMethods({"baseDirectory","inPlace"}) Builder builder() {
  return new AutoValue_ErrorProneOptions_PatchingOptions.Builder()
    .baseDirectory("")
    .inPlace(false);
}
```

Figure 3.19: Example AutoValue builder code, adapted from google/error-prone, that sets default values for required fields.

code in this way.

AutoValue The most common code pattern in the AutoValue case studies requiring manual annotation was setting of default values when creating a builder [51]. Figure 3.19 shows an example, adapted from the google/error-prone benchmark. Here, the `builder()` method used to construct a new builder sets the `baseDirectory` and `inPlace` properties to default values before returning the builder. Hence, client code need not explicitly set these properties before calling `build()`. A `@CalledMethods` annotation documents this fact.

AutoValue users have discussed the difficulty of finding which properties have default values when the above pattern is used [221]. Our introduced `@CalledMethods` annotations ease this problem by making the defaulted properties evident from the method signature.

The second most common need for annotations was when a builder is passed to a method that sets several required properties. We annotated the method with `@EnsuresCalledMethods`. We believe these annotations in particular are useful documentation, as it was non-obvious in many such cases why the code was safe.

We added a default case for one switch statement (two lines of code), capturing the fact that the other cases were exhaustive and enabling our tool to reason that a property was always set.

Our accumulation analysis found a defect in googleapis/gapic-generator (fig. 3.20). The `packageInfo` variable holds the relevant builder, and required method `packageInfo.outputPath()` is only invoked if the `Optional` returned by `findFirst()` is present. If the `Optional` is absent, then the call to `packageInfo.build()` will throw a run-time error. We reported the bug to the developers, who promptly verified and fixed the issue, saying "your static analysis tool sounds truly amazing!" [281] For the one false positive in gapic-generator, a non-trivial global invariant ensures the relevant property is always set.

The accumulation analysis reported 10 total false positive warnings in google/nomulus and google/error-prone. In all cases, the false positives were due to use of AutoValue features that our tool does not automatically support, like manually writing a builder's `build()` method with delegation to a generated `autoBuild()` method [52]. Adding support for such patterns is future (engineering) work.

```
model
    .getInterfaces(productConfig)
    .stream()
    .filter(productConfig::hasInterfaceConfig)
    .map(InterfaceModel::getFullName)
    .findFirst()
    .map(name -> pathMapper.getOutputPath(name, productConfig))
    .ifPresent(path -> packageInfo.outputPath(path + File.separator + "package-info.
[...]
return packageInfo.build();
```

Figure 3.20: Excerpt of real bug discovered in googleapis/gapic-generator by our accumulation analysis.

3.5.4.4 User study

To explore the usefulness of our accumulation analysis for a software engineer modifying a project that uses a builder, we undertook a small user study.

Participants Each participant was employed as a software engineer, regularly used Java, and was familiar with Lombok. Participants were not familiar with our tool. We recruited 6 participants; all were at the same "level" within their organization (i.e., they had the same job title) but worked on different teams.

Methodology The task for the study was to add a new required field to a class with an existing Lombok-generated builder, and then update all call sites to provide a reasonable value (each call site, if not updated, will throw an exception if executed).

The task was carried out on java-webauthn-server, one of the case studies in section 3.5.4.3. Participants started with a fully-annotated codebase that type-checks with the our accumulation analysis enabled; they were not required to onboard the tool. The original project has some tests written in Scala; we removed those, because our tool does not handle Scala code. This also allowed us to simulate another class of problems: changes to classes whose builders are not covered by tests.

We chose two different classes for participants to add a new field to. One task's class had a test case written in Java; the other class had no test. We used a factorial design: each participant executed the task for both of these classes; for one, they had access to our tool, and for the other, they did not. To control for learning effects, both the order of the tasks and the order of tool/not-tool were randomized independently for each participant.

No training on our tool was provided. Its messages came to participants via the standard compiler interface.

Measurement We recorded how long it took each participant to complete each task (participants were capped at one hour per task, though most were much faster). We also measured whether they completed each task correctly—defined by running the held-out Scala tests. We also surveyed the participants after they had completed the tasks. We asked the following questions:

- How often do you encounter tasks like those in the experiment in your day-to-day work?
- Did you find compiler messages indicating where required fields had not been set useful?

Results 3/6 participants failed to complete the task without our tool (two in the condition lacking a failing test), but all 6 succeeded with our tool. There was a difference in means in the time taken when considering only those who finished both tasks: using our tool was about 1.5x faster (≈200 seconds vs. ≈306 seconds).

In the surveys, 5/6 users said they encountered tasks like these at least monthly. The subjects were also convinced that the compile-time warnings were useful. For example, one subject said "It was easier to have the tool report issues at compile time." Several also mentioned the tool's value in localizing where to make changes: for example, one said the tool "allowed me to immediately hone in on the problem."

3.5.4.5 Threats to validity

The analyzed projects are written in Java, so our results might not generalize to other languages.

Our small user study uses only a few developers from a single company, and therefore may not be representative.

There is a threat to construct validity in the user study: the subjects may have guessed that we were evaluating the accumulation analysis, since they were familiar with Lombok but not with our work.

3.5.5 Related work: Initialization

Object Construction: There is scant related work directly on static analyses to ensure that all mandatory setters are called before a finalizer in the builder pattern. However, frustration with traditional constructors motivates some language design choices such as named and default parameters in languages like Python. The closest works are tools that generate builders that require clients to set mandatory fields in a pre-defined order before setting any optional fields. Examples include the AutoValue Step builder [274] and the Jilt library [264]. Type-safe builders can also be encoded using phantom types [127] or in the Scala type system [122]. Recent work shows how to generate a fluent API encoding a deterministic context-free language in Java while preserving type safety [142], which could in principle be used to generate a type-safe builder. All these techniques require either an exponential number of classes in the number of logical parameters, setting parameters in a pre-defined order, or both; none of them can be applied to legacy code without modifying it. Our analysis does not require programmers to rewrite their builders, does not require methods be called in a particular order, and does not require exponentially-many classes.

Object Initialization: Another category of related approaches are type systems and other static analyses for detecting nullness errors, especially those caused by object initialization. For example, freedom before commitment [286] type systems for reasoning about the initialization of objects defend against null pointer exceptions generally, but require significantly more annotations than our more-specialized approach, and are also less general in that they cannot be used for errors that will not throw a null-pointer exception, like our AMI sniping example. Similar type systems exist for Java bytecode [163]. Delayed [121] and mask [254] types track the fields that have been initialized on an object, and permit specifications on methods that require certain fields to be set before the method is invoked. Mask types can be viewed as a sort of accumulation analysis: enabling operations are the setting of fields, and the goal is some method that requires those fields to have been set. In this section, our focus is on builders; mask types are to an object's internal state in terms of the fields that have been set what our accumulation analysis is to the externally-visible initialization status of a builder (i.e., which logical arguments have been provided).

3.6 A Practical Accumulation Analysis for Resource Leaks

3.6.1 Motivation

A resource leak occurs when some finite resource managed by the programmer is not explicitly disposed of. In an unmanaged language like C, that explicit resource might be memory; in a managed language like Java, it might be a file descriptor, a socket, or a database connection. Resource leaks continue to cause severe failures, even in modern, heavily-used Java applications [141]. This state-of-the-practice does not differ much from two decades ago [308]. Microsoft engineers consider resource leaks to be one of the most significant development challenges [207]. Preventing resource leaks remains an urgent, difficult, open problem.

Ideally, a tool for preventing resource leaks would be:

- *applicable* to existing code with few code changes,
- *sound*, so that undetected resource leaks do not slip into the program;
- *precise*, so that developers are not bothered by excessive false positive warnings; and
- *fast*, so that it scales to real-world programs and developers can use it regularly.

Prior approaches fail at least one of these criteria. Language-based features may not apply to all uses of resource variables: Java's try-with-resources statement [233], for example, can only close resource types that implement the `java.lang.AutoCloseable` interface, and cannot handle common resource usage patterns that span multiple procedures. Heuristic bug-finding tools for leaks, such as those built into Java IDEs including Eclipse [104] and IntelliJ IDEA [165], are fast and applicable to legacy code, but they are unsound. Inter-procedural typestate or dataflow analyses [295, 328] achieve more precise results—though they usually remain unsound—but their whole-program analysis can require hours to analyze a large-scale Java program. Finally, ownership type systems [69] as employed in languages like Rust [183] can prevent nearly all resource leaks, but using them would require a significant rewrite for a legacy codebase, a substantial task which is often infeasible.

The goal of a leak detector for a Java-like language is to ensure that required methods (such as `close()`) are called on all relevant objects; we deem this a *must-call* property. Verifying a must-call property requires checking that required methods (or *must-call obligations*) have been called at any point where an object may become unreachable. A static verifier does this by computing an under-approximation of invoked methods. Our key insight is that checking of must-call properties is an accumulation problem, and hence does not require heavyweight whole-program analysis. The contribution of this section is a resource leak verifier that leverages this insight to satisfy all four requirements: it is applicable, sound, precise, and fast.

Section 3.5 presented an accumulation analysis for verifying that certain methods are invoked on each object before a specific call (e.g., `build()`). Resource leak checking is similar in that certain methods must be invoked on each object before it becomes unreachable. An object becomes unreachable when its references go out of scope or are overwritten. By making an analogy between object-unreachability points and method calls, we show that resource leak checking is an accumulation problem and hence is amenable to sound, modular, and lightweight analysis.

There are two key challenges for this leak-checking approach. First, due to subtyping, the declared type of a reference may not accurately represent its must-call obligations; we devised a simple type system to soundly capture these obligations. Second, the approach is sound, but highly imprecise without targeted reasoning about aliasing. The most important aliasing patterns to handle are:

- copying of resources via parameters and returns, or storing of resources in final fields (the RAII pattern [285]);
- wrapper types, which share their must-call obligations with one of their fields; and,
- storing resources in non-final fields, which might be lazily initialized or written more than once.

To address this need, we introduced an intra-procedural dataflow analysis for alias tracking. Unlike a traditional, whole-program alias analysis, our analysis underapproximates the aliasing in the program (and the fact that our core analysis is an accumulation analysis makes this underapproximation sound). We extended this simple alias analysis with three sound techniques to improve the precision of our accumulation analysis:

- a lightweight ownership transfer system. This system indicates which reference is responsible for resolving a must-call obligation. Unlike typical ownership type systems, our approach does not impact the privileges of non-owning references.
- resource aliasing, for cases in which a resource's must-call obligations can be resolved by closing one of multiple references.
- a system for creating new obligations at locations other than the constructor, which allows our system to handle lazy initialization or re-initialization.

Variants of some of these ideas exist in previous work. We bring them together in a general, modular manner, with full verification and the ability for programmers to easily extend checking to their own types and must-call properties. Our approach occupies a novel point in the design space for a leak detector: unlike most prior work, it is sound; it is an order of magnitude faster than state-of-the-art whole-program analyses; it has a false positive rate similar to a state-of-the-practice heuristic bug-finder; and, though it does require manual annotations from the programmer, its annotation burden is reasonable: about 1 annotation for every 1,500 lines of non-comment, non-blank code.

The remainder of this section is structured around the following contributions:

- the insight that the resource leak problem is an accumulation problem, and an analysis approach based on this fact (section 3.6.2).
- three innovations that improve the precision of our analysis via targeted reasoning about aliasing: a lightweight ownership transfer system (section 3.6.3), a lightweight resource-alias tracking analysis (section 3.6.4), and a system for handling lazy or multiple initialization (section 3.6.5).
- an open-source implementation for Java, called the Resource Leak Checker.
- an empirical evaluation (section 3.6.6): case studies on heavily-used Java programs, an ablation study that shows the contributions of each innovation to the Resource Leak Checker's precision, and a comparison to other state-of-the-art approaches that demonstrates the unique strengths of our approach.

We conclude with a survey of related work in the domain of resource leak checking (section 3.6.8).

3.6.2 Leak Detection via Accumulation

This section presents a sound, modular, accumulation-based resource leak checker ("the Resource Leak Checker"). Sections 3.6.3–3.6.5 soundly enhance its precision.

The Resource Leak Checker is composed of three cooperating analyses:

1. a taint-tracking type system computes a conservative *overapproximation* of the set of methods that might need to be called on each expression in the program (section 3.6.2.1).
2. an accumulation type system computes a conservative *underapproximation* of the set of methods that are actually called on each expression in the program (section 3.6.2.2), based on the accumulation analysis in section 3.5.3).
3. a dataflow analysis checks consistency of the results of the two above-mentioned type systems and provides a platform for targeted alias reasoning. It issues an error if some method that might need to be called on an expression is not always invoked before the expression goes out of scope or is overwritten (section 3.6.2.3).

3.6.2.1 Tracking Must-Call Obligations

We use a taint-tracking type system tracks which methods might need to be called on a given expression. This type system—and our entire analysis—is not specific to resource leaks. Another such property is that the `build()` method of a builder [137] should always be called (note that this is a different property of builders than in section 3.5, where the property being checked is that when a `build()` method is called, all required arguments have been provided).

```
Socket s = null;
try {
  s = new Socket(myHost, myPort);
} catch (Exception e) { // do nothing
} finally {
  if (s != null) {
    s.close();
  }
}
```

Figure 3.21: A safe use of a `Socket` resource.

$$\begin{tikzcd}
 & \texttt{@MustCallUnknown} = \top & \\
 & \texttt{@MustCall(\{"a", "b"\})} \arrow[u] & \\
\texttt{@MustCall(\{"a"\})} \arrow[ur] & & \texttt{@MustCall(\{"b"\})} \arrow[ul] \\
 & \texttt{@MustCall(\{\})} = \bot \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Figure 3.22: Part of the `MustCall` type hierarchy for representing which methods must be called; the full hierarchy is a lattice of arbitrary size. If an expression's type has qualifier `@MustCall({"a", "b"})`, then the methods "`a`" and "`b`" might need to be called before the expression is deallocated. Arrows represent subtyping relationships.

Our taint-tracking type system supports two qualifiers: `@MustCall` and `@MustCallUnknown`. The `@MustCall` qualifier's arguments are the methods that the annotated value must call. The declaration `@MustCall({"a"}) Object obj` means that before `obj` is deallocated, `obj.a()` might need to be called. The Resource Leak Checker conservatively requires all these methods to be called, and it issues a warning if they are not.

For example, consider fig. 3.21. The expression `null` has type `@MustCall({})`—it has no obligations to call particular methods—so `s` has that type after its initialization. The `new` expression has type `@MustCall("close")`, and therefore `s` has that type after the assignment. At the start of the `finally` block, where both values for `s` flow, the type of `s` is their least upper bound, which is `@MustCall("close")`.

Note that the type `@MustCall("close")` can represent anything that *might* need to call `close()`: for example, at the entrance to the `finally` block in fig. 3.21, `s`'s actual value might either be `null`, which does not need to call any methods, or an open `Socket`, which does. Thus, either the obligation to close or no obligation at all can be represented by the static type `@MustCall({"close"}) Socket`, which can be read as "a Socket that might need to call close before it is deallocated".

Part of the type hierarchy appears in fig. 3.22. All types are subtypes of `@MustCallUnknown`. The subtyping relationship for `@MustCall` type qualifiers is:

$$\frac{A \subseteq B}{\texttt{@MustCall(A)} \sqsubseteq \texttt{@MustCall(B)}}$$

The default type qualifier is `@MustCall({})` for base types without a programmer-written type qualifier.[15] Our implementation provides JDK annotations that require that every object of `Closeable` type must have the `close()` method called before it is deallocated, with exceptions for types that do not have an underlying resource, e.g., `ByteArrayOutputStream`.

3.6.2.2 A Type System for Called Methods

The Called Methods type system tracks a conservative underapproximation of which methods have been called on an expression: it is an accumulation analysis that computes the methods that have been called on each program expression. It is an extension of the similar system described in section 3.5.3 for the builder pattern. The primary difference in this version is that a method is considered called even if it throws an exception—a necessity in Java because the `close()` method in `java.io.Closeable` is specified to possibly throw an `IOException`. In the system in section 3.5.3, a method was only considered "called" when it terminated successfully. Further, this system has no explicit goal operation (unlike the system in section 3.5.3, whose goal operation is usually `build()`); instead, the goal operation is "go out of scope," and going-out-of-scope points are computed by the Consistency Checker.

3.6.2.3 The Consistency Checker

Given `@MustCall` and `@CalledMethods` types, the Consistency Checker ensures that the `@MustCall` methods for each object are always invoked before it becomes unreachable, via an intra-procedural dataflow analysis. We employ dataflow analysis to enable targeted reasoning about aliasing, crucial for precision. Here, we present a simple, sound version of the analysis. Sections 3.6.3–3.6.5 describe sound enhancements to this approach.

Language For simplicity, we present the analysis over a simple assignment language in three-address form. An expression e in the language is `null`, a variable `p`, a field read `p.f`, or a method call `m(p1,p2,...)` (constructor calls are treated as method calls). A statement s takes one of three forms: `p = e`, where e is an expression; `p.f = p'`, for a field write; or `return p`. Methods are represented by a control-flow graph (CFG) where nodes are statements and edges indicate possible control flow. We elide control-flow predicates because the consistency checker is path-insensitive.

For a method CFG, CFG.*statements* is the statements, CFG.*formals* is the formal parameters, CFG.*entry* is its entry node, CFG.*exit* is its exit node, and CFG.*succ* is its successor relation. For a statement s of the form `p = e`, $s.LHS = p$ and $s.RHS = e$.

[15] For unannotated local variable types, flow-sensitive type refinement infers a qualifier.

Algorithm 2: Finding unfulfilled `@MustCall` obligations in a method. Algorithms 3 and 4 define helper functions.

```
procedure FindMissedCalls(CFG):
    input  : A control-flow graph CFG
    output : A must-call violation if one exists.
    /* $D$ maps each statement $s$ to a set of dataflow facts reaching $s$. Each fact is of
       the form $\langle P, e \rangle$, where $P$ is a set of variables that must-alias $e$ and $e$ is an
       expression with a nonempty must-call obligation.                                   */
    $D \leftarrow$ InitialObligations(CFG)
    while $D$ has not reached fixed point do
        foreach $s \in$ CFG.statements, $\langle P, e \rangle \in D(s)$ do
            if $s$ is exit then
                report a must-call violation for $e$
            else if $\neg$MCSatisfiedAfter($P, s$) then
                $kill \leftarrow s$ assigns a variable ? $\{s.LHS\}$ : $\emptyset$
                $gen \leftarrow$ CreatesAlias($P, s$) ? $\{s.LHS\}$ : $\emptyset$
                $N \leftarrow (P - kill) \cup gen$
                $\forall t \in$ CFG.succ($s$) . $D(t) \leftarrow D(t) \cup \{\langle N, e \rangle\}$
```

Algorithm 3: Finding the initial `@MustCall` obligations in a method. Algorithm 4 defines helper functions.

```
procedure InitialObligations(CFG):
    input  : A control-flow graph CFG
    output : The initial must-call obligations when control enters CFG.
    $D \leftarrow \{s \mapsto \emptyset \mid s \in$ CFG.statements$\}$
    foreach $p \in$ CFG.formals, $t \in$ CFG.succ(CFG.entry) do
        if HasObligation($p$) then
            $D(t) \leftarrow D(t) \cup \{\langle \{p\}, p \rangle\}$
    foreach $s \in$ CFG.statements of the form p = m(p1, p2, ...) do
        $\forall t \in$ CFG.succ($s$) . $D(t) \leftarrow D(t) \cup$ FactsFromCall($s$)
    return $D$
```

Algorithm 4: Helper functions for algorithms 2 and 3. Except for MCAfter and CMAfter, all functions will be replaced with more sophisticated versions in sections 3.6.3–3.6.5.

```
procedure HasObligation(e):
    input  : An expression $e$.
    output : Does $e$ introduce a must-call obligation to check?
    return $e$ has a declared @MustCall type
procedure FactsFromCall(s):
    input  : A call statement $s$ of the form p = m(p1, p2, ...)
    output : New must-call obligations from the call.
    $p \leftarrow s.LHS, c \leftarrow s.RHS$
    return HasObligation($c$) ? $\{\langle\{p\}, c\rangle\}$ : $\emptyset$
procedure MCSatisfiedAfter(P, s):
    input  : A set of variables $P$ and a statement $s$.
    output : Is the must-call obligation for $P$ satisfied after $s$?
    return $\exists p \in P$. MCAfter($p, s$) $\subseteq$ CMAfter($p, s$)
procedure CreatesAlias(P, s):
    input  : A set of variables $P$ and a statement $s$.
    output : Does $s$ introduce a must-alias for a variable in $P$?
    return $\exists$q $\in P$ . $s$ is of the form p = q
procedure MCAfter(p, s):
    input  : A variable $p$ and a statement $s$.
    output : The must-call obligations of $p$ after $s$.
    return methods in @MustCall type of $p$ after $s$
procedure CMAfter(p, s):
    input  : A variable $p$ and a statement $s$.
    output : The methods that have definitely been called on $p$ after $s$.
    return methods in @CalledMethods type of $p$ after $s$
```

Pseudocode Algorithm 2 gives pseudocode for the basic version of our checker, with helper functions in algorithms 3 and 4. At a high level, the dataflow analysis computes a map D from each statement s in a CFG to a set of facts of the form $\langle P, e\rangle$, where P is a set of variables and e is an expression. The meaning of D is as follows: if $\langle P, e\rangle \in D(s)$, then e has a declared `@MustCall` type, and all variables in P are *must aliases* for the value of e at the program point before s. Computing a set of must aliases is useful since any must alias may be used to satisfy the must-call obligation of e. Using D, the analysis finds any e that does not have its `@MustCall` obligation fulfilled, and reports an error.

```
s = new Socket(...); // 1
if (...) {
  s = null; // 2
} else {
  t = s; // 3
  close(t); // 4
}
```

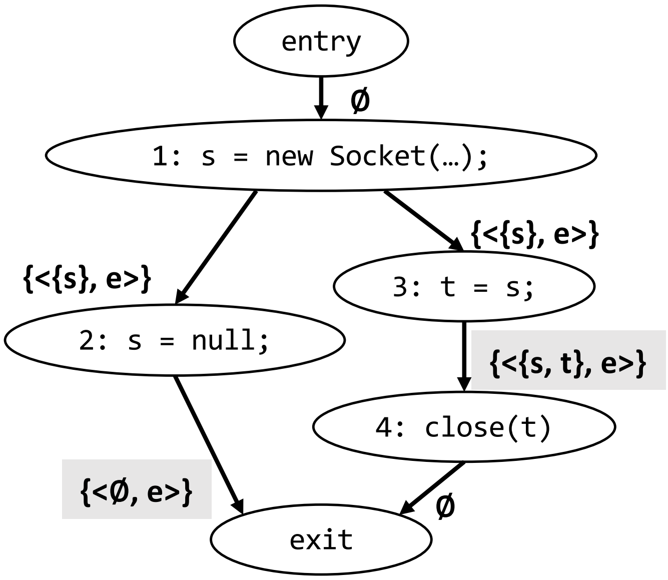


Figure 3.23: Example code and CFG for illustrating algorithm 2. "e" is "`new Socket(...)`". Non-shaded facts are created by InitialObligations, and shaded facts are propagated by the fixed-point loop.

Algorithm 2 proceeds as follows. First, it invokes InitialObligations (algorithm 3) to initialize D. Only formal parameters or method calls can introduce obligations to be checked (reads of local variables or fields cannot). The fixed-point loop iterates over all facts $\langle P, e\rangle$ present in any $D(s)$ (our implementation uses a worklist for efficiency). If s is the exit node, the obligation for e has not been satisfied, and an error is reported. Otherwise, the algorithm checks if the obligation for e is satisfied after s. For the basic checker, MCSatisfiedAfter in algorithm 4 checks whether there is some $p \in P$ such that after s, the set of methods in p's `@MustCall` type is contained in the set of methods in its `@CalledMethods` type; if true, all `@MustCall` methods have already been invoked. This check uses the inferred flow-sensitive `@MustCall` and `@CalledMethods` qualifiers described above.

If the obligation for e is not yet satisfied, the algorithm propagates the fact to successors with an updated set N of must aliases. N is computed in a standard gen—kill style. The kill set simply consists of whatever variable (if any) appears on the left-hand side of s. The gen set is computed by checking if s creates a new must alias for some variable in P, using the CreatesAlias routine. Since our analysis is accumulation, CreatesAlias could simply return false without impacting soundness. In algorithm 4, CreatesAlias handles the case of a variable copy where the right-hand side is in P. (Section 3.6.4 presents more sophisticated handling.) Finally, the algorithm propagates the new fact to successors. The process continues until D reaches a fixed point.

Example To illustrate our analysis, fig. 3.23 shows a simple program (irrelevant details elided) and its corresponding CFG. The CFG shows the dataflow facts propagated along each edge. For initialization, statement 1 introduces the fact $\langle\{s\}, e\rangle$ (where e is the `new Socket(...)` call) to $D(2)$ and $D(3)$. At statement 2, s is killed, causing $\langle\emptyset, e\rangle$ to be added to $D(exit)$. This leads to an error being reported for statement 1, as the socket is not closed on this path. Statement 3 creates a must alias t for s, causing $\langle\{s, t\}, e\rangle$ to be added to $D(4)$. For statement 4, $\text{MCS}\textsc{atisfied}\text{A}\textsc{fter}(\{s, t\}, \texttt{close(t)})$ holds, so no facts are propagated from 4 to *exit*.

3.6.3 Lightweight Ownership Transfer

Section 3.6.2 describes a sound accumulation-based checker for resource leaks. However, that checker often encounters false positives in cases where an `@MustCall` obligation is satisfied in another procedure via parameter passing, return values, or object fields. Consider the following code that safely closes a `Socket`:

```
void example(String myHost, int myPort) {
  Socket s = new Socket(myHost, myPort);
  closeSocket(s);
}
void closeSocket(@Owning @MustCall("close") Socket t) {
  t.close();
}
```

The `closeSocket()` routine takes ownership of the socket—that is, it takes responsibility for closing it. The checker described by section 3.6.2 would issue a false positive on this code, because it would warn when `s` goes out of scope at the end of `example()`.

This section describes a *lightweight ownership transfer* technique for reducing false positives in such cases. Programmers write annotations like `@Owning` that transfer an obligation from one expression to another. Programmer annotations *cannot* introduce any checker unsoundness; at worst, incorrect `@Owning` annotations will cause false positive warnings. Unlike an ownership type system like Rust's (see section 3.6.8), lightweight ownership transfer imposes no restrictions on what operations can be performed through an alias, and hence has a minimal impact on the programming model.

3.6.3.1 Ownership Transfer

`@Owning` is a declaration annotation, not a type qualifier; it can be written on a declaration such as a parameter, return, field, etc., but not on a type. A pseudo-assignment to an `@Owning` lvalue transfers the right-hand side's `@MustCall` obligation. More concretely, in the Consistency Checker, at a pseudo-assignment to an lvalue with an `@Owning` annotation, the right-hand side's `@MustCall` obligation is treated as satisfied.

The $\text{MCS}\textsc{atisfied}\text{A}\textsc{fter}(P, s)$ and $\text{H}\textsc{as}\text{O}\textsc{bligation}(e)$ procedures of algorithm 4 are enhanced for ownership transfer, as shown in algorithm 5.

Constructor returns are always `@Owning`. The Resource Leak Checker's default for unannotated method returns is `@Owning`, and for unannotated parameters and fields is `@NotOwning`. These assumptions coincide well with coding patterns we observed in practice, reducing the annotation burden for programmers. Further, this treatment of parameter and return types ensures sound handling of

Algorithm 5: Updated and new helper functions in algorithm 4 used by the lightweight ownership system of section 3.6.3.1.

```
procedure MCSatisfiedAfter$(P, s)$:
    return $\exists p \in P.\ \mathrm{MCAfter}(p, s) \subseteq \mathrm{CMAfter}(p, s)$
                $\lor$ ($s$ is return p $\land$ OwningReturn(CFG))
                $\lor$ PassedAsOwningParam$(s, p)$
                $\lor$ ($s$ is q.f = p $\land$ f is @Owning)
procedure HasObligation$(e)$:
    return $e$ has a declared @MustCall type and $e$'s declaration is @Owning
procedure OwningReturn(CFG):
    return CFG returns a value and CFG's return declaration is @Owning
procedure PassedAsOwningParam$(s, p)$:
    return $s$ passes $p$ to an @Owning parameter of its callee
```

unannotated third-party libraries: any object returned from such a library is tracked by default, and the checker never assumes that passing an object to an unannotated library satisfies its obligations.

3.6.3.2 Final Owning Fields

Additional class-level checking is required for `@Owning` fields, as the code satisfying their `@MustCall` obligations usually spans multiple procedures. This section handles final `@Owning` fields,[16] which cannot be overwritten after initialization of the enclosing object. When checking non-final `@Owning` fields, the checker must ensure that overwriting the field is safe (which is handled in section 3.6.5.1).

For final `@Owning` fields, our checking enforces the "resource acquisition is initialization (RAII)" programming idiom [285]. Some destructor-like method `d()` must ensure the field's `@MustCall` obligation is satisfied, and the enclosing class must have an `@MustCall("d")` obligation to ensure the destructor is called.

More formally, consider a final `@Owning` field f declared in class C, where f has type `@MustCall("m")`. To modularly verify that f's `@MustCall` obligation is satisfied, the Resource Leak Checker checks the following conditions:

1. All expressions of type C must have a type `@MustCall("d")` for some method `C.d()`.

2. `C.d()` must always invoke `this.f.m()`, thereby satisfying f's `@MustCall` obligation.

Condition 1 is checked by inspecting the `@MustCall` annotation on class C. Condition 2 is checked by requiring an appropriate `@EnsuresCalledMethods` postcondition annotation (section 3.5.3.5) on `C.d()`, which is then enforced by the Called Methods Checker.

[16] The Resource Leak Checker treats all static fields as non-owning. In our case studies, we did not observe any assignments of expressions with non-empty must-call obligations to static fields. We leave handling owning static fields to future work.

3.6.4 Resource aliasing

This section introduces a sound, lightweight, specialized must-alias analysis that tracks *resource alias* sets—sets of pointers that definitely correspond to the same underlying system resource. Closing one alias also closes the others. Thus, the Resource Leak Checker can avoid issuing false positive warnings about resources that have already been closed through a resource alias.

3.6.4.1 Wrapper Types

Java programs extensively use *wrapper types.* For example, the Java `BufferedOutputStream` wrapper adds buffering to some delegate `OutputStream`, which may or may not represent a resource that needs closing. The wrapper's `close()` method invokes `close()` on the delegate. Wrapper types introduce two additional complexities for `@MustCall` checking:

1. If a delegate has no `@MustCall` obligation, the corresponding wrapper object should also have no obligation.

2. Satisfying the obligation of *either* the wrapped object or the wrapper object is sufficient.

For example, if a `BufferedOutputStream` b wraps a stream with no underlying resource (e.g., a `ByteArrayOutputStream`), b's `@MustCall` obligation should be empty, as b has no resource of its own. By contrast, if b wraps a stream managing a resource, like a `FileOutputStream` f, then `close()` must be invoked on *either* b or f.

Previous work has shown that reasoning about wrapper types is required to avoid excessive false positive and duplicate reports [295, 104]. Wrapper types in earlier work were handled with hard-coded specifications of which library types are wrappers, and heuristic clustering to avoid duplicate reports for wrappers [295].

Our technique handles wrapper types more generally by tracking *resource aliases.* Two references r_1 and r_2 are resource aliases if r_1 and r_2 are must-aliased pointers, or if satisfying r_1's `@MustCall` obligation also satisfies r_2's obligation and vice-versa.

Introducing resource aliases To indicate where an API method creates a resource-alias relationship between distinct objects, the programmer writes a pair of `@MustCallAlias` qualifiers: one on a parameter of a method, and another on its return type. For example, one constructor of `BufferedOutputStream` is:

```
@MustCallAlias BufferedOutputStream(@MustCallAlias OutputStream arg0);
```

`@MustCallAlias` annotations are verified, not trusted.

At a call site to an `@MustCallAlias` method, there are two effects. First, the must-call type of the method call's return value is the same as that of the `@MustCallAlias` argument. If the type of the argument has no must-call obligations (like a `ByteArrayOutputStream`), the returned wrapper has no must-call obligations.

Second,the Consistency Checker treats the `@MustCallAlias` parameter and return as aliases. For the pseudocode in algorithm 2, the updated version of CreatesAlias from algorithm 4 in algorithm 6 handles resource aliases.

Algorithm 6: Updated version of the CreatesAlias helper function from algorithm 4 used with resource aliasing. Note that the first clause of the disjunction in CreatesAlias is the original definition from algorithm 4.

```
procedure CreatesAlias(P, s):
    return $\exists q \in P$ . $s$ is of the form p = q $\vee$ IsMustCallAliasParam($s, q$)
procedure IsMustCallAliasParam(s, p):
    return $s$ passes $p$ to an @MustCallAlias parameter of its callee
```

3.6.4.2 Beyond Wrapper Types

`@MustCallAlias` can also be employed in scenarios beyond direct wrapper types, a capability not present in previous work on resource leak detection. In certain cases, a resource gets shared between objects via an intermediate object that cannot directly close the resource. For example, `java.io.RandomAccessFile` (which must be closed) has a method `getFd()` that returns a `FileDescriptor` object for the file. This file descriptor cannot be closed directly—it has no `close()` method. However, the descriptor can be passed to a wrapper stream such as `FileOutputStream`, which if closed satisfies the original must-call obligation. By adding `@MustCallAlias` annotations to the `getFd()` method, our technique can verify code like the below (adapted from Apache Hadoop [291]):

```
RandomAccessFile file = new RandomAccessFile(myFile, "rws");
FileInputStream in = null;
try {
  in = new FileInputStream(file.getFD());
  // do something with in
  in.close();
} catch (IOException e){
  file.close();
}
```

Because the must-call obligation checker treats `@MustCallAlias` annotations polymorphically, regardless of the associated base type, the Resource Leak Checker can verify that the same resource is held by the `RandomAccessFile` and the `FileInputStream`, even though it is passed via a class without a `close()` method.

3.6.4.3 Verification of `@MustCallAlias`

A pair of `@MustCallAlias` annotations on a method or constructor `m`'s return type and its parameter `p` can be verified if either of the following holds:

1. `m`'s body passes `p` to another method or constructor in an `@MustCallAlias` position, and `m` returns that method's result, or the call is a `super()` constructor call annotated with `@MustCallAlias`.

2. `m`'s body stores `p` in an `@Owning` field of the class enclosing `m`.

The intuition for condition 1 is that `m`'s result is a resource alias with the parameter `p`—that is, that `m` wraps `p`. The intuition for condition 2 is that `m` creates a wrapper relationship between its receiver and `p`.

This verification procedure permits a programmer to soundly specify a resource-aliasing relationship in their own code, unlike prior work that relied on a hard-coded list of wrapper types.

3.6.5 Creating new obligations

Every constructor of a class that has must-call obligations implicitly creates obligations for the newly-created object. However, non-constructor methods may also create obligations when re-assigning non-final owning fields or allocating new system-level resources. To handle such cases soundly, we introduce a method post-condition annotation, `@CreatesMustCallFor`, to indicate expressions for which an obligation is created at a call.

At each call-site of a method annotated as `@CreatesMustCallFor(`*expr*`)`, the Resource Leak Checker removes any inferred Called Methods information about *expr*, reverting to `@CalledMethods({})`.

When checking a call to a method annotated as `@CreatesMustCallFor(`*expr*`)`, the Consistency Checker (1) treats the `@MustCall` obligation of *expr* as *satisfied*, and (2) creates a fresh obligation to check. These changes require updates to the FactsFromCall and MCSatisfiedAfter procedures of algorithm 4, which appear in algorithm 7.

Algorithm 7: Updated versions of helper functions from algorithm 4 to support creating new obligations. [. . .] stands for the cases shown previously, including those in section 3.6.3.1.

```
procedure FactsFromCall(s):
    $p \leftarrow s.LHS, c \leftarrow s.RHS$
    return $\{\langle\{p_i\}, c\rangle \mid p_i \in \text{CMCFTargets}(c)\}$
                $\cup\ (\text{HasObligation}(c)\ ?\ \{\langle\{p\}, c\rangle\} : \emptyset)$
procedure MCSatisfiedAfter(P, s):
    return $\exists p \in P.\ [\ldots] \vee p \in \text{CMCFTargets}(s)$
procedure CMCFTargets(c):
    return $\{\ p_i \mid p_i$ passed to an @CreatesMustCallFor target for $c$'s callee $\}$
```

These changes are sound: the checker creates a new obligation for calls to `@CreatesMustCallFor` methods, and the must-call obligation checker ensures the `@MustCall` type for the target will have a *superset* of any methods present before the call. There is an exception to this check: if an `@CreatesMustCallFor` method is invoked within a method that has an `@CreatesMustCallFor` annotation with the same target—imposing the obligation on its caller—then the new obligation can be treated as satisfied immediately.

3.6.5.1 Non-Final, Owning Fields

`@CreatesMustCallFor` allows the Resource Leak Checker to verify uses of non-final fields that contain a resource, even if they are re-assigned. Consider the following example:

```
@MustCall("close") // sets default qual. for uses of SocketContainer
class SocketContainer {

  private @Owning Socket sock;

  public SocketContainer() { sock = ...; }

  void close() { sock.close() };

  @CreatesMustCallFor("this")
  void reconnect() {
    if (!sock.isClosed()) {
      sock.close();
    }
    sock = ...;
  }
}
```

In the lifetime of a `SocketContainer` object, `sock` might be re-assigned arbitrarily many times: once at each call to `reconnect()`. This code is safe, however: `reconnect()` ensures that `sock` is closed before re-assigning it.

The Resource Leak Checker must enforce two new rules to ensure that re-assignments to non-final, owning fields like `sock` in the example above are sound:

- any method that re-assigns a non-final, owning field of an object must be annotated with an `@CreatesMustCallFor` annotation that targets that object.
- when a non-final, owning field f is re-assigned at statement s, its inferred `@MustCall` obligation must be contained in its `@CalledMethods` type at the program point before s.

The first rule ensures that `close()` is called after the last call to `reconnect()`, and the second rule ensures that `reconnect()` safely closes `sock` before re-assigning it. Because calling an `@CreatesMustCallFor` method like `reconnect()` resets local type inference for called methods, calls to `close` before the last call to `reconnect()` are disregarded.

3.6.5.2 Unconnected Sockets

`@CreatesMustCallFor` can also handle cases where object creation does not allocate a resource, but the object will allocate a resource later in its lifecycle. Consider the no-argument constructor to `java.net.Socket`. This constructor does not allocate an operating system-level socket, but instead just creates the container object, which permits the programmer to e.g. set options which will be used when creating the physical socket. When such a `Socket` is created, it initially has no must-call obligation; it is only when the `Socket` is actually connected via a call to a method such as `bind()` or `connect()` that the must-call obligation is created.

If all `Socket`s are treated as `@MustCall({"close"})`, a false positive would be reported in code such as the below, which operates on an unconnected socket (simplified from real code in Apache Zookeeper [292]):

```
static Socket createSocket() {
  Socket sock = new Socket();
  sock.setSoTimeout(...);
  return sock;
}
```

The call to `setSoTimeout` can throw a `SocketException` if the socket is actually connected when it is called. Using `@CreatesMustCallFor`, however, the Resource Leak Checker can soundly show that this socket is not connected: the type of the result of the no-argument constructor is `@MustCall({})`, and `@CreatesMustCallFor` annotations on the methods that actually allocate the socket—`connect()` or `bind()`—enforce that as soon as the socket is open, it is treated as `@MustCall("close")`.

3.6.6 Evaluating the Resource Leak Checker

Our evaluation of the Resource Leak Checker has three parts:

- case studies on open-source projects, which show that our approach is scalable and finds real resource leaks.
- an evaluation of the importance of lightweight ownership, resource aliasing, and obligation creation.
- a comparison to previous leak detectors: both a heuristic bug finder and a whole-program analysis.

All code and data for our experiments described in this section, including the version of the Resource Leak Checker's implementation used in the experiments, experimental machinery, and annotated versions of our case study programs, are publicly available at `https://doi.org/10.5281/zenodo.4902321`. The Resource Leak Checker is distributed and maintained as part of the Checker Framework [90].

3.6.6.1 Case Studies on Open-Source Projects

We selected 3 open-source projects that were analyzed by prior work [328]. For each, we selected and analyzed one or two modules with many uses of leak-able resources. We used the latest version of the source code that was available when we began. We also analyzed an open-source project maintained by one of the Resource Leak Checker's authors, to simulate the Resource Leak Checker's expected use case, where the user is already familiar with the code under analysis (see section 3.6.6.4).

For each case study, our methodology was as follows. (1) We modified the build system to run the Resource Leak Checker on the module(s), analyzing uses of resource classes that are defined in the JDK. It also reports the maximum possible number of resources (references to JDK-defined classes

Table 3.4: Verifying the absence of resource leaks. "LoC" is lines of non-comment, non-blank Java code. "Rs" is the number of resources created by the program. "RLs" are true positive warnings. "FPs" are false positives, where the tool reported a potential leak, but manual analysis revealed that no leak is possible. "As" is the number of manually-written annotations. "CC" is the number of code changes: edits to program text, excluding annotations. "WCT" is wall-clock time, computed as the median of five trials.

	LoC	Rs	RLs	FPs	As	CC	WCT
zookeeper:zookeeper-server	45,248	177	13	48	122	5	1m 24s
hadoop:hadoop-hdfs-project/hadoop-hdfs	151,595	365	23	49	117	13	16m 21s
hbase:hbase-server,hbase-client	220,828	55	5	22	45	5	7m 45s
plume-util	10,187	109	8	2	2	19	0m 15s
Total	427,858	706	49	121	286	42	-

with a non-empty `@MustCall` obligation) that could be leaked: each obligation at a formal parameter or method call. (2) We manually annotated each program with must-call, called-methods, and ownership annotations. (3) We iteratively ran the analysis to correct our annotations. We measured the run time as the median of 5 trials on a machine running Ubuntu 20.04 with an Intel Core i7-10700 CPU running at 2.90GHz and 64GiB of RAM. Our analysis is embarrassingly parallel, but our implementation is single-threaded because javac is single-threaded. (4) We manually categorized each warning as revealing a real resource leak (a true positive) or as a false positive warning about safe code that our tool is unable to prove correct.

Table 3.4 summarizes the results. The Resource Leak Checker found multiple serious resource leaks in every program. The Resource Leak Checker's overall precision on these case studies is 29% (49/170). Though there are more false positives than true positives, the number is small enough to be examined by a single developer in a few hours. The annotations in the program are also a benefit: they express the programmer's intent and, as machine-checked documentation, they cannot become out-of-date.

3.6.6.2 True and False Positive Examples

This section gives examples of warnings reported by the Resource Leak Checker in the case study programs.

Figure 3.24 contains code from Hadoop. If an IO error occurs any time between the allocation of the `FileInputStream` in the first line of the method and the `return` statement at the end—for example, if `channel.position(section.getOffset())` throws an `IOException`, as it is specified to do—then the only reference to the stream is lost. Hadoop's developers assigned this issue a priority of "Major" and accepted our patch [272]. One developer suggested using a try-with-resources statement instead of our patch (which catches the exception and closes the stream), but we pointed out that the file needs to remain open if no error occurs so that it can be returned.

The most common reason for false positives (which caused 22% of the false positives in our case studies) was a known bug in the Checker Framework's type inference algorithm for Java generics,

```
public InputStream getInputStreamForSection(
    FileSummary.Section section, String compressionCodec)
    throws IOException {
  FileInputStream fin = new FileInputStream(filename);
  FileChannel channel = fin.getChannel();
  channel.position(section.getOffset());
  InputStream in = new BufferedInputStream(new LimitInputStream(fin,
      section.getLength()));
  in = FSImageUtil.wrapInputStreamForCompression(conf, compressionCodec, in);
  return in;
}
```

Figure 3.24: A resource leak that the Resource Leak Checker found in Hadoop. Hadoop's developers merged our fix [272].

```
Optional<ServerSocket> createServerSocket(...) {
  ServerSocket serverSocket;
  try {
    if (...) {
       serverSocket = new ServerSocket();
       serverSocket.setReuseAddress(true);
       serverSocket.bind(...);
       return Optional.of(serverSocket);
    }
  } catch (IOException e) {
    // log an error
  }
  return Optional.empty();
}
```

Figure 3.25: Code from the ZooKeeper case study that causes the Resource Leak Checker to issue a false positive.

Table 3.5: The annotations we wrote in the case studies of the Resource Leak Checker.

Annotation	Count
`@Owning` and `@NotOwning`	98
`@EnsuresCalledMethods`	54
`@MustCall`	53
`@MustCallAlias`	41
`@CreatesMustCallFor`	40
Total	286

which the Checker Framework developers are working to fix [217]. The second most common reason (causing 15%) was a generic container object like `java.util.Optional` taking ownership of a resource, such as the example in fig. 3.25. Our lightweight ownership system does not support transferring ownership to generic parameters, so the Resource Leak Checker issues an error when `Optional.of` is returned. In this case, the use of the `Optional` class is unnecessary and complicates the code [111]. If `Optional` was replaced by a nullable Java reference, the Resource Leak Checker could verify this code. Future work should expand the lightweight ownership system to support Java generics. The third most common reason (causing 8%) is nullness reasoning: some resource is closed only if it is non-null, but our checker expects the resource to be closed on every path. Our checker handles simple comparisons with `null` (as in fig. 3.21), but future work could incorporate more complex nullness reasoning [237].

We also evaluated the degree to which an "ideal" alias analysis would improve the precision of the Resource Leak Checker. Our goal in doing so was to answer the question "how much of the remaining imprecision in the Resource Leak Checker is due to our choice of an accumulation analysis, which avoids a whole-program alias analysis?"

To answer this question, we examined each false positive issued by the Resource Leak Checker by hand and determined whether an analysis with access to an aliasing oracle could conclude that the code is safe. We found that 39 of the 121 (32%) false positive warnings issued by the Resource Leak Checker would be verifiable with access to an aliasing oracle; if these warnings were not issued, the precision of the Resource Leak Checker on our benchmarks would be 37% (an improvement of only 11%). We therefore conclude that lack of aliasing information is not the primary reason for the Resource Leak Checker's remaining imprecision: proving the absence of resource leaks is a challenging problem even with perfect aliasing information. We believe this result helps justify our choice of an accumulation analysis for this problem.

3.6.6.3 Annotations and Code Changes

We wrote about one annotation per 1,500 lines of code (table 3.5).

We also made 42 small, semantics-preserving changes to the programs to reduce false positives from our analysis. In 19 places in plume-util, we added an explicit `extends` bound to a generic type. The Checker Framework uses different defaulting rules for implicit and explicit upper bounds, and a common pattern in this benchmark caused our checker to issue an error on uses of implicit bounds.

Table 3.6: False positives in our case studies ("RLC") and without lightweight ownership ("w/o LO"), resource aliasing ("w/o RA"), and obligation creation ("w/o OC").

Project	w/o LO	w/o RA	w/o OC	RLC
apache/zookeeper	117	158	47	48
apache/hadoop	97	184	58	49
apache/hbase	82	93	26	22
plume-lib/plume-util	4	11	2	2
Total	300	446	133	121

In 18 places, we made a field `final`; this allows our checker to verify usage of the field without using the stricter rules for non-final owning fields given in section 3.6.5. In 9 of those cases, we also removed assignments of `null` to the field after it was closed; in 1 other we added an `else` clause in the constructor that assigned the field a `null` value. In 3 places, we re-ordered two statements to remove an infeasible control-flow-graph edge. In 2 places, we extracted an expression into a local variable, permitting flow-sensitive reasoning or targeting by an `@CreatesMustCallFor` annotation.

3.6.6.4 Simulating the User Experience

To simulate the experience of a typical user who understands the codebase being analyzed, my collaborator Michael Ernst used the Resource Leak Checker to analyze plume-util, a 10KLoC library he wrote 23 years ago. The process took about two hours, including running the tool, writing annotations, and fixing the 8 resource leaks that the tool discovered. The annotations were valuable enough that they are now committed to that codebase, and the Resource Leak Checker runs in CI to prevent the introduction of new resource leaks. This example is suggestive that the programmer effort to use our tool is reasonable.

3.6.6.5 Evaluating Our Enhancements

Lightweight ownership (section 3.6.3), resource aliasing (section 3.6.4), and obligation creation (section 3.6.5) reduce false positive warnings and improve the Resource Leak Checker's precision. To evaluate the contribution of each enhancement, we individually disabled each feature and re-ran the experiments whose results are reported in table 3.4.

Table 3.6 shows that each of lightweight ownership and resource aliases prevents more false positive warnings than the total number of remaining false positives on each benchmark. The system for creating new obligations at points other than constructors reduces false positives by a smaller amount: non-final, owning field re-assignments are rare.

3.6.6.6 Comparison to Other Tools

Our approach represents a novel point in the design space of resource leak checkers. This section compares our approach with two other modern tools that detect resource leaks:

Table 3.7: Comparison of resource leak checking tools: Eclipse, Grapple, and the Resource Leak Checker (RLC). Recall is the ratio of reported leaks to all leaks present in the code, and precision is the ratio of true positive warnings to all tool warnings. Different tools were run on different versions of the case study programs. The number of leaks and the recall are computed over the code that is common to all versions of the programs, so recall is directly comparable within rows. Precision is computed over the code version analyzed by each tool, so it may not be directly comparable within rows. Eclipse reports no high-confidence warnings for JDK types in HBase.

	Recall				**Precision**[17]		
Project	leaks	Eclipse	Grapple	RLC	Eclipse	Grapple	RLC
ZooKeeper	6	17%	17%	100%	33%	67%	21%
HDFS	7	14%	0%	100%	20%	71%	32%
HBase	2	0%	0%	100%	-	35%	19%
Total	15	13%	7%	100%	25%	50%	26%

- The analysis built into the Eclipse Compiler for Java (ecj), which is the default approach for detecting resource leaks in the Eclipse IDE [104]. We used version 4.18.0.
- Grapple [328], a state-of-the-art typestate checker that leverages whole-program alias analysis.

In brief, both of the above tools are unsound and missed 87–93% of leaks. Both tools neither require nor permit user-written specifications, a plus in terms of ease of use but a minus in terms of documentation and flexibility. Eclipse is very fast (nearly instantaneous) but has low precision (25% for high-confidence warnings, *much* lower if all warnings are included). Grapple is more precise (50% precision), but an order of magnitude slower than the Resource Leak Checker. The Resource Leak Checker had 100% recall and 26% precision. Users can select whichever tool matches their priorities.

Tables 3.7 and 3.8 quantitatively compare the tools. The table uses parts of the 3 case study programs that Grapple was run on in the past, because we were unable to run Grapple on the (more recent) versions of the case study programs used in our own experiments. Details follow in the section on Grapple.

Eclipse The Eclipse analysis is a simple dataflow analysis augmented with heuristics. Since it is tightly integrated with the compiler, it scales well and runs quickly. It has heuristics for ownership, resource wrappers, and resource-free closeables, among others; these are all hard-coded into the analysis and cannot be adjusted by the user. It supports two levels of analysis: detecting high-confidence resource leaks and detecting "potential" resource leaks (a superset of high-confidence resource leaks).

We ran Eclipse's analysis on the exact same code that we ran the Resource Leak Checker on for table 3.4 (excluding the plume-util case study). Table 3.7 reports results for a subset of the code;

[17] Not directly comparable within rows.

this paragraph reports results for the full code. In “high-confidence” mode on the three projects, Eclipse reports 8 warnings related to classes defined in the JDK: 2 true positives (thus, it misses 39 real resource leaks) and 6 false positives. In “potential” leak mode, the analysis reports many more warnings. Thus, we triaged only the 180 warnings about JDK classes from the ZooKeeper benchmark. Among these were 3 true positives (it missed 10 real resource leaks) and 177 false positives (2% precision). The most common cause of false positives was the unchangeable, default ownership transfer assumption at method returns, leading to a warning at each call that returns a resource-alias, such as `Socket#getInputStream`.

Grapple Grapple is a modern typestate-based resource leak analysis “designed to conduct precise and scalable checking of finite-state properties for very large codebases” [328]. Grapple models its alias and dataflow analyses as dynamic transitive-closure computations over graphs, and it leverages novel path encodings and techniques from predecessor-system Graspan [305] to achieve both context- and path-sensitivity. Grapple contains four checkers, of which two can detect resource leaks. Unlike the Resource Leak Checker, Grapple is unsound, as it performs a fixed bounded unrolling of loops to make path sensitivity tractable. The Resource Leak Checker reports violations of a user-supplied specification (which takes effort to write but provides documentation benefits), so it can ensure that a library is correct for all possible clients. By contrast, Grapple checks a library in the context of one specific client; it only reports issues in methods reachable from entry points (like a `main()` method) in a whole-program call graph [327].

The Grapple authors evaluated their tool on earlier versions of the first three case study programs in section 3.6.6.1 [328]. Unfortunately, a direct comparison on our benchmark versions is not possible, because Grapple’s leak detector currently cannot be run (by us or by the Grapple authors) due to library incompatibilities and bitrot in the implementation. The Grapple authors provided us with the finite-state machine (FSM) specifications used in Grapple to detect resource leaks, and also details of all warnings issued by Grapple in the versions of the benchmarks they analyzed.

We used the following methodology to permit a head-to-head comparison. We started with all warnings issued by either tool. We disregarded any warning about code that is not present identically in the other version of the target program (due to refactoring, added code, bug fixes, etc.). We also disregarded warnings about code that is not checked by both tools. For example, Grapple analyzed test code, but in our experiments we did not write annotations in test code nor type-check it. The remaining warnings pertain to resource leaks in identical code that both tools ought to report. For each remaining warning, we manually identified it as a true positive (a real resource leak) or a false positive (correct code, but the tool cannot determine that fact). Table 3.7 reports the precision and recall of Eclipse, Grapple, and the Resource Leak Checker. Some of Grapple’s false positives are reports about types like `java.io.StringWriter` with no underlying resource that must be closed. (These reports were mis-classified as true positives in [328], which is one reason the numbers there differ from table 3.7.) Grapple’s false negatives might be due to analysis unsoundness or gaps in API modeling (e.g., Grapple does not include FSM specifications for `OutputStream` classes).

Grapple runs can take many hours (run times are from [328]), whereas the Resource Leak Checker runs in minutes (table 3.8). Further, Grapple is not modular, so if the user edits their program, Grapple must be re-run from scratch [327]. After a code edit, the Resource Leak Checker only needs to re-analyze modified code (and possibly its dependents if the modified code’s interface changed).

Table 3.8: Run times of resource leak checking tools.

Project	Eclipse	Grapple	Resource Leak Checker
ZooKeeper	<5s	1h 07m 02s	1m 24s
HDFS	<5s	1h 54m 52s	16m 21s
HBase	<5s	33h 51m 59s	7m 45s

3.6.7 Limitations and Threats to Validity

Like any tool that analyzes source code, the Resource Leak Checker only gives guarantees for code that it checks: the guarantee excludes native code, the implementation of unchecked libraries (such as the JDK), and code generated dynamically or by other annotation processors such as Lombok. Though the Checker Framework can handle reflection soundly [25], by default (and in our case studies) the Resource Leak Checker compromises this guarantee by assuming that objects returned by reflective invocations do not carry must-call obligations. (Users can customize this behavior.) Within the bounds of a user-written warning suppression, the Resource Leak Checker assumes that 1) any errors issued can be ignored, and 2) all annotations written by the programmer are correct.

The Resource Leak Checker is sound with respect to specifications of which types have a `@MustCall` obligation that must be satisfied. We wrote such specifications for the Java standard library, focusing on IO-related code in the `java.io` and `java.nio` packages. Any missing specifications of `@MustCall` obligations could lead the Resource Leak Checker to miss resource leaks.

The results of our experiments may not generalize, compromising the external validity of the experimental results. The Resource Leak Checker may produce more false positives, require more annotations, or be more difficult to use if applied to other programs. Case studies on legacy code represents a worst case for a source code analysis tool. Using the Resource Leak Checker from the inception of a project would be easier, since programmers know their intent as they write code and annotations could be written along with the code. It would also be more useful, since it would guide the programmers to a better design that requires fewer annotations and has no resource leaks. The need for annotations could be viewed as a limitation of our approach. However, the annotations serve as concise documentation of properties relevant to resource leaks—and unlike traditional, natural-language documentation, machine-checked annotations cannot become out-of-date.

Like any practical system, it is possible that there might be defects in the implementation of the Resource Leak Checker or in the design of its analyses. We have mitigated this threat with code review and an extensive test suite: 119 test classes containing 3,776 lines of non-comment, non-blank code. This test suite is publicly available and distributed with the Resource Leak Checker.

3.6.8 Related Work: Resource Leaks

Most prior work on resource leak detection either uses program analysis to detect leaks or adds language features to prevent them. Here we focus on the most relevant work from these categories.

3.6.8.1 Analysis-Based Approaches

Static analysis Tracker [295] performs inter-procedural dataflow analysis to detect resource leaks, with various additional features to make the tool practical, including issue prioritization and handling of wrapper types. Tracker avoids whole-program alias analysis to improve scalability, instead using a local, access-path-based approach. While Tracker scales to large programs, it is deliberately unsound, unlike the Resource Leak Checker.

The Eclipse Compiler for Java includes a dataflow-based bug-finder for resource leaks [104]. Its analysis uses a fixed set of ownership heuristics and a fixed list of wrapper classes; unlike the Resource Leak Checker, it is unsound. It is very fast. Similar analyses—with similar trade-offs compared to the Resource Leak Checker—exist in other heuristic bug-finding tools, including SpotBugs [279], PMD [247], and Infer [164]. Section 3.6.6.6 experimentally evaluates the Eclipse analysis.

Relda and Relda2 [151, 314] are unsound resource-leak detection approaches that are specialized to the Android framework with call graphs that model the framework's use of callbacks for releasing resources.

Typestate analysis [284, 124] can be used to find resource leaks. Grapple [328] is the most recent system to use this approach, leveraging a disk-based graph engine to achieve unprecedented scalability on a single machine. Compared to the Resource Leak Checker, Grapple is more precise but suffers from unsoundness and longer run times. Section 3.6.6.6 gives a more detailed comparison to Grapple.

The CLOSER [97] automatically inserts Java code to dispose of resources when they are no longer "live" according to its dataflow analysis. Their approach requires an expensive alias analysis for soundness, as well as manually-provided aliasing specifications for linked libraries. the Resource Leak Checker uses accumulation analysis to achieve soundness without the need for a whole-program alias analysis.

Dynamic analysis Some approaches use dynamic analysis to ameliorate leaks. Resco [74] operates similarly to a garbage collector, tracking resources whose program elements have become unreachable. When a given resource (such as file descriptors) is close to exhaustion, the runtime runs Resco to clean up any resources of that type that are unreachable. With a static approach such as ours, leaks are impossible and a tool like Resco is unnecessary.

Automated test generation can also be used to detect resource leaks. For example, leaks in Android applications can be found by repeatedly running neutral—i.e. eventually returning to the same state—GUI actions [312, 323]. Other techniques detect common misuse of the Android activity lifecycle [12]. Testing can only show the presence of failures, not the absence of defects; the Resource Leak Checker verifies that no resource leaks are present.

Data sets and surveys The DroidLeaks benchmark [205] is a set of Android apps with known resource leaks. Unfortunately, it includes only the compiled apps. The Resource Leak Checker runs on source code, so we were unable to run the Resource Leak Checker on DroidLeaks. Ghanavati et al. [141] performed a detailed study of resource leaks and their repairs in Java projects, showing the pressing need for better tooling for resource leak prevention. In particular, their study showed that developers consider resource leaks to be an important problem, and that previous static analysis tools are insufficient for preventing resource leaks.

3.6.8.2 Language-Based Approaches

Ownership types and Rust Ownership type systems [69] impose control over aliasing, which in turn enables guaranteeing other properties, like the absence of resource leaks. We do not discuss the vast literature on ownership type systems [69] here. Instead, we focus on ownership types in Rust [183] as the most popular practical example of using ownership to prevent resource leaks.

For a detailed overview of ownership in Rust, see chapter 4 of [183]; we give a brief overview here. In Rust, ownership is used to manage both memory and other resources. Every value associated with a resource must have a *unique* owning pointer, and when an owning pointer's lifetime ends, the value is "dropped," ensuring all resources are freed. Rust's ownership type system statically prevents not only resource leaks, but also other important issues like "double-free" defects (releasing a resource more than once) and "use-after-free" defects (using a resource after it has been released). But, this power comes with a cost; to enforce uniqueness, non-owning pointers must be invalidated after an ownership transfer and can no longer be used. Maintaining multiple usable pointers to a value requires use of language features like references and borrowing, and even then, borrowed pointers have restricted privileges.

The Resource Leak Checker has less power than Rust's ownership types; it cannot prevent double-free or use-after-free defects. But, the Resource Leak Checker's lightweight ownership annotations impose *no* restrictions on aliasing; they simply aid the tool in identifying how a resource will be closed. Lightweight ownership is better suited to preventing resource leaks in existing, large Java code bases; adapting such programs to use a full Rust-style ownership type system would be impractical.

Other approaches Java's try-with-resources construct [233] was discussed in section 3.6.1. Java also provides finalizer methods [148, Chapter 12], which execute before an object is garbage-collected, but they should not be used for resource management, as their execution may be delayed arbitrarily.

Compensation stacks [308] generalize C++ destructors and Java's try-with-resources, to avoid resource leak problems in Java. While compensation stacks make resource leaks less likely, they do not guarantee that leaks will not occur, unlike the Resource Leak Checker.

3.7 A Practical Accumulation Analysis for NoSQL Databases

NoSQL databases [140] support a variety of query languages, some of which are unique to the database in question. This section focuses on Amazon's DynamoDB key-value store [80] and a set of errors that clients can make when querying it. Other databases have similar (but not identical) weaknesses in their query languages; the description of the problem and solution in this section are one example of how we can develop specialized typecheckers to handle such queries.

As a running example, consider the following query to DynamoDB:

```
Map<String, AttributeValue> keys = ImmutableMap.of(
    ":account_id", account_id,
    ":type", type
);
QueryRequest request = QueryRequest.builder()
    .tableName(this.perimetersTableName)
    .keyConditionExpression(" = :account_id")
    .expressionAttributeValues(keys)
    .expressionAttributeNames(ImmutableMap.of("#type", "type"))
    .filterExpression("#type = :type")
    .build();
```

Concluding that this query is safe requires three new accumulation analyses, in addition to the enhanced constant value analysis described later in section 5.4.2. Ultimately, it must conclude that:

- the required expression attribute name is "#type".
- the required expression attribute values are ":account_id" and ":type".
- the provided expression attribute name is "#type".
- the provided expression attribute values are ":account_id" and ":type".
- the required and provided expression attribute names match.
- the required and provided expression attribute values match.

The reason for these requirements is that when a user queryies DynamoDB, they specify a *key condition expression*, a string that determines the items to be read from the table. The key condition expression is made up of a combination of *expression attribute names*, *expression attribute values*, constants, and operators. Expression attribute names and values are placeholders in the key condition expression that will be substituted at run time. For example, suppose a client has a database table containing information about books. At run time, the client solicits the title of a particular book from its user, and then queries the database to look up all the information about that book. In the key condition expression, the client's source code would have to represent the title of the book that is to be looked up as an expression attribute value, and then provide the actual value to the query by populating a map at run time—in effect, the key condition expression represents *all possible* run time queries, and a specific query is chosen at run time. Expression attribute names are used in more specific contexts (for example, when the name of a key in the database conflicts with a reserved word), but are specified identically.

If a client supplies a key condition expression that uses a set of expression attribute values (or, equivalently, names) K, then the clientX must provide a corresponding map $M : K \to V$ from the elements of K to their run-time values. If M does not define at least all the elements of K, then the DynamoDB query will return incorrect results or fail with an error. Either outcome is undesirable. Worse, key condition expressions are not evaluated by common DynamoDB mocking tools, so these sort of errors are typically only caught once the client is deployed against a live DynamoDB table, leading to failures in production.

DynamoDB also supports a *filter expression* syntax that uses the same expression attribute names and values to filter the results of a query post-hoc. Malformed filter expressions cause similar problems to malformed key condition expressions.

3.7.0.1 Accumulation analyses for required names and values

DynamoDB differentiates expression attribute names from expression attribute values by the first character of each: names must start with "#" and values must start with ":". We can therefore define simple accumulation analyses for required names and values. When the request builder is created in the example above, it has the type `@RequiredNames() @RequiredValues() QueryRequest.Builder`. At each call to `keyConditionExpression` or `filterExpression`, the type qualifiers accumulate the names (respectively, values) used in the expression, if it is compile-time constant. If the expression is not a compile-time constant, then a special type qualifier, `@UnknownRequirements`, is substituted, which will lead to an error.

For example, when `keyConditionExpression` is called above, the type of the builder is updated to `@RequiredValues(":account_id")`. (Note that the constant propagation analysis is aware that `String.format` is polymorphic in its argument.) This accumulation analysis needs the same returns-receiver alias analysis described in section 3.5.3.3 due to the use of builders.

3.7.0.2 Accumulation analyses for provided names and values

The accumulation analyses for provided names and values work similarly to the analyses for required names and values described in the previous section. However, instead of using known constant values of a string as the source of what is to be accumulated, these analyses use the list of constant strings that are known to definitely be keys for the maps passed to `expressionAttributeNames` and `expressionAttributeValues`. These values are computed by the accumulation analysis described in the next section.

3.7.0.3 An accumulation analysis for map keys

This accumulation analysis determines the set of string values that are definitely keys for a given map, from calls to `Map.put`, constructors, or other sources. For example, in the definition of `keys` in the example above, the analysis determines that the type of `keys` is `@ConstantKeys(":account_id", ":type") Map`. A subsequent call to `Map.put`, if it existed, would refine the type further. This accumulation analysis needs no alias analysis, in our experience so far.

3.7.0.4 Checking that the required and provided names and values match

Finally, the type systems must cooperate to determine that a particular call to `build()` is correct. At this point, the required and provided names and values are compared. If the required names or values are `@UnknownRequirements`, then the check automatically fails—otherwise, the analysis would be unsound. If the required names and values are known, then the provided names and values must be supersets. If so, the check passes.

Together, these analyses prevent malformed queries. We have built an unsound prototype of these analyses. The prototype is unsound because it does not include `@UnknownRequirements`, and treats calls with unknown requirements as valid. We tested the prototype at AWS on proprietary code. Though the results were promising, we cannot share the results of that work. However, AWS has open-sourced the prototype [175].

3.8 Related Work: Accumulation

3.8.1 Heap Monotonic Typestates

Heap-monotonic typestates [119] are a class of typestate that, like accumulation typestate systems, do not require aliasing information for soundness. A heap monotonic typestate system is one in which the statically observable invariants of the relevant type become monotonically stronger as an object transitions through its typestates. Every heap-monotonic typestate system is an accumulation typestate system.

The present work goes further than the work on heap-monotonic typestates in three important ways. First, we have shown exactly which typestate systems (the accumulation typestate systems) can be checked without aliasing; heap-monotonic typestate systems were proven to be sound without aliasing information, but not proven to encompass all typestate systems that can be soundly checked without aliasing. Second, we have surveyed the literature to locate examples of typestate systems that can be checked soundly without aliasing; the paper on heap-monotonic typestates gives a few examples, but no procedure for discovering more. Third, we have implemented practical accumulation analyses: the prior work on heap-monotonic typestates was, to the best of our knowledge, entirely theoretical.

3.8.2 Other Categories of Typestate Systems

Others have identified interesting sub-categories of typestate systems that may be amenable to different kinds of analysis. While as far as we are aware we are the first to identify the accumulation typestate systems, the omission-closed typestate systems [123] are a close relative: an omission-closed typestate system is one in which every subsequence of every valid (i.e., not ending in the `error` state) path is also a valid path. In other words, omission-closed properties are those whose *valid* paths are closed under subsequence. By contrast, accumulation typestate systems are those whose *error-inducing* paths are closed under subsequence, if the last error-inducing transition is held constant. Unlike accumulation typestate systems, not all omission-closed typestate systems can be checked soundly without aliasing: for example, the typestate system for a `File` object whose FSM is defined by the regular expression "read*;close" is omission-closed, but cannot be checked soundly without aliasing information, because it is an error to call "close" more than once—or, put another way, "close" disables itself.

3.8.3 Typestate Surveys

Section 3.3.2.2 describes two previous papers that report on large quantities of typestate specifications [28, 101]. We have extended their work by surveying 101 papers that neither of those works considered and locating all typestates within them, and by identifying which typestate systems are accumulation typestate systems.

3.8.4 Practical Typestate Analyses

There have been many attempts to improve the scalability of typestate analyses. We mention only some of the most recent here. Rapid [108] is a modern typestate analysis built at AWS. Rapid's scalability is a design choice: it is intentionally unsound and therefore scales by not tracking all aliasing. Another recent example is Grapple [328], which uses a novel graph-reachability algorithm and a modern alias analysis together. Some of Grapple's optimizations make it unsound despite access to aliasing information. Because Grapple does track aliasing, it scales much more poorly than accumulation-based systems: for example, Grapple is more than an order of magnitude slower than our accumulation-based approach to resource-leak detection.

3.8.5 Typestate With Aliasing Restrictions

Another method to avoid the need to do an expensive whole-program alias analysis is to limit the programmer's use of aliasing. Examples include linear or affine type systems [81, 296], role analysis [189], ownership types [69, 294], and access permissions [35]. Accumulation analyses, unlike all of these approaches, do not impose any restrictions on the programming model.

3.8.6 Other Work on Typestate

Typestate is well-studied in the scientific literature, and this section is not intended as a full survey. However, the artifact for our literature survey [179] mentions all the papers that we examined (section 3.3).

3.9 Conclusions: Accumulation

Accumulation analysis is a promising technique for practical lightweight program verification for the subset of typestate problems for which accumulation is applicable. We have proved exactly which typestate problems those are, demonstrated that they are common (comprising 41% of the specifications we found in our literature survey), and built practical accumulation analyses for important problems—initialization and resource leaks—that outperform the state-of-the-art.

Chapter 4

LIGHTWEIGHT VERIFICATION OF ARRAY INDEXING

This chapter describes the Index Checker, a set of cooperating specialized pluggable typecheckers for preventing array bounds violations. The Index Checker is faster than prior sound approaches to preventing array bounds violations, with comparable precision and programmer effort. Therefore, it is an example of a technique for improving the expressiveness of lightweight verification: the Index Checker is more lightweight than the approaches for preventing array bounds violations that came before. The Index Checker was originally described in [176].

4.1 Motivation

An array access `a[i]` is in-bounds if $0 \leq \texttt{i}$ and $\texttt{i} < \mathit{length}(\texttt{a})$. 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 [234]: buffer overflows enabled the Morris Worm, SQL Slammer, Code Red, and Heartbleed, among many others, allowing hackers to, for example, steal 4.5 million medical records [125]. If all array accesses were guaranteed to be in-bounds, these attacks would be impossible. 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.

Many academic and industrial approaches have been put forward to address this important problem. These advances have made both theoretical and practical contributions to science and engineering. But, none of these approaches is a lightweight verification tool; all fail one or more of the criteria of the platonic ideal we described in chapter 1. *Dynamic bounds checking* augments the program with run-time checks, crashing the program (i.e., by throwing an exception) instead of performing illegal operations. This widely adopted approach 1) has run-time overhead and 2) still causes programs to behave in an undesirable way (crashing is less undesirable than a buffer overflow, but most programmers still do not want to write programs that crash!). Heuristic-based, compile-time *bug-finding tools* are useful for finding some defects, but provide no guarantee, failing the soundness criterion. Several types of sound static analyses could prevent bounds errors at compile time, though most sound tools have not been evaluated in substantive case studies. *Proof assistants* fail the compatibility, speed, and comprehensibility criteria: they require heroic effort to use and understand, and they require re-implementation of the program or the programming language. *Automated theorem provers* translate the verification problem into a satisfiability problem, then invoke a solver; they fail at comprehensibility and either speed or determinism. *Bounded verification* (model checking or exhaustive testing) is generally not sound. *Inference* approaches do not require programmer annotations but fail the speed, determinism, and comprehensibility criteria. Hybrid static–dynamic approaches trade off the criteria, but satisfy no more of them than their component approaches. Section 4.6 discusses related work in more detail and gives citations.

We propose 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. One of our contributions is to show that a carefully-designed collection of type systems—each specialized to a simple property—is an excellent fit to the problem.

We have developed a set of lightweight, easy-to-understand type systems, which we implemented in a tool called the Index Checker. The Index Checker 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. The Index Checker's type systems are simple enough for developers to reason about yet rich enough to guarantee that real programs are free of indexing errors (or to reveal subtle errors). The Index Checker verifies that programmer-written type annotations are consistent with the code; that is, at run time, the values have the given type. This provides a documentation benefit: programmers cannot forget to write documentation of necessary indexing-related properties, the documentation is guaranteed to be correct, and the types are both more formal and more concise than informal English documentation.

We implemented our type systems for Java. Our work generalizes to other languages because there is nothing about our type systems that is specific to Java. Our implementation handles arbitrary fixed-length data structures, such as arrays, strings, and user-defined classes. In fact, it found errors in collection classes defined in Google's Guava library.

We evaluated our type system with three case studies on open-source code in everyday industrial use. The case studies show that the Index Checker scales to practical programs at reasonable programmer effort. The Index Checker found bugs in well-tested, widely-used code that were acknowledged and subsequently fixed by maintainers. Most importantly, it certified that no more array bound errors exist in checked code (modulo soundness guarantees).

The primary contributions described in this chapter are:

- Reducing array bounds checking to 7 kinds of reasoning and modeling these kinds of reasoning as simple type systems (section 4.2).
- Case studies showing that our implementation, the Index Checker, finds bugs in real programs (section 4.3).
- A comparison of our type-based approach to other approaches (section 4.4).

4.2 Verification via Cooperating Type Systems

An array access `a[i]` is in-bounds if two properties hold: $0 \leq$ `i` and `i` $<$ length(`a`).[1] Sections 4.2.1 and 4.2.2 show how to establish them, thus proving an access safe. However, an analysis that computes only those two properties would flood the user with false positives. One of our contributions is

[1] Evaluation of `a[i]` could suffer other problems: a value could be undefined (e.g., uninitialized, deallocated memory, or `null`); the stack could overflow if array dereference is implemented as a procedure call; etc. Our work specifically addresses array indices being within their bounds.

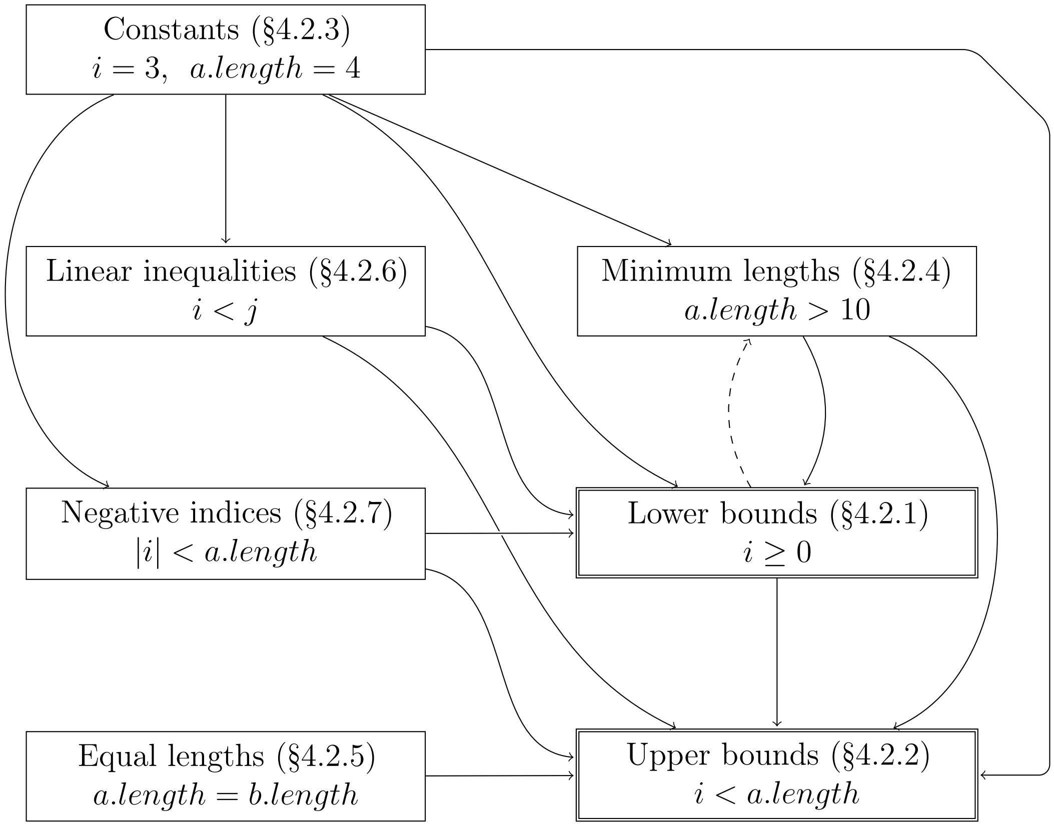


Figure 4.1: Information flows between type systems. The type systems with two boxes ensure each array access is safe; the other type systems support the work of these two. A dashed line indicates sound flow of information from user-written annotations (see section 4.2.8).

identifying 7 kinds of knowledge (fig. 4.1) that are adequately precise in practice, and designing abstractions (type systems) for each. Each subsequent section gives an example of safe code that cannot be typechecked under the analyses shown so far, and shows how we enhanced our design to accommodate that code. These enhancements improve precision without affecting soundness.

The Index Checker uses some dependent types [244]. A dependent type is a type whose definition depends on a value, such as “an integer less than the length of array `a`”. A dependent type may mention a value or the name of a variable.

In our new type systems, there are 86 type and inference rules beyond the standard ones. Each one is documented by a comment in the Index Checker implementation. This section gives the most important examples, but omits most of them because they are mostly obvious (this is a benefit of our simple type systems).

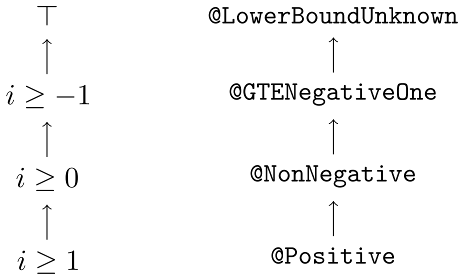


Figure 4.2: The type system for the lower bounds of integers. `@GTENegativeOne` stands for "**G**reater **T**han or **E**qual to **Negative One**". In each diagram in this section, arrows are subtyping relationships, properties described by the types are on the left, and type qualifiers in the Index Checker implementation are on the right.

4.2.1 A Type System for Lower Bounds

The first type system estimates a lower bound for each integer. Figure 4.2 shows the type hierarchy. An integer whose lower bound is less than zero may not index an array. Two rules for this type system are $e_2 : \text{@NonNegative} \vdash e_1\texttt{[}e_2\texttt{]}$ and $e_2 : \text{@NonNegative} \vdash e_1 \gg e_2 : \text{@NonNegative}$.

The simplest possible type system that would permit verification of (some) lower bounds would only include two types: non-negative and top. Our type system for lower bounds adds two additional types:

1. A type for positive integers, which is useful for one-based indices and for array accesses of the form `a[i-1]`.
2. A type for integers greater than or equal to -1, which is useful for loops that decrement the loop control variable by 1 and for `indexOf` methods that return -1 on failure.

For example, consider the following code from one of our case studies, which uses a one-indexed variable without documenting it:

```
/** Prints the matching item.
  * @param items the items to print from
  * @param itemNum specifies which item to print when there are multiple matches */
void printItem(Object[] items, int itemNum) {
  printItem(items[itemNum - 1]);
}
```

The Index Checker warns that `itemNum - 1` may be too low. The programmer should document `itemNum` as a 1-based index by declaring it as `@Positive int itemNum`. Then, the shown code type-checks, and the Index Checker also verifies that all clients of `printItem` respect its contract (i.e., all clients only pass `@Positive` integers for `itemNum`).

Our type system does not support other constant lower bounds; for example, it cannot express that $i \geq 2$. This design decision is intentional. Arbitrarily complex type systems require arbitrarily

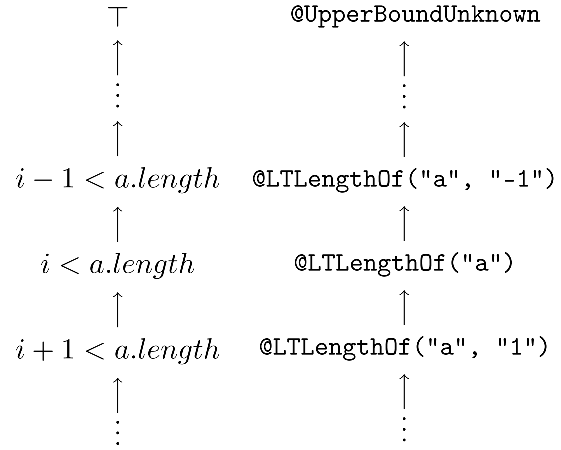


Figure 4.3: The type system for the upper bound of an integer i. `@LTLengthOf` stands for **L**ess **T**han **Length Of** and has two arguments: (1) an array a and (2) an offset x. The offset can be an arbitrary expression, such as `y + 6` or `z.indexOf(w)`.

complex reasoning. Instead, the Index Checker uses focused type systems that are sufficiently expressive to verify array bounds in practice.

4.2.2 A Type System for Upper Bounds

The index in an array access must be less than the length of the array. Figure 4.3 shows a dependent type system that soundly overestimates the relationship between all potential indices (i.e., integers) and the length of every array in scope. (The implementation is efficient, since it only stores types for in-scope arrays.) The type rules issue a type error when an index might exceed the bound of the array it is accessing. An integer may have several upper bound types: for instance, `i` may be less than the length of a and also less than or equal to the length of b. An access is safe if the index has at least one upper bound type that would permit it.

Every type in the upper bound type system also estimates an *offset* for the array. The variable plus the offset is less than the length of the array. Programmers usually omit the offset, which defaults to 0. The Index Checker infers offsets within method bodies, where they are most common. Each offset is an arbitrary expression. The Index Checker does sound, best-effort reasoning. As is typical for static analysis tools, it uses top as an estimate for non-linear arithmetic and other constructs that are challenging for static analysis.

To see why offsets are necessary, consider the following code from one of our case studies:

```
public void concat(int[] a, List<Object> b) {
    int b_size = b.size();
    Object[] res = new Object[a.length + b_size];
    for (int i = 0; i < b_size; i++) {
        res[i + a.length] = b.get(i);
```

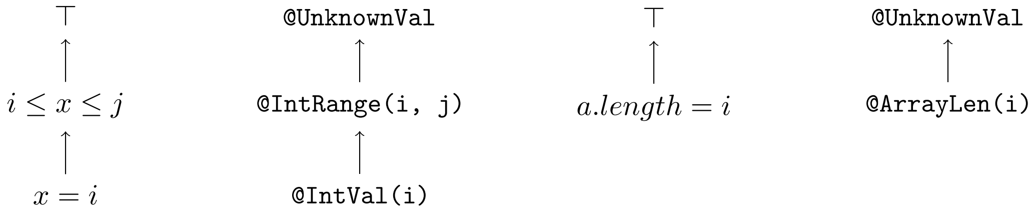


Figure 4.4: Type systems for constants. i and j are compile-time constants.

```
      }
      ...
  }
```

`res[i+a.length]` is safe: `i`'s type is `@LTLengthOf("res.length", "a.length")`. An inference rule automatically infers this type, without the need for programmer annotations, because `res.length = b_size + a.length` (which is non-negative) and `i` is less than `b_size`. The relevant type rule is:

$$\frac{e_1 : \text{@LTLengthOf}\,(\texttt{"}e_4\texttt{"}, \texttt{"}e_3\texttt{.length"}) \quad e_2 : \text{@LTELengthOf}\,(\texttt{"}e_3\texttt{"})}{e_1 \texttt{ + } e_2 : \text{@LTLengthOf}\,(\texttt{"}e_4\texttt{"})}$$

A programmer can give array `a` the type `@HasSubsequence("b", "startIndex", "endIndex")` to indicate that `b` is a view on a slice of `a`. This permits translation between indices for `a` and `b`.

4.2.3 Type Systems for Constants

The Index Checker obtains facts about indices and array lengths from an existing constant propagation and interval analysis. It provides three type qualifiers, whose type hierarchies are shown in fig. 4.4: `@IntRange(x, y)` represents an integer in the range x to y inclusive; `@IntVal(x)` is syntactic sugar for `@IntRange(x, x)`; and `@ArrayLen(x)` indicates that an array has exactly length x. In these type qualifiers, x and y are compile-time constants.

4.2.4 A Type System for Minimum Array Lengths

Consider this implementation of `min` from one of our case studies:

```
/** ... @param array a non-empty array ... */
public static int min(int @MinLen(1) ... array) {
  int min = array[0];
  for (int i=1; i<array.length; i++) { ... } ... }
```

The Javadoc states that the array must be non-empty, which the code relies on in `array[0]`. We expressed this formally as `@MinLen(1)`, and the Index Checker ensures that clients respect it. Figure 4.5 shows the type hierarchy.

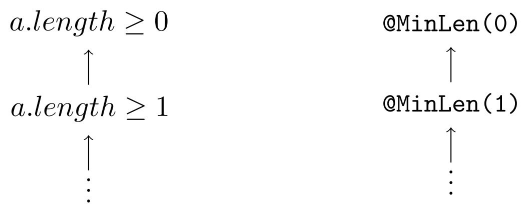


Figure 4.5: The type system for the minimum length of an array a. There is no distinguished top type since all arrays have zero or more elements.

Table 4.1: Type qualifiers for auxiliary type systems.

Section	Conceptual Type	Type Qualifier
§4.2.5	$a.length = b.length$	`T @SameLen("a") [] b`
§4.2.6	$i < j$	`@LessThan("j") int i`
§4.2.7	$\lvert i\rvert < a.length$	`@SearchIndexFor("a") int i`
§4.2.7	$\lvert i\rvert < a.length \land i < 0$	`@NegativeIndexFor("a")`

4.2.5 A Type System for Equal-length Arrays

This type system partitions arrays in scope at each program point into sets, where each element of a set has the same length. In a case study, we expressed that `xData` and `yData` have the same length:

```
double getSlope(Number[] xData, Number @SameLen("xData") [] yData) {
  ...
  for (int i = 0; i < xData.length; i++) {
    sxy = sxy + yData[i].doubleValue() * xData[i].doubleValue();
  }
}
```

The Index Checker verifies that both `yData[i]` and `xData[i]` are safe, and it rejects calls to `getSlope` where the arguments are not guaranteed to have the same length. Programmers rarely need to write `@SameLen` annotations; these annotations are inferred when two arrays are created using the same argument, when array lengths are tested against one another, etc. For example:

$$\frac{}{\texttt{new T[}e\texttt{.length]} : \text{@SameLen}\,(\texttt{"}e\texttt{"})}$$

The Index Checker uses a type system to express this partitioning. Each type represents one or more arrays with the same length as the array to which the type belongs. Although this representation is unusual, it permits the Index Checker to capture the partitioning in a way that retains the benefits of the type system approach. The transitivity of equality leads to an unusual least upper bound operation: if the two types have at least one array in common, then the least upper bound is the set of all of the arrays in either type. Otherwise, it is top.

4.2.6 A Type System for Simple Linear Inequalities

Consider the following annotated code from one of our case studies:

```
double calculateMedian(@LessThan("end + 1") int start, int end) {
  List working = new ArrayList(end - start + 1);
  ...
}
```

The `ArrayList` constructor argument must be non-negative. This type rulein the lower bound type system (section 4.2.1) establishes that `end - start + 1` is non-negative, using the `@LessThan` fact:

$$\frac{e_2 : \text{@LessThan}\,(\texttt{"}e_1\texttt{"})}{e_1 \;\texttt{-}\; e_2 : \text{@NonNegative}}$$

The type system for simple linear inequalities is intentionally limited: in practice, we found that tracking only the expressions that occur literally in comparison expressions in the source code is sufficient to achieve high precision. As is typical for static analyses, ours does not handle non-linear arithmetic.

4.2.7 A Type System for Negative Indices

The JDK's `binarySearch` method returns either the index of the target, or a negative value indicating where the target would be if it were present—that is, a negative index. The Index Checker models this contract with two type qualifiers: `@SearchIndexFor` and `@NegativeIndexFor`. `@SearchIndexFor` is refined to `@NegativeIndexFor` only by comparison with literal `0`, which we found was sufficient for precision in practice. The type systems described in sections 4.2.1 and 4.2.2 infer expression types based on this information. For example, the type system for upper bounds (section 4.2.2) includes the following type rule:

$$\frac{e_1 : \text{@NegativeIndexFor}\,(\texttt{"}e_2\texttt{"})}{e_1 \;\texttt{* -1} : \text{@LTLengthOf}\,(\texttt{"}e_2\texttt{"})}$$

One case study had a bug where a method should have returned -1, but returned the result of a binary search call. Before we implemented this type system, the Index Checker issued a warning at 14 calls to `binarySearch`, 13 of which were false positives. After, it issued a warning only at the error.

This type system is a microcosm for the advantages of using a series of simple analyses to verify code. In practice, we found that calls to `binarySearch` were a common source of false positives for our analysis. Based on this information, we could easily extend our system to handle `binarySearch`'s unusual semantics with another small, simple type system.

4.2.8 Cyclic Type System Dependence

Each type system uses facts computed by previously-run type systems (fig. 4.1). Some examples include:

$$e_1 : \text{@MinLen}\,(k_2)\,, k_2 < k_3 \vdash k_3 : \text{@LTLengthOf}\,(e_1) \qquad (\S 4.2.4)$$

Table 4.2: Annotations supported by the Index Checker as syntactic sugar for multiple annotations from the type systems of section 4.2.

Type Qualifier	Syntactic Sugar For	Meaning
`@IndexFor("a") int i`	`@NonNegative @LTLengthOf("a") int i`	$0 \leq i < a.length$
`@IndexOrHigh("a") int i`	`@NonNegative @LTLengthOf("a", "-1") int i`	$0 \leq i \leq a.length$
`@IndexOrLow("a") int i`	`@GTENegativeOne @LTLengthOf("a") int i`	$-1 \leq i < a.length$
`@LengthOf("a") int i`	`@NonNegative @LTLengthOf("a", "-1") int i`	$i = a.length$
`@LTEqLengthOf("a") int i`	`@LTLengthOf("a", "-1") int i`	$i \leq a.length$

$$e_1 : \text{@SameLen}\,(e_2)\,, e_3 : \text{@LTLengthOf}\,(e_1) \vdash e_3 : \text{@LTLengthOf}\,(e_2) \qquad (\S 4.2.5)$$

In some cases, two type systems could each benefit from the other running first. Two rules that are often needed in real-world code are:

$$e : \text{@Positive} \vdash \texttt{new int[}e\texttt{]} : \text{@MinLen}(1)$$

$$e : \text{@MinLen}\,(k) \vdash e\texttt{.length - }\ (k-1) : \text{@Positive.}$$

If the lower-bound type system (section 4.2.1) runs first, then the latter rule will never fire, because no types have `@MinLen` qualifiers yet. If the minimum-length type system (section 4.2.4) runs first, then the former rule will never fire.

One possible solution would be to run typecheckers multiple times until a fixed point, each time utilizing only knowledge that had already been established. However, we want to retain the speed of running each typechecker only once for each line of code.

Instead, the Index Checker uses rely-guarantee reasoning [169] to implement the mutual dependency. The analysis that computes minimum array lengths runs first, and relies on (that is, assumes the truth of) any annotation explicitly written by the programmer. The Index Checker guarantees that the explicit positive annotation will be checked later by the type system in section 4.2.1.

This required extending the Checker Framework, which made no distinction between user-written and inferred annotations before.

4.2.9 Implementation Notes

In addition to the type qualifiers described previously, every type system contains a bottom type $\perp$, which is the type of `null`. $\perp$ is needed because the Index Checker handles all of Java's numeric types (i.e., both `int` and `Integer`).

The Index Checker has annotations that are syntactic sugar for combinations of annotations from two type systems (table 4.2) to help programmers express common invariants.

The Index Checker provides 1,238 annotations for the JDK (Java's standard library), which we wrote based on the JDK's documentation. These annotations are trusted but not checked; a future case study could verify the JDK implementation. The Index Checker's users can write similar annotations for other libraries.

Table 4.3: A summary of the three case studies. Code size is non-comment, non-blank lines, as measured by `cloc` [75]. “Bugs fixed” is those fixed by the developers at the time of writing. “Annotatable locations” is the number of places that an annotation could be written (approximately all uses of integral and array types in the program). A “non-trivial check” of a type rule involves a type other than $\top$ or $\bot$, such as array accesses, calls to procedures with annotated formal parameters, etc. “False positive %” is the number of false positives divided by the number of non-trivial checks.

	Guava*	JFreeChart	Plume-lib	**Total**
Lines of code	10,694	94,233	14,586	119,503
Annotatable locations	10,571	65,051	12,074	87,696
Annotations	547	2,936	242	3,725
Bugs detected	5	64	20	89
Bugs fixed	5	12	20	37
False positives	114	350	43	507
Non-trivial checks	3,084	12,520	1,817	17,421
False positive %	3.7%	2.8%	2.4%	2.9%
Java casts	219	2,707	223	3,151

* Guava packages `base` and `primitives`.

The Index Checker is implemented in 5,539 lines of code. The Index Checker’s implementation consists of one typechecker for each type system in section 4.2. This structure keeps each typechecker relatively small and easy to understand. Each typechecker contains a definition of the type hierarchy, type rules, and inference rules.

The type and inference rules are implemented directly, without calling an external solver. This keeps performance fast and predictable, and can be more expressive, at the cost of an increase in implementation size.

The Index Checker is distributed and maintained as part of the Checker Framework [89].

4.3 Case Studies of the Index Checker

We ran the Index Checker on three open-source Java projects (table 4.3) used previously to evaluate bug-finding and verification tools [77, 145, 57]. Our goal was either to verify that each was free of array indexing bugs, or to find and fix all their array indexing bugs.

We first read the developer-provided documentation and re-wrote it formally, as Index Checker annotations. Then, we ran the Index Checker, investigated each warning it issued, and took one of these actions: (1) Added missing annotations to an incomplete specification and re-ran the Index Checker. (2) Fixed a bug in the code and reported it to the maintainers. (3) Determined that the code was correct, but the Index Checker was not sufficiently powerful to prove it correct. We suppressed these false positive warnings.

Each case study was performed by someone who was not familiar with the codebase being

Table 4.4: Annotation density: number of annotations per line of code. Annotations with larger denominators are rarer. `@SearchIndexFor` and `NegativeIndexFor` only appear in the JDK. The annotations appear in the same order as in section 4.2.

Annotation	Guava	JFreeChart	Plume-lib	Overall
`@NonNegative`	1 / 139	1 / 50	1 / 228	1 / 59
`@GTENegativeOne`	1 / 972	1 / 1193	1 / 2917	1 / 1258
`@Positive`	1 / 446	1 / 2005	1 / 1216	1 / 1440
`@LTLengthOf`	1 / 891	1 / 7249	1 / 912	1 / 2915
`@HasSubsequence`	1 / 972	0	0	1 / 10865
`@IntRange`	1 / 563	1 / 688	0	1 / 766
`@IntVal`	1 / 10694	1 / 13462	0	1 / 14939
`@ArrayLen`	1 / 5347	1 / 819	1 / 503	1 / 819
`@MinLen`	1 / 191	1 / 1812	1 / 972	1 / 972
`@SameLen`	1 / 1528	1 / 1428	1 / 858	1 / 1328
`@LessThan`	1 / 289	1 / 18847	1 / 2084	1 / 2439
`@SearchIndexFor`	0	0	0	0
`@NegativeIndexFor`	0	0	0	0
`@IndexFor`	1 / 281	1 / 202	1 / 521	1 / 224
`@IndexOrLow`	1 / 167	1 / 47117	1 / 3647	1 / 1707
`@IndexOrHigh`	1 / 67	1 / 4712	1 / 810	1 / 604
`@LTEqLengthOf`	1 / 891	1 / 9423	1 / 14586	1 / 5196
`@LengthOf`	0	1 / 2692	0	1 / 3415
All annotations	**1 / 20**	**1 / 32**	**1 / 60**	**1 / 32**

checked but familiar with the Index Checker. We spent most of our time understanding subtle and/or undocumented code and improving documentation or fixing errors. Reading the entire codebase was not necessary.

Overall, the case studies demonstrate three results:

1. The Index Checker scales to sizable programs.

2. The Index Checker finds real bugs even in well-tested programs. Every one of the bugs leads to a program crash. Of the bugs, 37 were validated by maintainers. Details of the bugs appear in sections 4.3.1–4.3.3.

3. The Index Checker issues fewer false positives than Java's type system. Java programmers write casts to suppress false warnings from Java's type system. The Index Checker handles casts soundly, issuing a warning at each annotated type downcast. Sections 4.3.4 and 4.3.5 discuss false positives and annotation burden.

4.3.1 Guava Case Study

Guava [146] is Google's general-purpose core libraries for Java. Guava is considered extremely reliable: it is used extensively in production at Google and elsewhere, and its test suite is larger than its code. We annotated two packages, `com.google.common.base` and `com.google.common.primitives`. Most of the uses of fixed-length sequences in `base` are strings, either directly or through the `CharSequence` interface. `primitives` uses arrays and defines custom fixed- and mutable-length collections. Much of the code is duplicated for each of Java's eight primitive types.

We found 5 bugs in Guava, all of which were instances of the same programming mistake. The bug involves factory methods for immutable collections, which begin with code such as:

```
public static ImmutableIntArray of(int first, int... rest) {
  int[] array = new int[rest.length+1];
```

This code uses unchecked integer addition to compute the length of a new array. If the `rest` array has a length equal to `Integer.MAX_VALUE`, the addition overflows, and the method attempts to allocate an array of negative size. Guava's maintainers classified this bug[2] as priority one[3] and accepted our patch[4] that documents the maximum allowed array length and checks this requirement at run time.

The Guava package `primitives` consists almost entirely of classes working with or representing sequences of Java primitive types. These classes require many annotations in their public interfaces, accounting for the relatively large number of annotations Guava required. For example, of the 160 total `@IndexOrHigh` annotations we wrote in the Guava case study, 114 were in this package on parameters of methods that take a pair of indices specifying a range in a sequence, such as:

```
List<Long> subList(@IndexOrHigh("this") int fromIndex,
    @IndexOrHigh("this") int toIndex) { ... }
```

4.3.2 JFreeChart Case Study

JFreeChart [143] is used by Java application developers to include graphs and charts in their programs. It uses arrays and fixed-size structures extensively to represent data it draws onto charts.

The Index Checker found 64 bugs in JFreeChart, all of which would lead to crashes. Of these bugs, 24 were in code and 40 were in documentation. Of the code bugs, 14 were failures to check arguments to public methods before indexing; 3 were inconsistencies between JFreeChart's time classes and the JDK's calendar class; and 2 resembled the bug in fig. 4.6. The other 7 code bugs all had different causes. The 40 documentation bugs involved undocumented assumptions made by code. Clients could pass values permitted by the documentation and cause a crash. We fixed these bugs by modifying the documentation to reflect the actual assumptions.

The maintainers accepted our first two patches (fixing 11 bugs), then went dormant. Our third patch [174] (which fixed 1 more bug, bringing the total to 12) was ignored for over two years before it was merged by the maintainers, so we did not file any more patches.

[2] `https://github.com/google/guava/issues/3026`

[3] The highest priority is zero, but the Guava team has never acknowledged a priority zero bug in their public repository.

[4] `https://github.com/google/guava/pull/3027`

```
static final int[] LAST_DAY_OF_MONTH
  = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
public static SerialDate addMonths(int months, SerialDate base) {
  int totMm = 12 * base.getYYYY() + base.getMonth() + months - 1;
  int yy = totMm / 12; int mm = totMm % 12 + 1;
  int lastDayOfMonth = SerialDate.lastDayOfMonth(mm, yy);
  ...
}
public static int lastDayOfMonth(int month, int year) {
  final int result = LAST_DAY_OF_MONTH[month];
  ...
}
```

Figure 4.6: The Index Checker found this bug in JFreeChart. The argument to `addMonths` may be negative, making `mm` negative and causing `lastDayOfMonth` to crash. For simplicity, the figure elides code for leap-year accounting.

4.3.3 Plume-lib Case Study

Plume-lib [112] was a library of Java utility methods[5]. Like Guava, it was well-tested: its JUnit tests contained 2,693 lines of code and 1,136 `assert` statements.

We found 20 bugs, all of which were fixed by the developers.

We found 9 code defects. One involved a table header that was printed even when the table itself was empty. Three were crashes due to unchecked, externally supplied indices; we added code that checks the user's input and prints a user-friendly error message rather than a stack trace. One was an access to an array that might have zero length. One was a crash that would occur only when two copies of the same array were passed to a method. One was a case where an array's length was checked, but then it was dereferenced unsafely anyway. Two were in routines that should have accepted ragged arrays, but used the length of the first subarray for all subarrays.

In 11 cases, methods were missing documentation about their requirements and assumptions; without the documentation we added, a user might have supplied illegal input and caused a crash.

4.3.4 Causes of False Positives

Every sound type system rejects some safe programs that never perform an undesired operation at run time. If the programmer is confident the code is safe (due to manual, or other, verification), the programmer can suppress the warning. When using the Index Checker, this is expressed using Java's `@SuppressWarnings` syntax.

The Index Checker issues 507 false positive warnings in our case studies. This number may seem high, but it is less than the Java type system. Programmers wrote 3,151 casts to suppress false

[5] It has since been split into several smaller projects.

positives from the Java type system. (One could measure the percentage of reports that are false positives, but this would be primarily a measure of code quality rather than tool quality. For any program, once its bugs are fixed, the percentage of reports that are false positives is by definition 100%. All our subject programs had high quality to begin with. The most effective way to use a typechecker is throughout development, not after testing and deployment.)

This section now discusses the three most common reasons for the Index Checker to issue a false positive in our case studies.

(1) The Index Checker is restricted to immutable length data structures (see section 4.5.1). When code relies on interoperation between mutable-length data structures and arrays, the Index Checker may issue false positives. For example, consider the following method from Plume-lib:

```
public <T> T[] concat(List<T> a, T[] b) {
    T[] result = (T[]) new Object[a.size() + b.length];
    for (int i = 0; i < a.size(); i++)
        result[i] = a.get(i); // false positive
    ...
}
```

`result`'s length is greater than `a`'s size, so `i` must be an index for `result`, but the Index Checker conservatively assumes that `a`'s size might have changed before `i` is used to access `result`. Interfaces for custom collections with implementations that are backed by either an array or a list are also common. All interactions between arrays and lists required 56 warning suppressions.

(2) JFreeChart commonly uses an object's index to fetch it from another object that, by construction, must contain it. In these cases, JFreeChart correctly does not check for a -1 return value when calling `indexOf` methods. An example follows:

```
public class DefaultPolarItemRenderer {
/** The plot the renderer is assigned to. */
private PolarPlot plot;
public LegendItem getLegendItem(...) {
  XYDataset dataset = plot.getDataset(plot.getIndexOf(this)); //false positive
  ...
```

Because `this` is a field in `plot`, `getIndexOf` cannot return -1 here even though its documentation indicates that it could. We suppressed 46 warnings caused by this pattern. Reasoning about an invariant like this is beyond the capabilities of the Index Checker.

(3) JFreeChart defines custom data structures that are backed by ragged arrays, called `Datasets`. Every method to access a `Dataset` requires two parameters: an index into the array of "series" (that is, other arrays), and an index into the corresponding series. For example, the definition of `getZ` in the `XYZDataset` is:

```
      public Number getZ(int series, int item);
```

The Index Checker handles most uses of `Datasets` correctly, but a limitation of the Checker Framework causes 40 false positives when a programmer casts a generic `Dataset` to a more specific subclass of `Dataset`, like `XYZDataset`. The Checker Framework conservatively assumes that this cast invalidates indexes dependent on behaviors because the new type may not support those behaviors.

4.3.5 Annotation Burden and Benefit

Table 4.4 shows how often each Index Checker annotation needed to be written. Verification of array indexing is a difficult problem and provides strong guarantees, so some programmer effort is expected. The annotation burden is less than for Java generics, for 2 of the 3 subject programs; that is, the programs contain fewer Index Checker annotations than Java generic type arguments. The annotations are also much smaller than the programs' test suites; this comparison is relevant because testing is another way to find errors, though in each of these programs the testing missed errors.

The annotations are more concise and precise than English, so writing them *reduces* the size of documentation. Since they are typechecked, they are more reliable than English documentation, which may be out of date. So the annotation count is less a measure of programmer burden than a measure of documentation benefit.

4.4 Comparison to Other Approaches

Array indexing bugs are an important problem in practice, so many tools have been built to detect them. We compared the Index Checker to three tools, representing three different approaches to the problem.

FindBugs is bug-finding tool widely deployed in industry that uses heuristic-based pattern-matching and static analysis [19]. It emphasizes its low false-positive rate and ease of use; unlike the other tools, it does not aim to be sound. We used FindBugs v. 3.0.1 with all 9 bug patterns related to array or string indexing.

KeY [9] verifies Java Modeling Language (JML) [196, 59] specifications, which can express full functional correctness properties, using an automated theorem prover. We used KeY v. 2.6.3 with Z3 [78] v. 4.5.1. We translated Index Checker annotations to JML, and otherwise instructed KeY to verify the weakest possible contracts.

Clousot checks Code Contracts[6] on .NET methods [120]. It works by abstract interpretation. We used the Code Contracts v. 1.9.10714.2 extension to Microsoft Visual Studio Enterprise 2015 v. 14.0.25431.01 Update 3 with Microsoft .NET Framework v. 4.7.02556, with arithmetic and bounds checks enabled and the SubPolyhedra [192] abstract domain. We disabled contract inference to prevent Clousot from detecting that parts of some test cases are unreachable. We hand-translated code from Java to C# and Index Checker annotations to Code Contracts. We used equivalent classes and methods from the .NET Framework in place of JDK classes if there was a clear correspondence; otherwise, we wrote stub classes to mimic them. For example, we translated `new String(bytes, 0, pos)` to `Encoding.ASCII.GetString(bytes, 0, pos)`.

We first tried to run each tool on the case study programs from section 4.3 (section 4.4.1). We also ran each tool on an example of each bug found by the Index Checker that was accepted by developers (section 4.4.2). Finally, we compared Clousot to the Index Checker using the test suites of the two tools (sections 4.4.3 and 4.4.4).

[6] `https://docs.microsoft.com/en-us/dotnet/framework/debug-trace-profile/code-contracts`, `https://www.microsoft.com/en-us/research/project/code-contracts/`

Table 4.5: Effectiveness of 4 tools in finding real indexing bugs in their default configurations.

	FindBugs	KeY	Clousot	Index Checker
True Positives	0 / 18	9 / 18	16 / 18	18 / 18
False Negatives	18 / 18	1 / 18	2 / 18	0 / 18

4.4.1 Case Study Programs

FindBugs neither found any bugs nor issued any false positives on the case study programs from section 4.3. This result seems in line with its mission: each of the bugs we found was at least moderately complex, and FindBugs will only issue a report about a very obvious indexing error. In particular, its main `RANGE` rule will only report a bug if a constant-valued array is accessed with a too-large constant. By construction, code that fits this pattern *is* a bug, but FindBugs does not do any further reasoning, to avoid false positives.

KeY failed to run on all of our case study programs. KeY is a whole-program analysis. The KeY distribution includes a tool to generate stubs for unavailable code, but the generated stubs omitted some JDK dependencies for each of our benchmarks. KeY also does not support features of Java used in real-world code: generics, floating point numbers, several Java 7 features, and all Java 8 features.

We did not run Clousot on the case study programs because hand-translating each program from Java to C# would have been prohibitively time-consuming.

4.4.2 Developer-accepted Bugs

We categorized all the bugs that the Index Checker detected and the case study programs' developers fixed into 18 categories[7]. We created a minimized example of each, a few lines of code long. We also wrote corrected versions of these minimized examples. We then ran all four tools on these 36 code snippets (table 4.5).

Although KeY aims to be sound, its default configuration has a false negative: it verified as correct the buggy code from Guava. By enabling the most faithful Java integer semantics, KeY can detect the bug. As expected, KeY rejected the other buggy test cases. However, in 8 cases it gave the identical verification failure on the fixed code, indicating that the verification failure had nothing to do with the bug. In only 2 cases did KeY verify the correct code without additional input. In 7 cases, KeY did not verify the correct code, but gave a different verification failure than it did on the buggy code. We found ourselves unable to interpret KeY's output to locate the bug, but we count these 7 cases as true positives in table 4.5 because a KeY expert might be able to do so.

Clousot is effective—it detected most of the bugs. In its default configuration, Clousot failed to detect two bugs, instead reporting that they were correct code. One involved conversion from an array of bytes to a string:

[7] We excluded the bug fixed in JFreeChart by our third patch [174], because the JFreeChart developers did not merge the patch for over two years, and when this analysis was conducted the JFreeChart developers had been dormant for over three months, so we did not know if they would ever accept our patch.

```
public static void doSession(InputStream stream, byte[] buf) {
  Contract.Requires(buf != null); int pos = stream.read(buf);
  string actual = Encoding.ASCII.GetString(buf, 0, pos);
  ...
}
```

The `InputStream.read` method returns `-1` when the end of the file is reached. Client code should check it before using it as an index.

The other bug is the Guava bug involving overflow shown in section 4.3.1. Clousot has a command-line option for unsoundly checking overflow, which is disabled by default and is not available from the Visual Studio extension. With this command-line option enabled, Clousot finds the Guava bug.

Based on the results in table 4.5, we dropped FindBugs and KeY from further comparisons and focused on Clousot.

4.4.3 The Index Checker's Test Suite

We ran Clousot on the Index Checker's test suite, which is a set of Java files containing correct and incorrect code with expected warnings. The Index Checker passes this suite with no false positive warnings. The tests are mainly real-world code encountered in our case studies, not corner cases designed to highlight the Index Checker's particular strengths.

We translated the test suite into C#. We skipped tests that use JDK classes without a direct equivalent in .NET, tests that use bottom types, and tests that use polymorphic qualifiers (which cannot be expressed by Code Contracts). We also skipped tests that check whether particular Java features are handled correctly. The resulting test suite consists of 163 C# files containing 4,608 lines of code.

One of the tests revealed a bug in Clousot. Clousot incorrectly reported that the asserted condition is false, but it is always true:

```
 int a = -1;
int d = 2;
int u = a / d;
Contract.Assert(u >= -1);
```

Clousot issued 56 false positives. The most common causes for Clousot to issue a false positive were: computing indices by division, max, min, and bitwise-and operations (20 warnings), inferring that arrays equal by == have the same length (10 warnings), and handling an array that contains all values of an enum type (5 warnings).

4.4.4 Clousot's Test Suite

Clousot's test suite has similar structure to the Index Checker's, with C# files and expected warnings. We translated the parts of the suite that check array accesses to Java, and the associated code contracts to Index Checker annotations. This test suite contained 26 files with 2,633 lines of code. We did not write type qualifiers for contracts or assertions that could not be expressed in our type systems.

The Index Checker issued 136 warnings, of which 78 were true positives (expected warnings) and 58 false positives. Eleven of these false positive warnings were because the code contracts could not be translated into the Index Checker's type systems. For example, the Index Checker does not have an annotation to express that the return value of a method is equal to the value of a field. The most common reasons for false positives were: the Index Checker failed to refine types of local variables after a check or in a loop (18 warnings), code that increments an independent index variable in a loop over an array (8 warnings), using a multiple of an index as a size of a newly allocated array (6 warnings). Clousot's test suite includes many linear inequalities that are more complex than those handled by the type system in section 4.2.6. This is probably because complex linear inequalities are a strength of Clousot's SubPolyhedra domain. The Index Checker uses a cheaper analysis that issues few false positive warnings on the real-world code in our case studies, suggesting that Clousot's test suite may be uncharacteristic of real-world code.

4.4.5 Discussion: Types vs. Expressions

Clousot checks specifications written as arbitrary C# expressions at statement boundaries. By contrast, Index Checker specifications are type qualifiers written on type uses. These differ in expressiveness, conciseness, and solver completeness.

Expressions are in general much more expressive than types. However, they are not strictly so. Unlike Code Contracts, types support polymorphism and variance. As an example, consider the following method fragment from the JFreeChart case study:

```
Map<@NonNegative Integer, CategoryDataset> datasets;
List<@NonNegative Integer> getDatasetIndices(DatasetRenderingOrder order) {
  List<@NonNegative Integer> result = new ArrayList<>();
  for (Map.Entry<@NonNegative Integer, CategoryDataset> entry :
        datasets.entrySet()) {
    if (entry.getValue() != null)
      result.add(entry.getKey());
  } ...
  return result;
}
```

Clousot can use a quantifier to specify that the returned list should have non-negative elements:

`Contract.ForAll( Contract.Result <List<int»(), j=>j>=0)` .

However, Clousot cannot check that only non-negative numbers are added to the list (because the `add` method doesn't have a precondition requiring that the argument is non-negative), and cannot prove this property. By contrast, the Index Checker can verify the whole method, using the fact that the values added to the list are keys from the dataset map which are non-negative. Similarly, wherever this method is called, the Index Checker can use the fact that any integer retrieved from the list is non-negative.

Types are often more compact and easier to read. Expressing properties in the underlying programming language is convenient for tooling, but often verbose. Compare declaring `i` as `@IndexFor("a")` versus writing `Contract.Requires(i >= 0 && i < a.Length)`. Or, consider annotating an interface handling non-negative values:

```
public interface Values {
    public @NonNegative int getItemCount();
    public Number getValue(@NonNegative int index);
}
```

The corresponding Code Contracts requires defining a ghost abstract class explicitly implementing the interface. The programmer must stub out each method just to write the specifications, which dramatically increases the size of the code: instead of modifying a few lines by adding annotations, the interface must be copied and then annotated. Combined with the more verbose syntax of Code Contracts (i.e. `Contract.Ensures(Contract.Result<int>() >= 0);` instead of `@NonNegative int`), the required changes to the code are almost an order of magnitude larger. Using the underlying language is sometimes convenient, but can significantly clutter the code.

A significant difference is that a programmer can predict if the Index Checker will verify a program's correctness. When a programmer states an argument for why each array access in a program is legal, if the argument are expressible in the Index Checker's language, then the Index Checker will verify the program is correct (unless the Index Checker has a bug). By contrast, using expressions to represent program facts allows arbitrarily complex reasoning, so programmers cannot predict whether Clousot will succeed or fail.

Although the Index Checker's specifications are more restrictive, they are effective in practice. The Index Checker's design also makes verification significantly faster. Though a direct comparison is impossible (because the tools operate on different languages), Clousot takes 222 minutes to check Boogie [23] (85664 LOC) and timed out on 9 methods, whereas the Index Checker takes 8 minutes to check JFreeChart (94233 LOC), on the same commodity hardware. The design of a set of types that is sufficiently expressive, yet efficient to check, is one of our contributions.

4.5 Limitations and Threats to Validity

Like any code analysis tool, the Index Checker only gives guarantees for code that it checks. The guarantee excludes native code, unchecked libraries like the JDK, and dynamically generated code. The Checker Framework handles reflective method calls soundly [25]. Type casts do not affect soundness: the Checker Framework issues a warning at every annotated type downcast. The Index Checker makes no guarantees about mutable-length data structures such as Java `Lists` (see section 4.5.1). The Index Checker makes no guarantees in the presence of overflow, though its best-effort analysis found some such errors in Guava (section 4.3.1).

Like any sound static analysis, the Index Checker cannot verify all correct code and produces some false positive warnings. A programmer must apply some other type of reasoning to such code; if the code is indeed safe, the programmer must suppress the warning. The Index Checker trusts that when a programmer suppresses a warning: (1) the code is safe, and (2) its annotations are correct.

Our results, while encouraging, may not generalize. The Index Checker might suffer more false positives or be harder to use if our subject programs are uncharacteristic. We chose our case studies to be array-heavy code; other code might not require as many annotations. Over a dozen people have used the Index Checker, but its usability by programmers has not been established.

Our case studies demonstrate the Index Checker's bug-finding power on well-tested, deployed code. Our case studies are a worst-case scenario for the tool's usefulness. It would be both more useful and easier to use the Index Checker from the inception of a project. This would validate the program's design and prevent bugs from ever entering the code.

It is possible that our type system or the Index Checker implementation might contain errors. We have mitigated this danger by having multiple authors review every type rule and every line of code, and with a large test suite of 403 test cases and 9,904 lines of test code.

4.5.1 Fixed-length vs. Mutable-length Collections

Our type systems work for fixed-length collections, whose size does not change after the object is constructed. The Index Checker's annotations can be written on type declarations and uses, which enables support both for JDK classes such as `String` and for user-defined classes. This works even for classes that do not extend `Collection` nor have one as a field, as long as the class's abstraction represents more than one item. To specify that a class contains an array or collection field that acts as a delegate, the programmer writes `@SameLen` annotations (section 4.2.5).

Handling mutable-length collections is interesting future work that requires three techniques. The major part is tracking of indexing and lengths, which is described and implemented in this paper. The second part is handling operations that change the size of a data structure, such as `add` and `remove`. This is not difficult, though it requires specifications about side effects or their absence. The Checker Framework invalidates dataflow facts about expressions at all possible reassignments, including non-pure method calls. The third part is precisely tracking all aliasing, so that when a list's length changes, the lengths of all (and only) aliased lists are also changed. This is challenging, but not specific to array indexing. New implementations of alias analyses could be substituted in as the community develops them. We do not know whether any existing analysis would be sufficiently precise for our needs.

4.6 Related Work: Array Bounds

The most common approach to array bound errors is dynamic checking, which crashes the program rather than permitting an unsafe operation. Most modern languages build this into the run-time system, and tools such as Purify [155] and Valgrind [230] do it for unsafe languages like C. These checks incur significant time or space overhead, despite research on reducing their cost [47, 255]. Other research aims to integrate bounds checks into unsafe languages [107, 107, 224, 227, 166, 71, 319]. Fundamentally, the goal of these approaches is to safely crash the program when an out-of-bounds array access would occur at run time. By contrast, the Index Checker imposes no run-time cost, and it prevents the program from crashing due to array bounds mistakes.

The classic dynamic approach of allocating, maintaining, and checking shadow bits can be performed statically via a dataflow analysis [117, 110]. An early example [288] was notable in not using the standard approach of unsound heuristics, but instead creating an infinite chain of approximations to the general loop invariant, using the "weakest liberal precondition".

Bug-finders such as FindBugs [19] and Coverity [31] are easy to use and useful in finding some errors. Unlike the Index Checker, their heuristics are too weak to find all errors so they offer no guarantees. We compared directly against FindBugs in section 4.4.

Extended Static Checking [199, 83, 126], KeY [9], and Dafny [198] translate verification conditions into the language of powerful satisfiability engines or automated theorem provers, such as Z3 [78].

This is the dominant paradigm in bounds verification and in some other types of program analysis. We compared directly to KeY in section 4.4. Unlike the Index Checker, these tools suffer brittleness or instability: a small, meaning-preserving change within a method implementation may change the tool's output from "verified" to "failed" or "timeout", or might lead to different diagnostics in unrelated parts of the program [200, 156]. Scalability and usability are also challenges.

Wei et al. [307] evaluated different program analyses, including representations for integers and heap abstractions. They and we both found that polygons are expensive and not very helpful; simpler analyses can suffice.

Other researchers have applied dependent type systems to the problem of array bounds. With Xi and Pfenning's dependent type system for a subset of SML [316], programmers write nearly arbitrary arithmetic linear inequalities, a type elaboration phase propagates them to unannotated expressions and collects a set of inequalities for the entire program, and then a solver for linear inequalities (such as Fourier variable elimination or the Omega Test [250]) is applied. It was evaluated on 8 procedures, and the annotation overhead was 17% of lines or 31% of characters. Liquid types [263] use the same type system but provide better inference, reducing the annotation overhead by combining type inference with predicate abstraction. The Dsolve tool found a bug in the Bitv library, then verified the array safety of 58 of its 65 routines, requiring 65 lines of annotations to verify 30 array access operations. No information is given about the unverified routines. The Index Checker's type system is more limited but works without an external solver. It has been designed to scale to real code with few false positives. ESPX [153] uses a dataflow analysis to find buffer overflows in C programs. It scales to large programs, but is unsound even on its own benchmarks. A type system with only upper and lower bounds [267] found 16 errors in one program. It is simpler than the Index Checker, but less general and has a higher false positive rate.

Abstract interpretation is as expressive as type systems [72], though in practice the two approaches lead to analyses that feel very different. Clousot or cccheck [120] has a number of similarities to the Index Checker: it is automated, checks programmer-written specifications, combines a set of interdependent analyses, works modularly, and performs some inference. We compared extensively to Clousot in section 4.4. Section 4.4.5 notes that Clousot's abstract domains are richer than the Index Checker's. Clousot was inspired by the limitations of theorem provers that we noted above.

The key challenge in designing a program analysis is selecting sufficiently precise abstractions that are still efficient. Clousot's authors found that octagons and intervals were too imprecise, buckets exacerbated non-determinism, and polyhedra were too inefficient. They settled on disjunctions of intervals, upper bounds, pentagons [208], linear inequalities, and SubPolyhedra [192] (a novel abstraction that is as expressive as Polyhedra, but has more limited inference). Clousot is non-deterministic, since it must apply a timeout to its long-running analysis. The Clousot researchers do not discuss case studies in which they examined Clousot's output, nor any bugs it revealed [120, 192, 208, 118]. Clousot issues a false positive at over 10% of array bounds [192]. We have found our choice of abstractions is simpler, more concise, more precise, faster, and effective in practice, and our implementation is more sound. Clousot made great strides, and the Index Checker makes significant further advances toward its vision.

4.7 Conclusion: the Index Checker

The Index Checker prevents array bounds violations using a set of 7 cooperating specialized pluggable typecheckers. Prior approaches to soundly preventing array bounds violations were heavier weight: they ran slower or required substantially more effort from the programmer than the Index Checker does. The Index Checker thereby increases the expressivity of lightweight verification: it is substantially more lightweight than the verification approaches that came before.

Chapter 5

LIGHTWEIGHT VERIFICATION FOR CONTINUOUS COMPLIANCE

Lightweight verification will not be a tool that everyday developers reach for unless they are aware that lightweight verification tools are *useful* to them. Applying existing lightweight verification techniques in new domains where they are effective is an important tool in *convincing* developers that the goal of lightweight verification is achievable. This section discusses one such domain where lightweight verification does well—compliance—and describes our experience deploying a collection of specialized pluggable typecheckers in a real, industrial setting. This chapter is adapted from [178].

5.1 Motivation

A compliance regime like the PCI DSS [240], FedRAMP [150], or SOC [10] encodes a set of best practices. For example, all of these regimes require that data be stored encrypted and that the encryption used be strong.

Many organizations are required by law, by contract, or by industry standard to only use software that is compliant with one or more regime. For example:

- VISA requires companies that process credit card transactions to use software that is compliant with the PCI DSS (Payment Card Industry Data Security Standard) [304]. PCI DSS certification assures card issuers that merchants will safely handle consumer credit card data [240]. Other card issuers have similar requirements [214, 280], and some U.S. states define non-compliance as a type of corporate negligence for which companies can be sued [222, 259].
- The U.S. government requires that cloud vendors be compliant with FedRAMP (Federal Risk and Authorization Management Program) [301, 150].
- Many customers of software providers expect a SOC (System and Organization Controls) report [10], which is used to evaluate how seriously potential vendors take security [6, 173].

Organizations with compliance requirements typically do not check the software they use for compliance themselves. Instead, when making a purchasing decision, an organization with compliance requirements typically requests an up-to-date compliance certificate from an accredited third-party auditor, also known as a Qualified Security Assessor (QSA) [242].

A compliance regime is made up of many *requirements*. For each requirement, the QSA imposes some *control*—a specific rule, usually defined by industry standard, and a process for enforcing that rule. For example, a QSA might impose the control “use 256-bit mode AES” for the requirement “use strong encryption.”

A compliance regime may also make requirements about the *process* used to create or run the software, such as what data is logged or which employees have access to data. This section focuses on requirements about the *source code*. Continuous compliance automates checking of these compliance requirements.

5.1.1 Problems with manual audits

Currently, the enforcement of source-code controls is primarily manual: employees of the auditor examine selected parts of the software to ensure it follows each control. The state of the art suffers the following problems:

Cost To sell its product, a vendor must participate in audits—often multiple times per year to show continuing adherence to the compliance regime. The vendor must pay the salary of its internal compliance officers, spend engineering time gathering evidence, and pay external auditors—often at significant and rising expense (more than $3.5 million each for a sample of 46 organizations in 2011) [248, 109]. A failed audit can cost millions of dollars more [134].

Judgment Humans can make mistakes of judgment. Engineers may provide non-compliant code for audit, which may lead to expensive failed audits. Auditors may incorrectly certify non-compliant code—false negatives. Auditors may raise concerns about safe code—false positives that must be investigated at further expense.

Sampling Auditors routinely sample randomly from the code under audit, because it is too expensive to manually examine it all. The standard reporting format for a PCI DSS audit includes a section dedicated to sampling procedures [241].

Regressions Audits occur periodically—typically every six or twelve months. Every code change is a chance for the software to fall out of compliance. In a study by Verizon's audit division, only 52.5% of organizations with an active compliance certification passed their re-audit without significant changes [300].

Our goal is to reduce costs, increase assurance and coverage, and prevent regressions by deploying lightweight verification tools.

5.1.2 Our approach: continuous compliance

We propose *continuous compliance*, which runs a lightweight verification tool on every commit to check compliance properties in source code. More formally, continuous compliance is the process of automatically enforcing source-code compliance controls whenever the code is changed, such as on every compiler invocation, commit, or pull request. Continuous compliance is an instance of continuous testing [265] and continuous integration [48, 131].

Continuous compliance eliminates the need for manual audits for specific source-code controls, resolving the problems described in section 5.1.1:

- The marginal *cost* of an audit is negligible, because auditors accept the results of running the verifier.
- The opportunity for mistakes of *judgment* is smaller: our tools found dozens of findings of interest to compliance auditors that all prior approaches had missed.

- *Sampling* in no longer an issue, because the verifier checks the entire codebase.
- *Regressions* are caught immediately when they occur, when it is cheaper for developers to fix them [55].

Even if continuous compliance is implemented only for some source-code controls, it reduces the scope of manual audits and makes them easier, cheaper, and more reliable.

Implementing a system for continuous compliance is challenging. To be acceptable to auditors, developers, and compliance officers, the continuous compliance system must be:

- **sound**: it must not miss defects. If it might suffer a false negative (missed alarm), then a manual audit would still be required.
- **applicable** to legacy source-code.
- **scalable** to real-world codebases.
- **simple** so that both developers and non-technical auditors can understand it and interpret its output.
- **precise** enough to produce few time-wasting false alarms.

These criteria are similar to the platonic ideal of a bug-prevention tool with a soundness requirement, so a lightweight verifier such as a specialized pluggable typechecker is the ideal way to satisfy them.

5.1.3 Contributions

This chapter has four main contributions:

- a *conceptual* contribution: the recognition that source-code compliance is an excellent domain for the strengths and weaknesses of lightweight verification.
- an *engineering* contribution: we designed and built five practical lightweight verification tools corresponding to common compliance controls.
- an *empirical* contribution: we evaluated the verification tools' efficacy on 654 open-source projects. We also compared them to state-of-the-art alternatives to demonstrate that only verification tools are suitable for continuous compliance—unsound bug-finding tools are insufficient.
- an *experiential* contribution: we deployed continuous compliance at Amazon Web Services (AWS). We report the reactions of developers and auditors to the introduction of continuous compliance. We believe that this contribution is the most important: it is a concrete step toward making verification practical for everyday developers.

Our key conceptual contribution is recognizing the benefits of verification tools to compliance auditors. The ideas were not obvious to *compliance officers and auditors*. The state of the practice is manual code examination, and the state of the art is run-time checking. Research roadmaps for improving the certification process do not even mention source code verification [298, 212, 204]. The ideas were not obvious to working *developers*. They believed that formal verification would require high annotation burden and would produce many false positive warnings. The ideas were not obvious to the *verification community*, who have focused on programmers (or modelers) rather than other important stakeholders such as compliance auditors.

Our engineering contributions are modest but non-trivial. We implemented five open-source verification tools for Java. The five compliance controls are common to many compliance regimes: encryption keys must be sufficiently long, insecure cryptographic algorithms must not be used, source code must not contain hard-coded credentials, outbound connections must use HTTPS, and cloud data stores must not be world-readable. We implemented our analyses as typecheckers, because typecheckers scale well and are more familiar to developers than other automated verification approaches such as abstract interpretation, model checking, and SMT-based analysis.

Our empirical contributions apply these tools to 654 open-source projects (section 5.6) and compare them to state-of-the-art tools for finding misuses of cryptographic APIs on a previously-published benchmark, with a focus on their suitability for continuous compliance (section 5.7). Only our tools suffered no false negatives—that is, they did not miss any real problems.

Our experiential contribution is deploying a continuous compliance system at scale at AWS, as part of its regular development process. Currently, 7 of its core services with a compliance requirement run verification tools on each commit, ensuring continuous compliance. External auditors accepted our verification tools as replacements for manual audits for these 7 services (section 5.8.1). Both developers and compliance teams are now more receptive to formal methods than they were before: both AWS and the auditors have spoken publicly on how verification has improved their process [311]. Security and compliance teams also run verifiers on a significant fraction of code at the company on a regular schedule—the most recent run when [178] was published (section 5.8.2) scanned over 68 million lines of code and required only 23 type annotations.

5.2 Compliance certification workflow

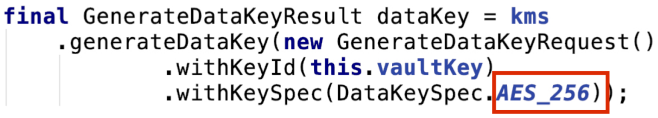
Section 5.2.1 describes the state-of-the-art approach for compliance certification of source-code properties, and section 5.2.2 describes our continuous compliance approach. Each subsection highlights three key phases of the workflow for comparison:

- **development** of the source code,
- **preparation** for an audit, and
- **review** by auditors.

As a running example, consider the industry compliance standard for AES encryption, which is to use the 256-bit mode. This rule corresponds to Testing Procedure 3.6.1.b in the PCI DSS [240].

5.2.1 Traditional audit workflow

While developers **develop** software, they must keep in mind the compliance rules and mentally check their code as they write it. Because compliance failures are very serious, significant code review effort is also expended toward keeping the codebase compliant.



```
final GenerateDataKeyResult dataKey = kms
    .generateDataKey(new GenerateDataKeyRequest()
            .withKeyId(this.vaultKey)
            .withKeySpec(DataKeySpec.AES_256));
```

Figure 5.1: A sample of evidence that the nitor-vault program [303] only uses 256-bit keys to encrypt data in its source code.

To **prepare** for the review, an internal compliance officer requests *evidence* that the program uses 256-bit keys. Each engineering team must take time to respond to this request. Typically, the developers search the codebase for encryption keys, API usages, and related code. The evidence they provide is screenshots like fig. 5.1 or links into their codebase.

At the time of the **review**, the human auditor randomly samples these code snippets and checks the selected snippets manually. If the auditor has a concern about the code, they contact the engineering team responsible and question them about the code. If the engineers are unable to satisfy the auditor, then the auditor refuses to certify compliance. This process is dependent on the auditor's judgment and trust in the engineering teams—the auditor only examines a small part of the code directly.

5.2.2 Audit workflow with continuous compliance

While developers **develop** software, they write and maintain lightweight machine-checked specifications of its behavior. In a case study at AWS, these specifications consisted of 9 annotations across 107,628 lines of code (section 5.8.1.1). The verification tool runs on every commit and, optionally, every time the developer compiles the code. If the tool issues a warning, the developer examines it. If the warning is a true positive—that is, the code is incorrect—the developer fixes the code. If the warning is a false positive, the developer suppresses the warning and writes a brief explanation as a code comment, which creates an easily searchable audit trail in the code. Suppressing a warning was necessary only once in over 68 million lines of source code at AWS (see section 5.8).

No action is needed to **prepare** for a review.

At the time of the **review**, the auditor rejects the code if the verification tool outputs any warnings. If developers suppressed any warnings, the auditor inspects the code near the suppressed warning (they are automatically searchable). The auditor can also check the implementation of the verification tool, which is very short, changes rarely, and is publicly available. In our experience, auditors are willing to accept that the tool is part of the trusted computing base, in much the same way that they do not inspect the compiler.

5.3 Continuous compliance controls

We have implemented verification tools for the following controls.

```
Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
cipher.init(Cipher.ENCRYPT_MODE, mySecretKey);
return cipher.doFinal(message);
```

Figure 5.2: Code example for encrypting a message. A common compliance requirement is that the algorithm name that is passed to `Cipher.getInstance` must be FIPS compliant [2].

5.3.1 Cryptographic key length

The PCI DSS and other compliance regimes require strong encryption keys to be used. In practice, a control used for this requirement is that encryption keys must be sufficiently long. Our analysis handles 4 key-generation libraries.

For `javax.crypto.spec`, a `SecretKeySpec` object may be constructed using a length parameter ≥ 32 (since it is specified in bytes) or a byte array that is at least 32 bytes long.

For `java.security.SecureRandom`, the `nextBytes(byte[])` method must be passed an array of at least 32 bytes, and `next(int)` must be passed an integer ≥ 256. Both methods are often used to generate keys.

For `org.bouncycastle.crypto`, every `KeyGenerationParameters` object must be constructed with a `strength` argument that is ≥ 256.

For AWS's Key Management Service (KMS) [60], a data key must be at least 256 bits long. A client sets the size of the generated data key by calling methods on a "key request object":

- call `withNumberOfBytes(int)` with a value ≥ 32, or
- call `withDataKeySpec(String)` with the string `"AES_256"`, or
- call `withDataKeySpec(DataKeySpec)` with `DataKeySpec.AES_256`.

5.3.2 Cryptographic algorithms

Another common requirement in compliance regimes is the use of *strong cryptographic algorithms* [240]. Figure 5.2 shows a use of encryption in Java. A compliance control for this code is that the string passed into the JCE method `Cipher.getInstance` must be on an *allow list* from the compliance regime [2, 22].

AWS had previously written a lexical analysis to validate uses of cryptographic APIs, but it was not sufficient. In figs. 5.1 and 5.2, a literal is the argument to a key-generation routine, but this was rarely the case at AWS, whose default style guide suggests the use of static final fields. These fields are not necessarily in the same class as the method call, and the values can be held in variables and passed around the program. Another failed attempt at AWS was to search for all string literals in the program and reject the program if any literal string was not on the compliance allow list. This suffered too many false positives that required human examination, because different algorithms are permitted for different uses. These issues motivated the need for a semantic analysis like ours.

5.3.3 Web requests

PCI DSS requirement 4.1 [240] mandates that communication across open networks be encrypted; other compliance regimes have similar requirements. A common control for these requirements is that web requests be made over HTTPS rather than over HTTP. In practice, this control is satisfied in Java code by checking that strings passed to the URL constructor start with "https". A syntactic check is insufficient: a URL might be constructed by concatenating several variables, or might be stored in a field far from its use.

5.3.4 Cloud data store initialization

Data subject to compliance requirements is sometimes stored in the cloud. Even if the cloud provider has the appropriate compliance certification, there are often additional controls on how cloud services are used.

For example, third-party guidelines for HIPAA-compliant use of Amazon S3 [13, 229], a popular object storage service, include:

- new buckets must not be, and cannot become, world-readable,
- new buckets must be encrypted, and
- new buckets are versioned, so that data is not lost if overwritten.

Enforcing these guidelines requires checking that the corresponding setter methods of the builder used to construct the bucket are called, and that their arguments are certain constant values.

5.3.5 Hard-coded credentials

Credentials—passwords, cryptographic keys, etc.—must not be hard-coded in source code. The PCI DSS has an entire section (§8) devoted to requirements on passwords [240]. Hard-coded credentials violate several of these requirements: that passwords must be unreadable during storage and transmission (§8.2.1) and that credentials not be shared between multiple users (§8.5).

Our analysis handles these APIs:

- In the `java.security` package, `SecureRandom` must not be initialized with a hard-coded seed. `KeyStore`'s `store` and `load` methods must not use a hard-coded password.
- In the `javax.crypto.spec` package, these must not be hard-coded: `SecretKeySpec`'s `key` parameter, `PBEKeySpec`'s `password` parameter, `PBEParameterSpec`'s `salt` parameter, and `IvParameterSpec`'s `iv` parameter.

5.3.6 Other controls

Our vision for continuous compliance—that is, automated checking of source-code compliance properties—is broad. The above are just a few examples of controls that can be enforced using continuous compliance. We believe that any compliance requirement currently controlled by manual audits of source code could be automated using our proposed approach of lightweight verification tools. The audit procedure is designed to be tractable for a human unfamiliar with the source code, so the property to be checked is usually simple and local—which both make it likelier to be amenable to program analysis. Two further examples that we have prototyped are that data must be encrypted at rest (that is, when stored on disk as opposed to in RAM) and data must be protected by a checksum.

The procedure to implement a new analysis (which we followed for the above) is: talk to the auditors, find a check they currently enforce with manual code audits, then formalize and implement it.

5.4 Technical approach

In order to satisfy the requirements of section 5.1.2, we designed dataflow analyses to perform verification.

5.4.1 Dataflow analysis via typechecking

We chose to implement each analysis as a specialized pluggable typechecker. The continuous compliance approach can be instantiated with other automated verification techniques, such as abstract interpretation or symbolic execution. We chose type-checking because it was already familiar to the Java developers at AWS. Type-checking is also modular, fast, and scalable. Pluggable type-checking is sound [128], and the proof extends directly to all the type systems in this paper.

5.4.2 An enhanced constant value analysis

Our analysis needs to estimate, for each expression in the program, whether the expression's value is any of the following:

- a single integer value.
- a single string value.
- an element of some set of values. For example, an expression might be known at compile time to evaluate to one of the strings `"aes/cbc/pkcs5padding"`, `"aeswrap"`, or `"rsa/ecb/oaeppadding"`.
- in an integer range, including an unbounded range.
- an array of a particular length, or an array whose length is an element of some set.
- a single value of some user-defined enumerated type, or an element in some restricted set of such values.
- matched by a regular expressions representing sets of strings (and our analysis supports sets of regular expressions so users do not need to write disjunctions within regexes).

A traditional constant propagation and folding analysis [306] handles the first two features. We use an enhanced constant folding and interval analysis that handles the third and fourth features [116]. We use an array indexing analysis that handles the fifth feature [176]. We made numerous bug fixes

Table 5.1: Examples of annotations from [116] that are used by our verification tool. All annotation arguments are compile-time constants.

Declaration	Meaning
`@IntVal(42) int x`	`x` has exactly the value 42
`@StringVal({"a", "b"}) String s`	`s` has the value `"a"` or `"b"`
`@IntRange(from=0, to=9) int x`	`x`'s value is in the range [0,9]
`byte @ArrayLen(32) [] a`	`a` contains exactly 32 elements

and enhancements to the existing tools to improve precision. We designed and implemented the last two features (sections 5.4.3 and 5.4.4).

Our implementation expresses abstract values as types. For example, `@IntVal({-1, 1})` is a type qualifier, and the type "`@IntVal({-1, 1}) int`" represents an integer whose run-time value is either -1 or 1; equivalently, it represents the set $\{-1, 1\}$. Table 5.1 shows the most important abstractions of the constant value analysis. Our type systems use and/or extend these abstractions. The type hierarchy appears in [116, 176]; our extensions fit in naturally.

5.4.3 Enums

To handle enums, we repurposed the existing handling of strings (the `@StringVal` annotation). Our implementation treats the enum name as the string value. This implementation approach re-uses existing logic without the need for code duplication.

5.4.4 Regular expressions

We added a new abstraction `@MatchesRegex` that expresses a possibly-infinite set of strings via a set of regular expressions. For example [268]:

```
class Cipher {
  Cipher getInstance(@MatchesRegex({"aes/gcm.*", "rsa/ecb.*"}) String algorithm);
}
```

The type of the `algorithm` parameter is `@MatchesRegex(...) String`, and it restricts the values that may be passed as arguments.

Subtyping for regular expression types is a hard problem. Subsumption for regular expressions is EXPTIME-complete [269]. Standard (but not regular) features such as backreferences make even regex *matching* NP-hard [99]. Precise subtyping for regular expression types [133, 159] is as least as hard as these problems. However, we need a fast, decidable algorithm that is understandable to developers. Our implementation imposes the following sound, approximate subtyping relationship (S_1 and S_2 are sets of regular expressions):

$$\frac{\texttt{@MatchesRegex(}S_1\texttt{) String} <: \texttt{@MatchesRegex(}S_2\texttt{) String}}{S_1 \subseteq S_2}$$

This approximation was always adequate in our case studies.

A type qualified by `@StringVal` can be a subtype of one qualified by `@MatchesRegex` (s_k is a string and r_k is a regular expression):

$$\frac{\texttt{@StringVal(\{}s_1,\ldots,s_m\texttt{\}) String} <: \texttt{@MatchesRegex(\{}r_1,\ldots,r_n\texttt{\}) String}}{\forall i, \exists j : \quad s_i\texttt{.matches(}r_j\texttt{)}}$$

No other types are subtypes of `@MatchesRegex(...) String`. If another type flows to an expression with such a type (including string values not in the allow list), the tool issues a warning.

5.4.5 Type inference

We implemented a whole-program type inference tool [91] that infers types via fixpoint analysis. The Checker Framework implements local (intra-method) type inference. The type inference tool repeatedly runs a type-checker, records the results of local type inference, and applies them to the next iteration. The annotations are stored in a side file to avoid changing programmers' source code. When a fixed point is reached, the user is shown the final results of type-checking (not the contents of the side file, though they can optionally be inspected, too).

For example, consider the following program:

```
int id(int y) { return y; }
int x = 1;
id(x, ...);
```

Type inference on the possible integer values in this program would produce three `@IntVal(1)` annotations:

- one on the field `x`, because `1` is assigned to `x`,
- one on the parameter `y`, because `id` is called with `x` as an argument, and
- one on the return value of `id`, because the return value flows from the parameter

To annotate the above program, our type inference approach would take three rounds, one for each of the required annotations, because each is dependent on the previous one. Note that this type inference approach is sound, because it still runs the verifier on the annotated code: incorrect annotations produced by type inference are rejected just as incorrect annotations written by a human annotator are rejected. By the same token, inference can write overly-restrictive types, as in the example above (`id`'s parameter and return type are annotated as `@IntVal(1)`, but a human would have instead written a polymorphic specification).

Type inference is useful to auditors who otherwise would be ill-equipped to reason about source code. It also enables type systems whose annotation burden would be impractical for a human (see section 5.5.5).

5.5 *Verifying compliance controls*

This section details the verification tools we built to verify that Java programs are compliant with the controls in section 5.3. Our framing of the problem made it simple to express and implement the dataflow analyses.

```
package com.amazonaws.services.kms.model;

class GenerateDataKeyRequest {
  withKeySpec(@StringVal("AES_256") String keySpec);
  withKeySpec(@StringVal("AES_256") DataKeySpec keySpec);
  withNumberOfBytes(@IntRange(from=32) Integer numberOfBytes);
  setKeySpec(@StringVal("AES_256") String keySpec);
  setKeySpec(@StringVal("AES_256") DataKeySpec keySpec);
  setNumberOfBytes(@IntRange(from=32) Integer numberOfBytes);
}

class GenerateDataKeyWithoutPlaintextRequest {
  withKeySpec(@StringVal("AES_256") String keySpec);
  withKeySpec(@StringVal("AES_256") DataKeySpec keySpec);
  withNumberOfBytes(@IntRange(from=32) Integer numberOfBytes);
  setKeySpec(@StringVal("AES_256") String keySpec);
  setKeySpec(@StringVal("AES_256") DataKeySpec keySpec);
  setNumberOfBytes(@IntRange(from=32) Integer numberOfBytes);
}
```

Figure 5.3: Our full specification for AWS KMS. These library annotations guarantee that KMS is only invoked to generate keys that are 256 bits or more.

5.5.1 Cryptographic key length

Our key-length typechecker is just an application of our enhanced constant value analysis.

Any analysis requires a specification of library APIs. This one-time, manual process is performed by a verification engineer working with a compliance officer. Once written, the specification can be re-used until the library interface changes (which is highly unlikely) or the compliance regime is updated (which is rare).

Figure 5.3 is the full specification for the KMS API. When these restrictions on all uses of the API are enforced at compile time, no data key can be generated that is smaller than 256 bits, meeting the compliance control in section 5.3.1. The specifications for the other libraries of section 5.3.1 are similar but simpler; fig. 5.3 is the largest.

5.5.2 Cryptographic algorithms

Our cryptographic algorithm typechecker is implemented on top of the enhanced constant value analysis.

We annotated library methods that accept cryptographic algorithms as input, such as `javax.crypto.Cipher` or `java.security.Signature`, with an allow list of accepted algorithm names.

For user convenience, our tool defines `@CryptoAllowed` as an alias for `@MatchesRegex`. `@CryptoAllowed` behaves identically but makes it clear to readers that the code is cryptographically relevant.

Our tool has aliases of particular `@CryptoAllowed` annotations for each compliance regime. `@CryptoAllowedPCI`, for example, corresponds to the requirements of the PCI DSS. Each alias is defined once, by a cryptography expert and a compliance officer together. A welcome side effect of centrally-defined allow-listing annotations is that adjusting the analysis to changes in compliance requirements is easy: the regular expressions in the allow list can be updated without changing any program source code, not even type annotations.

5.5.3 Web requests

Our web request typechecker is a simple extension to constant value analysis that introduces a new annotation. `@StartsWith("x")` is syntactic sugar for `@MatchesRegex("x.*")`. For example, `@StartsWith("https://")` matches `"https://www.foo.com"` but not `"foo"` or `"http://www.foo.com"`.

5.5.4 Cloud data store initialization

To prove that a new Amazon S3 bucket is properly initialized, two kinds of facts are necessary:

- setter methods for the required properties on the `Bucket` or `BucketProps` builder object must have been called, and
- the arguments to the setter methods must be certain constants.

For example, to show that a bucket is versioned, the `versioned(boolean)` method must be called, and its argument must be `true`.

For the former, our analysis must track the set of methods that have definitely been called on the builder object, and check that the required methods are all included in the set when `build` is called. This is exactly the problem solved by the accumulation analysis in section 3.5: verifying that all required methods are called on a builder. We used the implementation from section 3.5 and wrote specifications for `Bucket` and `BucketProps`.

Our enhanced constant value analysis handles the latter.

5.5.5 Hard-coded credentials

We implemented a dataflow analysis (similar to taint tracking [162]) to track the flow of manifest literals through the program. The sources in our taint analysis are manifest literals in the program text (strings like `"abcd"`, integers like `5`, byte arrays like `{0xa, 0x1}`, etc.). The sinks are calls to the APIs in section 5.3.5. The typechecker enforces that manifest literals do not flow to the sinks.

Our type system has two type qualifiers:

- `@MaybeDerivedFromConstant` is the type of any manifest literal, and of any expression into which a manifest literal might flow. For example, `"abcd"` and `x + 1` have this type.
- `@NonConstant` is the type of any other expression in the program. It is the default qualifier, meaning that an unannotated type like `String` actually means `@NonConstant String`.

Table 5.2: Our verification tools, run on open-source projects that use relevant APIs. Ver is verified projects. TP is projects with true positives, but no false positives. T&FP is projects with both true and false positives. FP is projects with false positives, but no true positives. IE is “infrastructure errors”: projects on which do-like-javac fails. TO is timeouts (1-hour limit). Total is the total number of projects. The LoC column omits infrastructure errors and timeouts. Throughout, LoC is non-comment, non-blank lines of Java code.

API	Ver	TP	T&FP	FP	IE	TO	Total	LoC
Key Length	27%	22%	12%	9%	8%	23%	78	373K
Cryptographic Algorithms	19%	42%	8%	3%	11%	17%	237	2.4M
Web Request	56%	6%	13%	6%	0%	19%	16	6K
Cloud Data	21%	68%	0%	5%	0%	5%	19	5K
Credentials	26%	15%	15%	22%	15%	7%	304	3.0M
Total	157	176	77	82	78	84	654	5.7M
	24%	27%	12%	13%	12%	13%	100%	

`@NonConstant` is a subtype of `@MaybeDerivedFromConstant`. This means that a program may assign a non-constant value to a variable whose type is qualified with `@MaybeDerivedFromConstant`, but not vice-versa.

5.5.5.1 Using whole-program type inference

This taint-tracking type system requires substantially more user-written annotations than the preceding constant-propagation type systems, because many variables and values in programs are derived from constants.

In general, type inference for taint-tracking is difficult, because a human must first locate all the sources and all the sinks. In our case, however, the sources can be identified automatically (manifest literals in the program), and the sinks are known ahead of time (the APIs listed in section 5.3.5). The inference tool (section 5.4.5) can therefore determine whether each program element might have been derived from a constant, without the need for human intervention—that is, all required annotations can be derived automatically.

5.6 Case Study on Open-Source Software

To permit reproduction, we open-sourced our tools [84, 85, 86, 93] and applied them to open-source software. The scripts and data used for sections 5.6 and 5.7 are available at `https://doi.org/10.5281/zenodo.3976221`.

5.6.1 Methodology

For each API mentioned in section 5.3, we searched GitHub for projects that contain at least one use of the relevant API. We used all projects for which running a standard Maven or Gradle build task (`mvn compile` or `gradle compileJava`) in the root directory succeeds, under either Java 8 or Java 11.

Running our tool requires supplying a `-processor` argument to each invocation of `javac`. We augmented do-like-javac [98] for that purpose. It first runs the build system in debug mode and scans the logs for invocations of `javac`. Then, it replays those invocations, with the `-processor` command-line argument added, in the same environment—for example, after other build steps that compilation may depend on. Sometimes, replaying the build is not successful; this is reported as "infrastructure error" in table 5.2. The most common reasons are that the project's custom build logic is not idempotent, there are no observable javac commands, or the project uses javac options that are incompatible with the `-processor` flag.

To fully automate the process, we ran all verifiers with whole-program inference (section 5.4.5) enabled. We set a timeout of one hour. Our verifiers are fast, but inference might not terminate. Our typecheckers contain widening operators to prevent infinite ascending chains, but do not contain corresponding narrowing operators. In some cases, inference therefore introduces an infinite descending chain, leading to a timeout.

We manually classified each warning issued by each verifier as a true positive (a failure to conform to a compliance requirement) or a false positive (a warning issued by the tool that does not correspond to a compliance violation). We counted crashes and bugs in the Checker Framework as false positives.

5.6.2 Findings

Table 5.2 shows the results. The key takeaways of our study were:

- Much open-source software, in its default configuration, contains compliance violations. Compliance officers should review open-source software before it handles customer data.
- Our tools found true positives in more projects than they issued false positives. A major attraction of unsound bug-finding tools is that they tend to have low false-positive rates, but our sound verification tools do reasonably well (see section 5.7 for a direct comparison to bug-finding tools).

The majority (72%) of false positives are issued by the credentials checker. The relatively high rate of false positives from this checker is due to the limitations of the type inference tool (section 5.4.5): it cannot always infer the appropriate type qualifiers for type arguments (Java generics). Any time it is incorrect, the credentials checker issues a false positive.

5.6.3 Example compliance violations

Figures 5.4 and 5.5 show two examples of compliance violations:

1. An HSM (Hardware Security Module) simulator [160] uses the DES encryption algorithm (fig. 5.4). An HSM is a physical device used for managing encryption keys. Practical brute-force attacks against DES were public knowledge as early as 1998 [106].

```
if (sCommand.contains("#S#>")) {
  SecretKey sk_w_key_VNN = new SecretKeySpec(b_w_key_VNN, "DES");
  ...
}
```

Figure 5.4: An example use of an insecure encryption algorithm.

```
private static final String KEY = "j8m2gnzbvkavx7c2a94g";
...
byte[] keyBytes = KEY.getBytes("UTF-8");
MessageDigest sha = MessageDigest.getInstance("SHA-1");
keyBytes = sha.digest(keyBytes);
SecretKeySpec secretKeySpec = new SecretKeySpec(keyBytes, "AES");
```

Figure 5.5: An example use of a hard-coded key.

2. A command-line email client [271] uses a hard-coded key (fig. 5.5). The `SecretKeySpec` thus generated is used to encrypt user passwords, a major security risk.

The maintainers of these projects might not consider these compliance violations to be bugs, because they might not care about whether their projects are usable in contexts that require compliance certification, such as education, healthcare, commerce, or government work. However, if these projects were to be used in such contexts, each compliance violation would be a serious concern.

5.7 Comparison to Other Tools

We compared our tool to previous tools for preventing misuse of cryptographic APIs. Previous tools do not warn about short key lengths or misuse of cloud APIs, so our evaluation focuses on selecting cryptographic algorithms, hard-coded credentials, and the use of HTTP vs. HTTPS. The developers of CryptoGuard [258] have developed a microbenchmark set of misuses of cryptography, which they call CryptoAPIBench [8]. Their paper evaluates CryptoGuard against SpotBugs, Coverity, and CogniCrypt$_{SAST}$. We repeated their experiments, and extended them to include our verification tools, for the subset of their evaluation that our tools cover (11/16 categories of cryptographic misuse). We evaluated on two versions of the benchmark: the original and a version whose labeling of safe and unsafe code reflects compliance rules.

5.7.1 Tools compared

We compared our verifiers to four state-of-the-art tools that detect misuses of cryptographic APIs.

- SpotBugs [94] is the successor of FindBugs [19], a heuristic-based static analysis tool that uses bug patterns. Some bug patterns relate to cryptography. It is heavily used in industry. We used two versions of SpotBugs, configured differently: the standard desktop version 4.0.2 ($\text{SpotBugs}_{\text{D}}$), and version 3.1.12 configured with the ruleset from the SWAMP [290], which contains additional security bug patterns ($\text{SpotBugs}_{\text{S}}$). For both versions, we only enabled warnings in the `SECURITY` category.

- Coverity [31] is a commercial bug-finding tool. We used Coverity's free trial in April 2020 for the experiments in this section. They provided no version number.

- CogniCrypt_{SAST} [188] is a tool that checks user-written specifications (in the custom CrySL language) consisting of typestate properties, required predicates, forbidden methods, and constraints on method parameters using synchronized push-down systems. We used CrySL version 2.7.1 for these experiments, with the included JCA rules.

- CryptoGuard [258] is a bug-finding tool augmented with a slicing algorithm to allow it find more bugs. Its design emphasizes maintaining a low false positive rate while scaling to realistic programs. We built CryptoGuard from source code [92].

These tools were designed to prevent misuse of cryptography[1], not to support the compliance certification process. These two goals are related—both aim to reduce the number and cost of vulnerabilities that occur in the wild—but lead to different design choices:

- Bug-finding tools like the above four tools aim for low false positive rates (high precision, or high confidence that each reported warning is useful), even at the cost of false negatives (unsoundness) [168]. By contrast, automated compliance requires verification—no false negatives. Given an unsound tool, the code would still need to be audited by hand in case the tool missed an error. Put another way, auditors prefer sound approaches over manual examination, and they prefer manual examination over unsound tools.

- Compliance requirements can be stronger than typical developer guidelines. For example, section 5.3.1 describes the compliance requirement to use a 256-bit key. None of the above tools implements this check, so (to avoid disadvantaging those tools) we did not use it in our comparison.

5.7.2 Results

Table 5.3 shows the results of the comparison. Precision and recall are defined identically to CryptoAPIBench [8]. Our numbers differ from [8] slightly because we used newer versions of the tools. Only our verifier achieves 100% recall; the other tools are unsound.

From a compliance perspective, CryptoAPIBench misclassifies some unsafe code as safe:

- CryptoAPIBench labels 19 unsafe calls in unexecuted code, similar to fig. 5.6, as safe.

[1] The tools also have other capabilities, but our evaluation focuses on this aspect of their functionality.

Table 5.3: Comparison of tools for finding misuses of cryptographic APIs, on relevant parts of CryptoAPIBench. "Original" and "Compliance" refer to the labeling schemes described in the text.

	SpotBugs$_\mathrm{D}$	**SpotBugs$_\mathrm{S}$**	**Coverity**	**CogniCrypt**	**CryptoGuard**	**Ours**
Original						
Precision	-	0.46	0.67	0.69	0.86	0.78
Recall	0.0	0.24	0.29	0.66	0.93	1.0
Compliance						
Precision	-	0.69	1.0	0.79	1.0	0.97
Recall	0.0	0.32	0.38	0.61	0.88	1.0

- CryptoAPIBench's "insecure asymmetric encryption" requirement allows any RSA algorithm, so long as the key is not 1024 bits. Our compliance controls also specify the padding scheme because there are published attacks against the default padding scheme used by Java [39]. CryptoAPIBench labels 11 calls to `Cipher.getInstance("RSA")` that use the default padding as safe.

The "Compliance" labeling in table 5.3 reclassifies these calls to reflect compliance rules.

Overall, the results show the promise of sound approaches to detecting and preventing program errors, such as misuses of cryptography, with high precision while maintaining soundness.

5.8 Case Studies at AWS

We performed two case studies at Amazon Web Services (AWS).

In the first case study, 7 teams with a compliance requirement ran the key-length verifier (section 5.3.1) on each commit. If the verifier fails to prove compliance, their continuous delivery process is blocked. This case study shows that the verifier is robust enough to be deployed in a realistic setting, and that developers and compliance officers see enough value in it to opt into a verification tool that could block deployment.

In a second case study, we ran both the key-length verifier and the cryptographic algorithm verifier as part of large-scale security scanning infrastructure. This second case study shows that both verifiers can be easily integrated in an automated system, and that they produce high-quality findings.

5.8.1 Continuous delivery case study

This case study investigates whether a) compliance officers care about the output of our verifier, and b) developers accept a verification tool as part of their continuous integration. Some key findings of this case study were:

- The verifier reports no warnings on any of the core AWS services that were subject to compliance requirements.

```
public SecretKey getKMSKey(final int keyLength) {
  GenerateDataKeyRequest request = new GenerateDataKeyRequest();
  if (keyLength == 128) {
    request.withKeySpec(DataKeySpec.AES_128);
  } else {
    request.withKeySpec(DataKeySpec.AES_256);
  }
  // set other parameters...
  GenerateDataKeyResponse response = awsKMS.generateDataKey(request);
  ...
}
```

Figure 5.6: Code from a service with a code path that could have been used to generate a 128-bit key.

- Old manual audit workflows missed compliance-relevant code.
- Using verification tools saved time and effort for developers.
- Developers who were initially skeptical of formal verification technology were convinced of its value by our tool's ease of use and effectiveness.

5.8.1.1 Results

The key-length verifier was easy to use. Developers had to write only 9 type qualifiers in 107,628 lines of code: 3 `@StringVal` annotations, 4 `@IntVal` annotations, and 2 `@IntRange` annotations. The tool issued only 1 warning that the compliance officers did not consider a true positive. This was an easy decision for them: the code was manifestly not compliance related. We determined that it was caused by the Checker Framework's overly-conservative polymorphic (that is, Java generics) type inference algorithm [217].

While running the verifiers, developers found several services that were compliant but error-prone or confusing. As one example, consider the code in fig. 5.6. This code can generate a 128-bit key, but its clients never cause it to do so. A developer verified this fact by changing the type of the `keyLength` parameter from `int` to `@IntVal(256) int` and running our verification tool. It verified every client codebase, proving that the `keyLength == 128` codepath is not used. Without a verification tool like ours that can run on each commit, the presence of such code paths, even if unused, is dangerous: a developer might change client code, or write new client code, without considering the compliance requirement. Our tool allows developers to discover unsafe code paths, and also to be certain that they are not being used when they are discovered.

At the time of writing, the continuous integration job has run 1426 times and has issued a warning 3 times, each of which was quickly fixed. The small number of failures is probably because most developers run it on their local machines before committing. We do not know how many of those local runs have revealed a problem with the code.

Another discovery while typechecking was that four services had provided incomplete evidence to auditors: the evidence did not cover every part of their codebase that generated encryption keys. Developers explained that they had not realized that those parts of the code were compliance-relevant. By contrast, our verifier checks all of the code. The external auditors were particularly excited by this finding: one said that "it eliminates a lot of the trust" that auditors previously needed to have in engineering teams to provide them with complete evidence.

External auditors were excited to be on the cutting edge of automation for compliance: they can advertise as providing higher assurance than other auditors, and their costs go down. The AWS internal compliance officers can continuously monitor compliance via continuous integration jobs triggered on every commit.[2]

AWS encourages its customers, and providers of third-party services, to use these tools [311].

5.8.1.2 Developers' reactions

We began rolling out our verification tools to compliance-relevant services at AWS in September 2018. To our surprise, we encountered little resistance as we began the rollout—the first team we contacted immediately integrated the key-length verifier and enabled it in their continuous integration process, and then canceled the meeting we had scheduled with them. These early-adopting developers told us that they were frustrated by compliance's ongoing cost: gathering evidence is an irritating distraction from their regular work.

Other engineering teams are also convinced. Each team saves time by not having to prepare for audits. One developer told us, "The Checker Framework solution is a great mechanism and step toward automating audit evidence requests. This has saved my team 2 hours every 6 months and we also don't have to worry about failing an audit control." (The 2-hour savings is per team, per control, for the developers alone.) The effort of onboarding a project to use a verification tool is less than the engineering cost of providing evidence for the very first audit, not to mention savings to compliance officers and the external auditor. After that, the savings accumulate.

5.8.2 Scanning-at-scale case study

In the second case study, a security team ran two of our verifiers (key-length and crypto algorithms, sections 5.5.1 and 5.5.2) on code beyond what needs to be audited. This case study demonstrates that our approach requires few developer-written annotations and that warnings often reveal interesting issues. Our verifiers are integrated into a system that scans a set of highly-used packages on a fixed schedule. Findings of these scans are reported to security engineers and triaged manually. The security team is interested in analyzers that report security-related findings that can be triaged without in-depth code knowledge, and that have a signal-to-noise ratio that is manageable by a security engineer.

Table 5.4 categorizes each package into one of four categories:

[2] They set up a second CI service, so that compliance is monitored even if the engineering team were to disable the verifier in their CI setup.

Table 5.4: Running the key length and crypto algorithm verifiers at AWS. The key length verifier is only run on packages that use the specific library routine. The crypto algorithm verifier is run on a subset of all Java code at AWS.

	Key length	**Crypto algorithms**
Verified, no annotations	215 packages	37,077 packages
Verified, annotations	23 packages	0 packages
True positive warning	15 packages	158 packages
False positive warning	1 package	0 packages
Total	254 packages	37,235 packages
	8,481,188 LoC	68,416,620 LoC

Verified, no annotations: The verifier completed successfully without any manually written annotations. If subject to a compliance regime, these codebases would be verifiable *without* any human onboarding effort.

Verified, annotations: The verifier initially issued an error on these codebases. After writing one or more type annotations in the codebase, the verifier succeeds. If subject to a compliance regime, these projects would be verifiable *with* human onboarding effort. In 23 cases (once in each of 23 packages), the call to a key generation library was wrapped by another method; a developer had to write one annotation to specify each wrapper method. Because typechecking is intra-procedural, an annotation must be placed where relevant dataflows cross procedure boundaries or enter the heap. The typechecker issues a warning if a needed annotation is missing. Thus, developers can use the tool to identify these locations. Note that developer-written annotations are checked, not trusted. The only trusted annotations are those for libraries (e.g., fig. 5.3).

True positive: The verifier issued an error that corresponds to a compliance violation, if that codebase were to be subjected to a compliance audit. The key length verifier found 15 instances of code that used 128-bit keys. The crypto algorithm verifier found 158 uses of weak or outdated crypto algorithms. AWS's internal compliance officers confirmed that none of these codebases were actually subject to audits. All true positives were examined by a security engineer to ensure that the findings were correct and that no production code was affected. The crypto algorithm verifier received positive feedback from security engineers since it is easy to configure and outperformed an existing text-based check that was running in the scanning infrastructure.

False positive: The verifier issued an error, indicating that it cannot prove a property. Manual examination determined that the code never misbehaves at run time, but for a reason that is beyond the capabilities of the verification tool. The key length verifier reported 1 false positive: the key length was hard-coded correctly, but was loaded via dependency injection, which our verifier does not precisely model.

We computed the compile-time overhead of using our tools. We randomly sampled 52 projects using the key length verifier and 87 projects using the cryptographic algorithm verifier from those in table 5.4. We recorded their run time with and without our tools. On average, our tools increased the full compile time for the project from 51 to 134 seconds (2.6×). As part of a continuous integration workflow, developers found this overhead acceptable.

5.9 Threats to Validity

Our verification tools check only some properties; a program they approve might fail unrelated compliance controls or might contain other bugs. It does not check native code, and a verified program may be linked with unverified libraries. It has modes that adopt unsoundnesses from Java, such as covariant array subtyping and type arguments in casts. Like any implementation, it may contain bugs.

Our sample programs may not be representative. We mitigated this threat by considering over 70 million lines of code from a variety of projects, but it is all Java code.

5.10 Lessons Learned

Verification is a good fit for compliance A key contribution of this work is the observation that source-code compliance is a good target for verification. Existing compliance controls are informal specifications that are already being checked by humans. These properties are relatively simple. Yet, the domain is mission-critical. Though researchers have struggled to make verification appealing to developers, we have discovered another customer for verification technology—compliance auditors.

Because controls are designed to be checked by a human unfamiliar with the source code, most are amenable to verification. There are two properties of compliance controls that make them more verification-friendly:

- the controls are usually local, so that a human can check them quickly.
- the controls are usually simple, so that a human without in-depth knowledge of the code can check them.

Both of these properties make it more likely that a given control can be automated. We believe that any compliance property that is currently checked by manual examination of source code can be automated.

Does someone else ever have to read the code? Compliance certification is an example of a *code reading* task: someone other than the developer examines the code to check for a specific property. Other code reading tasks are also amenable to automation. For example, checking the formatting of code is another code reading task which has already been automated.

Using verification tools changes developer attitudes This work has had a significant effect in changing attitudes toward verification. Developers and compliance officers started out skeptical of formal methods, but now they are enthusiastic. Equally importantly, developers on teams *not* subject to compliance requirements are observing their peers using verification. The adoption of new technology is fundamentally a social process [172], and social pressure is an important factor influencing whether security tools are adopted by practitioners [310]. We believe that simple, scalable techniques are both a research contribution and the best way to widely disseminate formal verification. We encourage other researchers who are interested in impact to deploy their tools in ways that reduce developers' workload by eliminating existing tasks that developers must perform regularly.

Verification can save time for developers When developers consider using verification technologies in isolation, they must trade off developer time (to write annotations, run the verification tool, etc.) against improved software quality. The developers we worked with at AWS are busy, and some were initially skeptical of verification. They believed that a formal verification tool would have two serious costs[3]:

- Developers would have to spend a lot of time annotating the codebase before seeing benefits from the tool.

- The verification tool would issue false positives that would waste engineering time to investigate, then rewrite the code or the annotations.

These fears were grounded in experience with formal verification tools like OpenJML [195] that are designed to prove complex properties. Because developers must already do the work to certify their software as compliant, they found the introduction of verification to automate that task a welcome change. Rather than verification becoming an extra task for them, verification *replaced* an existing, unpleasant task. We encourage other verification researchers interested in impact in practice to use verification to replace existing tasks developers must perform.

Move other non-testing tasks to continuous integration In much the same way that continuous integration improves software quality by running tests more frequently, continuous compliance increases the confidence of auditors that compliance is maintained between audits. We believe that researchers should explore whether there are other software-adjacent tasks that can be moved into the continuous integration workflow, as we have done for compliance using our verification tools.

Verification is useful for stakeholders other than programmers Compliance auditors are a non-traditional customer of verification technology. Nevertheless, we found that auditors readily accepted verification and that it fit well into their workflow. Compliance, like verification, is concerned with soundness—the cost of a failed audit is astronomical, especially for a company like AWS with many customers who must remain compliant themselves. This similarity in thinking and goals between compliance and verification made our success possible. We encourage other verification research interested in practical impact to investigate other stakeholders in the correctness of software besides the developers themselves.

[3] The developers were *not* concerned about code clutter; they were used to the benefits of annotations from using tools like Lombok, Guice, and Spring.

5.11 *Related Work: Compliance*

Practitioners and researchers recognize the current limitations of manual compliance audits, and they are actively seeking improvements. We classify previous work into manual approaches, testing, run-time checking, and static analysis.

Manual The industry-standard approach to code-level compliance is manual examination. There has been some work on improving the current manual audit approach by simplifying the software inspection process [218]. By contrast, our approach aims to replace parts of the manual process with an automated one.

Testing Most previous research on source-code level compliance has aimed to apply automated or semi-automated testing [298, 283, 26, 158]. Automated tests reduce costs and prevent mistakes made while manually executing the tests (but not those in designing and implementing the tests). However, tests are still incomplete: tests can show the presence of defects, but not their absence.

E-commerce merchants who must be compliant with the PCI DSS can use an Approved Scanning Vendor (ASV) to automatically certify that their websites meet some parts of the PCI DSS. Recent work [257] has shown that extant ASVs are unsound and mis-certify many vulnerable websites in practice. Further, most (∼86%) merchant websites have one or more "must-fix" vulnerability, showing the need for sound verification tools like ours.

Run-time checking A recent approach is "proactive" compliance, which is analogous to run-time checking. Even if run-time checks are exhaustive and correct, a violation causes the program to crash. Research in this area aims to improve run-time performance and retain interoperability with uninstrumented code [211, 40, 212].

Static analysis To our knowledge, our work is the first to use automated, sound static analysis (lightweight verification) for source-code compliance properties like those described in section 5.3.

A recent literature review split compliance automation into three categories: retroactive (i.e. log scanning), intercept-and-check (i.e. at run time, check operations for compliance), and proactive (which they describe as like intercept-and-check, but with some precomputation to reduce the run-time burden) [212]. They do not mention sound verification. Ullah et al. describe a framework for building an automated cloud security compliance tool [298]. Their framework does not include sound static analysis *per se*, but does have a place for ASVs, which they regard as best-effort bug finders. Recent work on designing a cloud service which could be continuously compliant did not consider using a verification tool to achieve that goal [204].

Formal methods like process modeling have been applied to compliance problems, especially in safety-critical domains such as railway [30] and automotive systems [20]. The COMPAS project [100] is a collection of formal approaches to business process modeling applied to compliance. Kokash and Arbab modeled processes in the Reo language and analyzed them for compliance [186]. Tran et al. developed a framework for expressing compliance requirements in a service-oriented architecture [297]. These approaches are complementary to ours. They check properties about a process or about a *model* of the system, but they give no guarantees about its source code or its implementation.

There is a wealth of specification and verification work that is *not* related to compliance requirements. For example, Pavlova et al. developed a technique for inferring JML annotations that encode security policies—a domain adjacent to compliance—of JavaCard applets [239]. Their approach utilizes the JACK proof assistant, so it is neither automated nor usable by workaday programmers or auditors. Furthermore, the security policies they check do not overlap with the requirements of compliance regimes.

Our work assumes cooperation between a developer and an auditor. A similar assumption is made by the SPARTA [113] toolkit for statically verifying that Android apps do not contain malicious information flows, which posits a hypothetical high-assurance app store. We address a different domain—compliance—and we report on wide-scale, real-world usage.

Analyzing uses of cryptography APIs Most ($> 90\%$) Java applications that use cryptography *misuse* it [63], and most ($> 80\%$) security vulnerabilities related to cryptography are due to improper usage of cryptographic APIs [193]. CogniCrypt$_{SAST}$ [188] is a technique based on synchronized push-down systems for finding unsafe uses of cryptography APIs. CryptoGuard [258] is a heuristic-based tool based on program slicing for finding unsafe uses of cryptography APIs. We compare to both in section 5.7.

5.12 Conclusion: Compliance

Compliance is an excellent domain to show that verification tools are ready for real-world deployment to solve real-world problems, especially to developers who might otherwise be skeptical of the value of verification. Lightweight verification tools like specialized typecheckers are a good fit for compliance: they provide much higher assurance than either manual audits or unsound bug-finding tools, at lower cost. Sound verifiers can be narrowly scoped to individual properties like compliance controls. This makes them simple to design and implement. It also maintains a low annotation burden, making them as easy to use as unsound bug-finding tools.

Our experience shows that verification scales to industrial software at AWS, and that the business derived significant value from our efforts. As long as verification automates work they are already doing, developers are enthusiastic about adopting it.

We look forward to a future in which lightweight verification technology is widespread—both for compliance and for correctness. Our tools—running in production at AWS for a large cohort of real developers, saving them time and effort—are a step towards realizing that goal.

BIBLIOGRAPHY

[1] *POPL '77: Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, Los Angeles, CA, January 1977.

[2] FIPS PUB 140-2, Security requirements for cryptographic modules, 2002. U.S.Department of Commerce/National Institute of Standards and Technology.

[3] *PLDI 2002: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[4] *OOPSLA 2007, Object-Oriented Programming Systems, Languages, and Applications*, Montreal, Canada, October 2007.

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

[6] Bhargav Acharya. Why cloud providers need a SOC report. `https://www.schellman.com/blog/why-cloud-providers-need-a-soc-report`, 2016. Accessed 28 March 2019.

[7] Stephen Adams, Thomas Ball, Manuvir Das, Sorin Lerner, Sriram K Rajamani, Mark Seigle, and Westley Weimer. Speeding up dataflow analysis using flow-insensitive pointer analysis. In *International Static Analysis Symposium*, pages 230–246. Springer, 2002.

[8] Sharmin Afrose, Sazzadur Rahaman, and Danfeng Yao. CryptoAPI-Bench: A comprehensive benchmark on Java cryptographic API misuses. In *2019 IEEE Cybersecurity Development (SecDev)*, pages 49–61. IEEE, 2019.

[9] 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. The KeY platform for verification and analysis of Java programs. In *VSTTE 2014: 6th Working Conference on Verified Software: Theories, Tools, Experiments*, pages 55–71, Vienna, Austria, July 2014.

[10] AICPA. SOC 2 examinations and SOC for Cybersecurity examinations: Understanding the key distinctions. `https://www.aicpa.org/content/dam/aicpa/interestareas/frc/assuranceadvisoryservices/downloadabledocuments/cybersecurity/soc-2-vs-cyber-whitepaper-web-final.pdf`, 2017. Accessed 1 February 2019.

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

[12] Domenico Amalfitano, Vincenzo Riccio, Porfirio Tramontana, and Anna Rita Fasolino. Do memories haunt you? An automated black box testing approach for detecting memory leaks in Android apps. *IEEE Access*, 8:12217–12231, 2020.

[13] Amazon Web Services, Inc. Amazon S3. `https://aws.amazon.com/s3/`, 2006. Accessed 17 April 2020.

[14] Amit Seal Ami, Kaushal Kafle, Kevin Moran, Adwait Nadkarni, and Denys Poshyvanyk. Systematic mutation-based evaluation of the soundness of security-focused Android static analysis techniques. *ACM Transactions on Privacy and Security (TOPS)*, 24(3):1–37, 2021.

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

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

[17] Matthew Arnold, Martin Vechev, and Eran Yahav. QVM: An efficient runtime for detecting defects in deployed systems. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 143–162, 2008.

[18] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI 2014: Proceedings of the ACM SIGPLAN 2014 Conference on Programming Language Design and Implementation*, pages 259–269, Edinburgh, UK, June 2014.

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

[20] Ghada Bahig and Amr El-Kadi. Formal verification of automotive design in compliance with ISO 26262 design verification guidelines. *IEEE Access*, 5:4505–4516, 2017.

[21] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. 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)*, pages 740–750, Tallinn, Estonia, August 2019.

[22] Elaine B. Barker, William C. Barker, William E. Burr, W. Timothy Polk, and Miles E. Smid. SP 800-57. Recommendation for key management, part 1: General (revised), 2007. U.S.Department of Commerce/National Institute of Standards and Technology.

[23] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005.

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

[25] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Amorim, and Michael D. Ernst. Static analysis of implicit control flow: Resolving Java reflection and Android intents. In *ASE 2015: Proceedings of the 30th Annual International Conference on Automated Software Engineering*, pages 669–679, Lincoln, NE, USA, November 2015.

[26] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the art: Automated black-box web application vulnerability testing. In *2010 IEEE Symposium on Security and Privacy*, pages 332–345. IEEE, 2010.

[27] Chris Beams. @Builder should require invoking methods associated with final fields. `https://github.com/rzwitserloot/lombok/issues/707`, 2014. Accessed 20 August 2019.

[28] Nels E Beckman, Duri Kim, and Jonathan Aldrich. An empirical study of object protocols in the wild. In *European Conference on Object-Oriented Programming*, pages 2–26. Springer, 2011.

[29] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. DeFlaker: Automatically detecting flaky tests. In *International Conference on Software Engineering (ICSE)*, pages 433–444. IEEE, 2018.

[30] Cinzia Bernardeschi, Alessandro Fantechi, Stefania Gnesi, Salvatore Larosa, Giorgio Mongardi, and Dario Romano. A formal verification environment for railway signaling system design. *Formal Methods in System Design*, 12(2):139–161, 1998.

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

[32] Jeremy Bicha and Nancy Alvine. CVE-2018-15869: –owners flag isn't mandatory. `https://github.com/aws/aws-cli/issues/3629`, 2018. Accessed 5 June 2019.

[33] Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. *ACM SIGSOFT Software Engineering Notes*, 30(5):217–226, 2005.

[34] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. *ACM SIGPLAN Notices*, 42(10):301–320, 2007.

[35] Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical API protocol checking with access permissions. In *European Conference on Object-Oriented Programming*, pages 195–219. Springer, 2009.

[36] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The essence of computation*, pages 85–108. Springer, 2002.

[37] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. *SIGPLAN Not.*, 38(5):196–207, May 2003.

[38] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating code comments to procedure specifications. In *ISSTA 2018, Proceedings of the 2018 International Symposium on Software Testing and Analysis* [5], pages 242–253.

[39] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS#1. In *Annual International Cryptology Conference*, pages 1–12. Springer, 1998.

[40] Sören Bleikertz, Carsten Vogel, Thomas Groß, and Sebastian Mödersheim. Proactive security analysis of changes in virtualized infrastructures. In *Proceedings of the 31st annual computer security applications conference*, pages 51–60. ACM, 2015.

[41] Eric Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 5–14. IEEE, 2010.

[42] Eric Bodden. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *International Workshop on State of the Art in Java Program Analysis*, SOAP '12, page 3–8, New York, NY, USA, 2012. Association for Computing Machinery.

[43] Eric Bodden. Static flow-sensitive & context-sensitive information-flow analysis for software product lines: position paper. In *Workshop on Programming Languages and Analysis for Security*, pages 1–6, 2012.

[44] Eric Bodden, Laurie Hendren, and Ondřej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *European Conference on Object-Oriented Programming*, pages 525–549. Springer, 2007.

[45] Eric Bodden, Patrick Lam, and Laurie Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 36–47, 2008.

[46] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 241–250. IEEE, 2011.

[47] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating array bounds checks on demand. In *PLDI 2000: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 321–333, Vancouver, BC, Canada, June 2000.

[48] Grady Booch. *Object Oriented Design with Applications.* Benjamin/Cummings, 1991.

[49] Olivier Bouissou, Eric Conquet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Khalil Ghorbal, Eric Goubault, David Lesens, Laurent Mauborgne, Antoine Miné, et al. Space software validation using abstract interpretation. In *The International Space System Engineering Conference: Data Systems in Aerospace-DASIA 2009*, volume 1, pages 1–7. European Space Agency, 2009.

[50] Kevin Bourrillion and Éamonn McManus. AutoValue. `https://github.com/google/auto/tree/master/value`, 2019. Accessed 14 August 2019.

[51] Kevin Bourrillion and Éamonn McManus. AutoValue: How do I specify a default value for a property? `https://github.com/google/auto/blob/master/value/userguide/builders-howto.md#default`, 2019. Accessed 14 August 2019.

[52] Kevin Bourrillion and Éamonn McManus. AutoValue: How do I validate property values? `https://github.com/google/auto/blob/master/value/userguide/builders-howto.md#-validate-property-values`, 2019. Accessed 14 August 2019.

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

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

[55] Kari Ann Briski, Poonam Chitale, Valerie Hamilton, Allan Pratt, Brian Starr, Jim Veroulis, and Bruce Villard. Minimizing code defects to improve software quality and lower development costs. *Development Solutions White Paper. IBM.*, 2008.

[56] Jan Brodda. Comment on "Mark fields as required for Builder". `https://github.com/rzwitserloot/lombok/issues/1043#issuecomment-405509087`, 2018. Accessed 12 August 2019.

[57] Dan Brotherston, Werner Dietl, and Ondřej Lhoták. Granullar: Gradual nullable types for Java. In *CC 2017: 26th International Conference on Compiler Construction*, pages 87–97, Austin, TX, USA, February 2017.

[58] Christian Brunotte. Mark fields as required for Builder. `https://github.com/rzwitserloot/lombok/issues/1043`, 2016. Accessed 20 August 2019.

[59] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[60] Matthew Campagna. AWS Key Management Service cryptographic details, 2015.

[61] João Campos. Comment on "Mark fields as required for Builder". `https://github.com/rzwitserloot/lombok/issues/1043#issuecomment-389344262`, 2018. Accessed 12 August 2019.

[62] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 45–58, 2009.

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

[64] Yoonsik Cheon and Gary T Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *European Conference on Object-Oriented Programming*, pages 231–255. Springer, 2002.

[65] Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 480–491, 2007.

[66] Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. In *PLDI 2005: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 85–95, Chicago, IL, USA, June 2005.

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

[68] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP 2000: Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, Montreal, Canada, September 2000.

[69] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification.* Springer, Berlin, Heidelberg, 2013.

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

[71] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *ESOP 2007: 16th European Symposium on Programming*, pages 520–535, Braga, Portugal, March 2007.

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

[73] Patrick Cousot and Radhia Cousot. 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 [1], pages 238–252.

[74] Ziying Dai, Xiaoguang Mao, Yan Lei, Xiaomin Wan, and Kerong Ben. Resco: Automatic collection of leaked resources. *IEICE TRANSACTIONS on Information and Systems*, 96(1):28–39, 2013.

[75] Al Danial. *cloc*, Accessed June 7, 2018. `http://cloc.sourceforge.net/`.

[76] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 57–68, 2002.

[77] Oege De Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble. .QL: Object-oriented queries made easy. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 78–133, Braga, Portugal, July 2007. Springer.

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

[79] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

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

[81] Markus Degen, Peter Thiemann, and Stefan Wehr. Tracking linear and affine resources with Java(X). In *European Conference on Object-Oriented Programming*, pages 550–574. Springer, 2007.

[82] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 59–69, 2001.

[83] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended Static Checking. SRC Research Report 159, Compaq Systems Research Center, December 18, 1998.

[84] aws-kms-compliance-checker Developers. awslabs/aws-kms-compliance-checker. `https://github.com/awslabs/aws-kms-compliance-checker`, 2020. Accessed 11 August 2020.

[85] awslabs/aws-crypto-policy-compliance-checker Developers. awslabs/aws-crypto-policy-compliance-checker. `https://github.com/awslabs/aws-crypto-policy-compliance-checker`, 2020. Accessed 11 August 2020.

[86] bucket-complaince-checker Developers. kelloggm/bucket-compliance-checker. `https://github.com/kelloggm/bucket-compliance-checker`, 2020. Accessed 11 August 2020.

[87] Checker Framework Developers. Called Methods Checker for the builder pattern and more. `https://checkerframework.org/manual/#called-methods-checker`. Accessed 6 May 2022.

[88] Checker Framework Developers. The Checker Framework manual: Custom pluggable types for Java. `https://checkerframework.org/manual/`. Accessed 1 June 2022.

[89] Checker Framework Developers. Index Checker for sequence bounds (arrays and strings). `https://checkerframework.org/manual/#index-checker`. Accessed 6 May 2022.

[90] Checker Framework Developers. Resource Leak Checker for must-call obligations. `https://checkerframework.org/manual/#resource-leak-checker`. Accessed 6 May 2022.

[91] Checker Framework Developers. Whole-program inference. `https://checkerframework.org/manual/#whole-program-inference`, 2020. Accessed 17 April 2020.

[92] CryptoGuard Developers. CryptoGuardOSS/cryptoguard. `https://github.com/CryptoGuardOSS/cryptoguard/commit/2898b5b5ec25d94bbedda271638385c0fa6e0c9c`, 2020. Accessed 26 April 2020.

[93] no-literal-checker Developers. kelloggm/no-literal-checker. `https://github.com/kelloggm/no-literal-checker`, 2020. Accessed 11 August 2020.

[94] SpotBugs Developers. SpotBugs. `https://spotbugs.github.io/`, 2020. Accessed 24 April 2020.

[95] The WALA Developers. T.J. Watson Libraries for Static Analysis (WALA). `https://github.com/wala/WALA`, 2020. Accessed 26 May 2020.

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

[97] Isil Dillig, Thomas Dillig, Eran Yahav, and Satish Chandra. The CLOSER: automating resource management in Java. In *International symposium on Memory management*, pages 1–10, 2008.

[98] do-like-javac Developers. do-like-javac. `https://github.com/SRI-CSL/do-like-javac`, 2020. Accessed 24 April 2020.

[99] Mark Jason Dominus. Perl regular expression matching is NP-hard, April 2001. `https://perl.plover.com/NPC/`.

[100] Schahram Dustdar. COMPAS: Compliance-driven models, languages, and architectures for services: Publishable summary. `https://cordis.europa.eu/docs/projects/cnect/5/215175/080/reports/001-publishablesummarylongversion1.pdf`, 2010. Accessed 4 April 2019.

[101] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering*, pages 411–420, 1999.

[102] Matthew B. Dwyer, Madeline Diep, and Sebastian Elbaum. Reducing the cost of path property monitoring through sampling. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 228–237. IEEE, 2008.

[103] Matthew B. Dwyer and Rahul Purandare. Residual dynamic typestate analysis exploiting static analysis: results to reformulate and reduce the cost of dynamic analysis. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 124–133, 2007.

[104] Eclipse developers. Avoiding resource leaks. `https://help.eclipse.org/2020-12/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Ftasks%2Ftask-avoiding_resource_leaks.htm&cp%3D1_3_9_3`, 2020. Accessed 3 February 2021.

[105] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In *International Symposium on Foundations of Software Engineering (ISSTA)*, pages 235–245, 2014.

[106] Electronic Frontier Foundation. Cracking DES: Secrets of encryption research, wiretap politics and chip design, 1998.

[107] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. Checked C: Making C safe by extension. In *SecDev 2018: IEEE Cybersecurity Development Conference*, Cambridge, MA, USA, September 2018.

[108] Michael Emmi, Liana Hadarean, Ranjit Jhala, Lee Pike, Nicolás Rosner, Martin Schäf, Aritra Sengupta, and Willem Visser. RAPID: checking API usage for the cloud in the cloud. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1416–1426, 2021.

[109] Stacy English and Susannah Hammond. Cost of compliance 2018. `https://legal.thomsonreuters.com/content/dam/ewp-m/documents/legal/en/pdf/reports/cost-of-compliance-special-report-2018.pdf`, 2018. Accessed 26 February 2019.

[110] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: Workshop on Dynamic Analysis*, pages 24–27, Portland, OR, USA, May 2003.

[111] Michael D. Ernst. Nothing is better than the Optional type. `https://homes.cs.washington.edu/~mernst/advice/nothing-is-better-than-optional.html`, October 2016.

[112] Michael D. Ernst. *plume-lib*, Accessed June 7, 2018. `https://github.com/mernst/plume-lib`.

[113] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu. Collaborative verification of information flow for a high-assurance app store.

In *CCS 2014: Proceedings of the 21st ACM Conference on Computer and Communications Security*, pages 1092–1104, Scottsdale, AZ, USA, November 2014.

[114] Michael D. Ernst, Alberto Lovato, Damiano Macedonio, Fausto Spoto, and Javier Thaine. Locking discipline inference and checking. In *ICSE 2016, Proceedings of the 38th International Conference on Software Engineering*, pages 1133–1144, Austin, TX, USA, May 2016.

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

[116] Michael D. Ernst, Jason Xu, Suzanne Millstein, David McArthur, Vlastimil Dort, Martin Kellogg, and Paul Vines. Constant Value Checker. `https://checkerframework.org/manual/#constant-value-checker`, 2019. Accessed 10 August 2019.

[117] David Evans. Static detection of dynamic memory errors. In *PLDI '96: Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 44–53, Philadelphia, PA, USA, May 1996.

[118] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *SAC 2010: Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2103–2110, Sierre, Switzerland, March 2010.

[119] Manuel Fähndrich and K. Rustan M. Leino. Heap monotonic typestates. In *IWACO 2003: International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*, pages 58–72, Darmstadt, Germany, July 2003.

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

[121] Manuel Fähndrich and Songtao Xia. Establishing object invariants with delayed types. In *OOPSLA 2007, Object-Oriented Programming Systems, Languages, and Applications* [4], pages 337–350.

[122] Rafael Ferreira. Type-safe builder pattern in Scala. `http://blog.rafaelferreira.net/2008/07/type-safe-builder-pattern-in-scala.html`, 2008. Accessed 15 August 2019.

[123] John Field, Deepak Goyal, G Ramalingam, and Eran Yahav. Typestate verification: Abstraction techniques and complexity results. In *International Static Analysis Symposium*, pages 439–462. Springer, 2003.

[124] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology*, 17(2), 2008.

[125] James Finkle and Supriya Kurane. U.S. hospital breach biggest yet to exploit Heartbleed bug: expert. Reuters, August 2014. Accessed 14 May 2020.

[126] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *PLDI 2002: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* [3], pages 234–245.

[127] Matthew Fluet and Riccardo Pucella. Practical datatype specializations with phantom types and recursion schemes. In *ML 2005: Proceedings of the 2005 workshop on ML*, pages 211–237, Tallinn, Estonia, September 2005.

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

[129] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Transactions on Programming Languages and Systems*, 28(6):1035–1087, November 2006.

[130] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *PLDI 2002: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* [3], pages 1–12.

[131] Martin Fowler. Continuous integration. `https://martinfowler.com/articles/originalContinuousIntegration.html`, 2000. Accessed 1 June 2022.

[132] Fredrik Friis. Calling final builder step without providing required arguments. `https://github.com/rzwitserloot/lombok/issues/1202`, 2016. Accessed 20 August 2019.

[133] Alain Frisch and Luca Cardelli. Greedy regular expression matching. In *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004*, pages 618–629, Turku, Finland, July 2004.

[134] Dan Fritsche and Bhavana Sasne. Whitepaper: The costs of failing a PCI-DSS audit. `https://www.hytrust.com/wp-content/uploads/2015/08/HyTrust_Cost_of_Failed_Audit.pdf`, 2015. Accessed 18 March 2019.

[135] Asya Frumkin, Yotam M. Y. Feldman, Ondřej Lhoták, Oded Padon, Mooly Sagiv, and Sharon Shoham. Property directed reachability for proving absence of concurrent modification errors. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 209–227. Springer, 2017.

[136] Mark Gabel and Zhendong Su. Testing mined specifications. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

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

[138] Qi Gao, Wenbin Zhang, Zhezhe Chen, Mai Zheng, and Feng Qin. 2ndStrike: Toward manifesting hidden concurrency typestate bugs. *ACM Sigplan Notices*, 46(3):239–250, 2011.

[139] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of typestate-oriented programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(4):1–44, 2014.

[140] Felix Gessert, Wolfram Wingerath, Steffen Friedrich, and Norbert Ritter. NoSQL database systems: a survey and decision guidance. *Computer Science-Research and Development*, 32(3-4):353–365, 2017.

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

[142] Yossi Gil and Ori Roth. Fling — a fluent API generator. In *ECOOP 2019 — Object-Oriented Programming, 33rd European Conference*, pages 13:1–13:25, London, UK, July 2019.

[143] David Gilbert. *JFreeChart*, Accessed June 7, 2018. `https://github.com/jfree/jfreechart`.

[144] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. Automatic generation of oracles for exceptional behaviors. In *ISSTA 2016, Proceedings of the 2016 International Symposium on Software Testing and Analysis*, pages 213–224, Saarbrücken, Genmany, July 2016.

[145] Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, Mauro Pezzè, and Paolo Tonella. Search-based synthesis of equivalent method sequences. In *FSE 2014: Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*, pages 366–376, Hong Kong, November 2014.

[146] Google Inc. *Google Guava*, Accessed June 7, 2018. `https://github.com/google/guava`.

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

[148] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition.* Addison-Wesley Professional, 2014.

[149] David Greenfieldboyce and Jeffrey S. Foster. Type qualifier inference for Java. In *OOPSLA 2007, Object-Oriented Programming Systems, Languages, and Applications* [4], pages 321–336.

[150] GSA. FedRAMP security assessment framework, version 2.4. `https://www.fedramp.gov/assets/resources/documents/FedRAMP_Security_Assessment_Framework.pdf`, 2017. Accessed 31 January 2019.

[151] Chaorong Guo, Jian Zhang, Jun Yan, Zhiqiang Zhang, and Yanli Zhang. Characterizing and detecting resource leaks in Android applications. In *Automated Software Engineering (ASE)*, pages 389–398. IEEE, 2013.

[152] Christian Haack and Erik Poll. Type-based object immutability with flexible initialization. In *European Conference on Object-Oriented Programming*, pages 520–545. Springer, 2009.

[153] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *ICSE 2006, Proceedings of the 28th International Conference on Software Engineering*, pages 232–241, Shanghai, China, May 2006.

[154] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE 2002, Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, Orlando, Florida, May 2002.

[155] Reed Hastings and Bob Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *USENIX: Proceedings of the Winter 1992 USENIX Conference*, pages 125–138, San Francisco, CA, USA, January 1992.

[156] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In *SOSP 2015, Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 1–17, Monterey, CA, USA, October 2015.

[157] Graham Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 3(1):326–336, 1952.

[158] Hossein Homaei and Hamid Reza Shahriari. Athena: A framework to automatically generate security test oracle via extracting policies from source code and intended software behaviour. *Information and Software Technology*, 107:112–124, 2019.

[159] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems*, 27(1):46–90, January 2005.

[160] hsm-simulator Developers. gjyoung1974/hsm-simulator. `https://github.com/gjyoung1974/hsm-simulator/blob/432b2b6e9fd63936347293743e54a8e572367fda/src/com/goyoung/crypto/hsmsim/commands/crypto/GenerateVISAWorkingKey.java`, 2019. Accessed 5 May 2020.

[161] Jeff Huang, Qingzhou Luo, and Grigore Rosu. GPredict: Generic predictive concurrency analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 847–857. IEEE, 2015.

[162] Wei Huang, Yao Dong, and Ana Milanova. Type-based taint analysis for Java web applications. In *International Conference on Fundamental Approaches to Software Engineering*, pages 140–154. Springer, 2014.

[163] Laurent Hubert, Thomas Jensen, Vincent Monfort, and David Pichardie. Enforcing secure object initialization in Java. In *ESORICS 2010: Proceedings of the 15th European Symposium on Research in Computer Security*, pages 101–115, Athens, Greece, September 2010.

[164] Infer developers. Resource leak in Java. `https://fbinfer.com/docs/checkers-bug-types#resource-leak-in-java`, 2021. Accessed 4 February 2021.

[165] JetBrains. List of Java inspections. `https://www.jetbrains.com/help/idea/list-of-java-inspections.html#resource-management`, 2020. Accessed 5 February 2021.

[166] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX 2002: Proceedings of the 2002 USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, USA, June 2002.

[167] Dongyun Jin, Patrick O'Neil Meredith, Dennis Griffith, and Grigore Rosu. Garbage collection for monitoring parametric properties. *ACM SIGPLAN Notices*, 46(6):415–424, 2011.

[168] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *ICSE 2013, Proceedings of the 35th International Conference on Software Engineering*, pages 672–681, San Francisco, CA, USA, May 2013.

[169] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):596–619, 1983.

[170] Pallavi Joshi and Koushik Sen. Predictive typestate checking of multithreaded Java programs. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 288–296. IEEE, 2008.

[171] Arash Kamangir. Using Lombok to create builders for classes with required and optional attributes. `https://stackoverflow.com/questions/54155315/using-lombok-to-create-builders-for-classes-with-required-and-optional-attribute`, 2019. Accessed 20 August 2019.

[172] Elena Karahanna, Detmar W Straub, and Norman L Chervany. Information technology adoption across time: a cross-sectional comparison of pre-adoption and post-adoption beliefs. *MIS quarterly*, pages 183–213, 1999.

[173] Audrey Katcher. Understanding how users would make use of a SOC2 report. `https://www.rubinbrown.com/soc2_user_document_111710.pdf`, 2019. Accessed 28 March 2019.

[174] Martin Kellogg. Fix incorrect return value in DefaultKeyedValues2D#getRowIndex. `https://github.com/jfree/jfreechart/pull/78`, 2018. Accessed 27 April 2022.

[175] Martin Kellogg. DynamoDB Request Checker. `https://github.com/kelloggm/dynamodb-request-checker`, 2019. Accessed 20 May 2020.

[176] Martin Kellogg, Vlastimil Dort, Suzanne Millstein, and Michael D. Ernst. Lightweight verification of array indexing. In *ISSTA 2018, Proceedings of the 2018 International Symposium on Software Testing and Analysis* [5], pages 3–14.

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

[178] Martin Kellogg, Martin Schäf, Serdar Tasiran, and Michael D. Ernst. Continuous compliance. In *ASE 2020: Proceedings of the 35th Annual International Conference on Automated Software Engineering*, pages 511–523, Melbourne, Australia, September 2020.

[179] Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. Artifact for "Accumulation analysis". `https://doi.org/10.5281/zenodo.5771196`, December 2021.

[180] Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. Lightweight and modular resource leak verification. In *ESEC/FSE 2021: The ACM 29th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 181–192, Athens, Greece, August 2021.

[181] Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. Accumulation analysis. In *ECOOP 2022 — Object-Oriented Programming, 33rd European Conference*, pages 10:1–10:31, Berlin, Germany, June 2022.

[182] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[183] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. 2018.

[184] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *Symposium on Operating systems principles (SOSP)*, pages 207–220, 2009.

[185] Naoki Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 416–428, 2009.

[186] Natallia Kokash and Farhad Arbab. Formal behavioral modeling and compliance analysis for service-oriented systems. In *International Symposium on Formal Methods for Components and Objects*, pages 21–41. Springer, 2008.

[187] Goh Kondoh and Tamiya Onodera. Finding bugs in Java Native Interface programs. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 109–118, 2008.

[188] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. CrySL: An extensible approach to validating the correct usage of cryptographic APIs. In *ECOOP 2018 — Object-Oriented Programming, 32nd European Conference*, pages 10:1–10:27, Amsterdam, Netherlands, July 2018.

[189] Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 17–32, 2002.

[190] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. iDFlakies: A framework for detecting and partially classifying flaky tests. In *Conference on Software Testing, Validation and Verification (ICST)*, pages 312–322. IEEE, 2019.

[191] William Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification. In *POPL '91: Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 93–103, Orlando, FL, January 1991.

[192] Vincent Laviron and Francesco Logozzo. SubPolyhedra: a family of numerical abstract domains for the (more) scalable inference of linear inequalities. *Software Tools for Technology Transfer*, 13(6):585–601, 2011.

[193] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why does cryptographic software fail?: a case study and open problems. In *Asia-Pacific Workshop on Systems*, page 7. ACM, 2014.

[194] Wei Le and Mary Lou Soffa. Marple: Detecting faults in path segments using automatically generated analyses. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(3):1–38, 2013.

[195] Gary T Leavens, Albert L Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[196] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. *JML Reference Manual*, May 31, 2013.

[197] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. MemFix: static analysis-based repair of memory deallocation errors for C. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 95–106, 2018.

[198] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR 2010: Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Dakar, Senegal, April 2010.

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

[200] K. Rustan M. Leino and Clément Pit-Claudel. Trigger selection strategies to stabilize program verifiers. In *CAV 2016: 28th International Conference on Computer Aided Verification*, pages 361–381, Toronto, Canada, July 2016.

[201] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. Efficient unit test case minimization. In *ASE 2007: Proceedings of the 22nd Annual International Conference on Automated Software Engineering*, pages 417–420, Atlanta, GA, USA, November 2007.

[202] Xavier Leroy et al. The CompCert verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt*, 53, 2012.

[203] Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. Program tailoring: Slicing by sequential criteria. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[204] Sebastian Lins, Stephan Schneider, and Ali Sunyaev. Trust is good, control is better: Creating secure clouds by continuous auditing. *IEEE Transactions on Cloud Computing*, 6(3):890–903, 2018.

[205] Yepang Liu, Jue Wang, Lili Wei, Chang Xu, Shing-Chi Cheung, Tianyong Wu, Jun Yan, and Jian Zhang. DroidLeaks: a comprehensive database of resource leaks in Android apps. *Empirical Software Engineering*, 24(6):3435–3483, 2019.

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

[207] David Lo, Nachiappan Nagappan, and Thomas Zimmermann. How practitioners perceive the relevance of software engineering research. In *ESEC/FSE 2015: The 10th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 415–425, Bergamo, Italy, September 2015.

[208] Francesco Logozzo and Manuel Fähndrich. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. In *SAC 2008: Proceedings of the 2008 ACM Symposium on Applied Computing*, pages 184–188, Fortaleza, Ceará, Brazil, March 2008.

[209] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *International Symposium on Foundations of Software Engineering (ICSE)*, pages 643–653, 2014.

[210] Bennett Lynch. [FEATURE] @StepBuilder. `https://github.com/rzwitserloot/lombok/issues/2055`, 2019. Accessed 20 August 2019.

[211] Suryadipta Majumdar, Yosr Jarraya, Momen Oqaily, Amir Alimohammadifar, Makan Pourzandi, Lingyu Wang, and Mourad Debbabi. Leaps: Learning-based proactive security auditing for clouds. In *European Symposium on Research in Computer Security*, pages 265–285. Springer, 2017.

[212] Suryadipta Majumdar, Taous Madi, Yosr Jarraya, Makan Pourzandi, Lingyu Wang, and Mourad Debbabi. Cloud security auditing: Major approaches and existing challenges. In *Symposium on Foundations & Practice of Security*, 11 2018.

[213] Shane Markstrum, Daniel Marino, Matthew Esquivel, Todd Millstein, Chris Andreae, and James Noble. JavaCOP: Declarative pluggable types for Java. *ACM Transactions on Programming Languages and Systems*, 32(2):1–37, January 2010.

[214] Mastercard. Site Data Protection (SDP) program, frequently asked questions. https://globalrisk.mastercard.com/wp-content/uploads/2017/03/Site-Data-Protection-SDP-Program-FAQs-1-March-2017.pdf, 2017. Accessed 18 March 2019.

[215] Micro Focus. Fortify static code analyzer. https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview, 2020. Accessed 27 May 2020.

[216] Filipe Militao, Jonathan Aldrich, and Luís Caires. Rely-guarantee protocols. In *European Conference on Object-Oriented Programming*, pages 334–359. Springer, 2014.

[217] Suzanne Millstein. Implement Java 8 type argument inference. https://github.com/typetools/checker-framework/issues/979, 2016. Accessed 17 April 2020.

[218] Deepti Mishra and Alok Mishra. Simplified software inspection process in compliance with international standards. *Computer Standards & Interfaces*, 31(4):763–771, 2009.

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

[220] MITRE. Inclusion decisions for CVE Numbering Authority (CNA) rules. https://cve.mitre.org/cve/cna/rules.html#Appendix_C_inclusion_decisions, January 2018.

[221] Kevin Most. Allow default values to be set on AutoValue builders in property default impls. https://github.com/google/auto/issues/704, 2019. Accessed 14 August 2019.

[222] MS 325E.64. Access devices; breach of security. Minnesota Statutes (2018): Chapter 325E, Section 64, 2007.

[223] Nomair A Naeem and Ondrej Lhoták. Typestate-like analysis of multiple interacting objects. *ACM Sigplan Notices*, 43(10):347–366, 2008.

[224] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Everything you want to know about pointer-based checking. In *SNAPL 2015: the Inaugural Summit oN Advances in Programming Languages*, pages 190–208, Asilomar, CA, USA, May 2015.

[225] Atsushi Nakagawa. Feature: Allow fields to be specified only via builder's constructor. https://github.com/rzwitserloot/lombok/issues/1303, 2017. Accessed 20 August 2019.

[226] Mangala Gowri Nanda, Christian Grothoff, and Satish Chandra. Deriving object typestates in the presence of inter-object references. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 77–96, 2005.

[227] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *POPL 2002: Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139, Portland, Oregon, January 2002.

[228] Andrej Nemec and Riccardo Schirone. awscli: Allows loading of an undesired AMI by setting similar image properties. `https://bugzilla.redhat.com/show_bug.cgi?id=1623095`, October 2018.

[229] Jacob Nemetz and Brett Lieblich. Dash compliance automation — S3 security controls. `https://www.dashsdk.com/docs/aws/hipaa/amazon-s3/`, 2019. Accessed 8 April 2020.

[230] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary insrumentation. In *PLDI 2007: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 89–100, San Diego, CA, USA, June 2007.

[231] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[232] Peter W O'Hearn. Incorrectness logic. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, 2019.

[233] Oracle. The try-with-resources statement (the Java tutorials). `https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html`, 2020. Accessed 24 February 2021.

[234] Serkan Özkan. CVE details. `https://www.cvedetails.com/vulnerabilities-by-types.php`, January 2018. Summary of `http://cve.mitre.org/`.

[235] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE 2007, Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, Minneapolis, MN, USA, May 2007.

[236] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with Zest. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 329–340, 2019.

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

[238] ParcPlace Systems. *ObjectWorks\Smalltalk Release 4 Users Guide*. Mountain View, CA, USA, 1990.

[239] Mariela Pavlova, Gilles Barthe, Lilian Burdy, Marieke Huisman, and Jean-Louis Lanet. Enforcing high-level security properties for applets. In *Conference on Smart Card Research and Advanced Applications*, pages 1–16. Springer, 2004.

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

[241] PCI Security Standards Council. PCI DSS v. 3.2.1 template for report on compliance. https://www.pcisecuritystandards.org/documents/PCI-DSS-v3_2_1-ROC-Reporting-Template.pdf, 2018. Accessed 4 April 2019.

[242] PCI Security Standards Council. Qualified security assessors. https://www.pcisecuritystandards.org/assessors_and_solutions/qualified_security_assessors, 2020. Accessed 14 April 2020.

[243] Hila Peleg, Sharon Shoham, Eran Yahav, and Hongseok Yang. Symbolic automata for static specification mining. In *International Static Analysis Symposium*, pages 63–83. Springer, 2013.

[244] Frank Pfenning. Dependent types in logic programming. In Frank Pfenning, editor, *Types in Logic Programming*, chapter 10, pages 285–311. MIT Press, Cambridge, MA, 1992.

[245] Scott Piper. Investigating malicious AMIs. https://summitroute.com/blog/2018/09/24/investigating_malicious_amis/, 2018. Accessed 5 June 2019.

[246] Goran Piskachev, Tobias Petrasch, Johannes Späth, and Eric Bodden. AuthCheck: Program-state analysis for access-control vulnerabilities. In *International Symposium on Formal Methods*, pages 557–572. Springer, 2019.

[247] PMD developers. CloseResource. https://pmd.github.io/pmd-6.31.0/pmd_rules_java_errorprone.html#closeresource, 2021. Accessed 4 February 2021.

[248] Ponemon Institute LLC. The true cost of compliance. https://www.ponemon.org/local/upload/file/True_Cost_of_Compliance_Report_copy.pdf, 2011. Accessed 3 April 2019.

[249] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R Gross. Statically checking API protocol conformance with mined multi-object specifications. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 925–935. IEEE, 2012.

[250] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings, Supercomputing '91*, pages 4–13, Albuquerque, New Mexico, November 18–22, 1991.

[251] Mohit Punjabi. FindBugs detecter for NonNull Lombok builder attributes. https://stackoverflow.com/questions/51324922/findbugs-detecter-for-nonnull-lombok-builder-attributes, 2018. Accessed 20 August 2019.

[252] Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. Monitor optimization via stutter-equivalent loop transformation. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 270–285, 2010.

[253] Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. Optimizing monitoring of finite state properties through monitor compaction. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 280–290, 2013.

[254] Xin Qi and Andrew C. Myers. Masked types for sound object initialization. In *POPL 2009: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 53–65, Savannah, Georgia, USA, January 2009.

[255] Feng Qian, Laurie Hendren, and Clark Verbrugge. A comprehensive approach to array bounds check elimination for Java. In *CC 2002: Compiler Construction: 11th International Conference*, pages 325–341, Grenoble, France, April 2002.

[256] Lina Qiu, Yingying Wang, and Julia Rubin. Analyzing the analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 176–186, 2018.

[257] Sazzadur Rahaman, Gang Wang, and Danfeng Yao. Security certification in payment card industry: Testbeds, measurements, and recommendations. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 481–498, 2019.

[258] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. CryptoGuard: High precision detection of cryptographic vulnerabilities in massive-sized Java projects. In *CCS 2019: Proceedings of the 21st ACM Conference on Computer and Communications Security*, pages 2455–2472, London, UK, November 2019.

[259] RCW 19.255.020. Liability of processors, businesses, and vendors. Revised Code of Washington, Title 19, Chapter 19.255, Section 19.255.020, 2010.

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

[261] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[262] Brian Robinson, Michael D. Ernst, Jeff H. Perkins, Vinay Augustine, and Nuo Li. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *ASE 2011: Proceedings of the 26th Annual International Conference on Automated Software Engineering*, pages 23–32, Lawrence, KS, USA, November 2011.

[263] Patrick M. Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *PLDI 2008: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 159–169, Tucson, AZ, USA, June 2008.

[264] Adam Ruka. The type-safe builder pattern in Java, and the Jilt library. `https://www.endoflineblog.com/type-safe-builder-pattern-in-java-and-the-jilt-library`, 2017. Accessed 15 August 2019.

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

[266] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1-2):131–170, 1996.

[267] Joseph Santino. Enforcing correct array indexes with a type system. In *FSE 2016: Proceedings of the ACM SIGSOFT 24th Symposium on the Foundations of Software Engineering*, pages 1142–1144, Seattle, WA, USA, November 2016.

[268] Martin Schaef. Example of how to whitelist crypto algorithms. `https://github.com/awslabs/aws-crypto-policy-compliance-checker/blob/master/stubs/javax.crypto.astub`, 2019. Accessed 11 August 2020.

[269] Helmut Seidl. Deciding equivalence of finite tree automata. *SIAM J. Comput.*, 19(3):424–437, June 1990.

[270] Koushik Sen. Concolic testing. In *Automated Software Engineering (ASE)*, ASE '07, page 571–572, New York, NY, USA, 2007. Association for Computing Machinery.

[271] sendmail Developers. NewSaigonSoft/sendmail. `https://github.com/NewSaigonSoft/sendmail/blob/e31d9a86c7f863c59fc51d5fd2c1b60cc4586faf/src/main/java/com/newsaigonsoft/sendmail/SecurePassword.java`, 2015. Accessed 5 May 2020.

[272] Narges Shadab. HDFS-15791. Possible Resource Leak in FSImageFormatProtobuf. `https://github.com/apache/hadoop/pull/2652`, 2021. Accessed 16 June 2021.

[273] Sharon Shoham, Eran Yahav, Stephen J Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. *IEEE Transactions on Software Engineering*, 34(5):651–666, 2008.

[274] Kamil Sopko. auto-value-step-builder. `https://github.com/sopak/auto-value-step-builder`, 2019. Accessed 14 August 2019.

[275] Johannes Späth, Karim Ali, and Eric Bodden. IDE^{al}: Efficient and precise alias-aware dataflow analysis. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–27, 2017.

[276] Roel Spilker. Answer to stack overflow question titled "Optional in Lombok". `https://stackoverflow.com/a/31674917`, 2015. Accessed 21 August 2019.

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

[278] Joshua Spoerri. Fail fast for lack of default. `https://github.com/google/auto/issues/554`, 2019. Accessed 14 August 2019.

[279] SpotBugs developers. OBL: Method may fail to clean up stream or resource. `https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html#obl-method-may-fail-to-clean-up-stream-or-resource-obl-unsatisfied-obligation`, 2021. Accessed 4 February 2021.

[280] Square, Inc. PCI compliance: What you need to know. `https://squareup.com/guides/pci-compliance`, 2017. Accessed 18 March 2019.

[281] Manu Sridharan. Possible missing packageInfo property in JavaSurfaceTransformer. `https://github.com/googleapis/gapic-generator/issues/2892`, July 2019.

[282] Stack Overflow user "jax". Required arguments with a Lombok @Builder. `https://stackoverflow.com/questions/29885428/required-arguments-with-a-lombok-builder`, 2015. Accessed 20 August 2019.

[283] Philipp Stephanow and Christian Banse. Evaluating the performance of continuous test-based cloud service certification. In *International Symposium on Cluster, Cloud and Grid Computing*, pages 1117–1126. IEEE Press, 2017.

[284] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, January 1986.

[285] Bjarne Stroustrup. 16.5, resource management. In *The design and evolution of C++*, pages 388–389. Addison-Wesley Professional, 1994.

[286] Alexander J. Summers and Peter Müller. Freedom before commitment: A lightweight type system for object initialisation. In *OOPSLA 2011, Object-Oriented Programming Systems, Languages, and Applications*, pages 1013–1032, Portland, OR, USA, October 2011.

[287] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in Plaid. *ACM SIGPLAN Notices*, 46(10):713–732, 2011.

[288] Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *POPL '77: Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages* [1], pages 132–143.

[289] Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. Making pointer analysis more precise by unleashing the power of selective context sensitivity. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–27, 2021.

[290] The SWAMP Team. Welcome to the SWAMP. `https://continuousassurance.org/`, 2020. Accessed 24 April 2020.

[291] The Apache Hadoop developers. StorageInfo.java. `https://github.com/apache/hadoop/blob/aa96f1871bfd858f9bac59cf2a81ec470da649af/hadoop-hdfs-project/hadoop-hdfs/src/main/java/org/apache/hadoop/hdfs/server/common/StorageInfo.java#L246`, 2018. Accessed 22 February 2021.

[292] The Apache ZooKeeper developers. Learner.java. `https://github.com/apache/zookeeper/blob/c42c8c94085ed1d94a22158fbdfe2945118a82bc/zookeeper-server/src/main/java/org/apache/zookeeper/server/quorum/Learner.java#L465`, 2020. Accessed 24 February 2021.

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

[294] The Rust teams. Rust programming language. `https://www.rust-lang.org/`, 2021. Accessed 30 November 2021.

[295] Emina Torlak and Satish Chandra. Effective interprocedural resource leak detection. In *International Conference on Software Engineering (ICSE)*, pages 535–544, 2010.

[296] Jesse A Tov and Riccardo Pucella. Practical affine types. *ACM SIGPLAN Notices*, 46(1):447–458, 2011.

[297] Huy Tran, Ta'id Holmes, Ernst Oberortner, Emmanuel Mulo, Agnieszka Betkowska Cavalcante, Jacek Serafinski, Marek Tluczek, Aliaksandr Birukou, Florian Daniel, Patricia Silveira, et al. An end-to-end framework for business compliance in process-driven SOAs. In *2010 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 407–414. IEEE, 2010.

[298] Kazi Wali Ullah, Abu Shohel Ahmed, and Jukka Ylitalo. Towards building an automated security compliance tool for the cloud. In *Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1587–1593. IEEE, 2013.

[299] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International conference on compiler construction*, pages 18–34. Springer, 2000.

[300] Ciske van Oosten, Anne Turner, Cynthia B. Hanson, Dyana Pearson, Ronald Tosto, and Andi Baritchi. Verizon 2018 payment security report, 2018. Accessed 26 February 2019.

[301] Steven VanRoekel. Security authorization of information systems in cloud computing environments. `https://www.fedramp.gov/assets/resources/documents/FedRAMP_Policy_Memo.pdf`, 2011. Accessed 31 January 2019.

[302] Cláudio Vasconcelos and António Ravara. From object-oriented code with assertions to behavioural types. In *Proceedings of the Symposium on Applied Computing*, pages 1492–1497, 2017.

[303] vault Developers. NitorCreations/vault. `https://github.com/NitorCreations/vault/blob/3c3ec65879c82bb353b4cf4d22898abb0b7b578f/java/src/main/java/com/nitorcreations/vault/VaultClient.java`, 2020. Accessed 8 May 2020.

[304] VISA, Inc. Data security compliance requirements for service providers. `https://usa.visa.com/dam/VCOM/download/merchants/data-security-compliance-service-providers.pdf`, 2017. Accessed 31 January 2019.

[305] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing (Harry) Xu, and Ardalan Amiri Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 389–404, 2017.

[306] Mark N Wegman and F Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, 1991.

[307] Shiyi Wei, Piotr Mardziel, Andrew Ruef, Jeffrey S. Foster, and Michael Hicks. Evaluating design tradeoffs in numeric static analysis for Java. In *ESOP 2018: 27th European Symposium on Programming*, Thessaloniki, Greece, April 2018.

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

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

[310] Jim Witschey, Olga Zielinska, Allaire Welk, Emerson Murphy-Hill, Chris Mayhorn, and Thomas Zimmermann. Quantifying developers' adoption of security tools. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 260–271. ACM, 2015.

[311] Chad Woolf, Byron Cook, and Tom McAndrew. Automate compliance verification on AWS using provable security. `https://www.youtube.com/watch?v=BbXK_-b3DTk`, 2019. Accessed 25 August 2020.

[312] Haowei Wu, Yan Wang, and Atanas Rountev. Sentinel: generating GUI tests for Android sensor leaks. In *International Workshop on Automation of Software Test (AST)*, pages 27–33. IEEE, 2018.

[313] Haowei Wu, Shengqian Yang, and Atanas Rountev. Static detection of energy defect patterns in Android applications. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 185–195, 2016.

[314] Tianyong Wu, Jierui Liu, Xi Deng, Jun Yan, and Jian Zhang. Relda2: an effective static analysis tool for resource leak detection in Android apps. In *Automated Software Engineering (ASE)*, pages 762–767. IEEE, 2016.

[315] Tianyong Wu, Jierui Liu, Zhenbo Xu, Chaorong Guo, Yanli Zhang, Jun Yan, and Jian Zhang. Light-weight, inter-procedural and callback-aware resource leak detection for Android apps. *IEEE Transactions on Software Engineering*, 42(11):1054–1076, 2016.

[316] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *PLDI '98: Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, Canada, June 1998.

[317] Xusheng Xiao, Gogul Balakrishnan, Franjo Ivančić, Naoto Maeda, Aarti Gupta, and Deepak Chhetri. Arc++: effective typestate and lifetime dependency analysis. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 116–126, 2014.

[318] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Scalable runtime bloat detection using abstract dynamic slicing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(3):1–50, 2014.

[319] Wei Xu, Daniel C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. *ACM SIGSOFT Software Engineering Notes*, 29(6):117–126, 2004.

[320] Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure summaries. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 221–234, 2008.

[321] Hengbiao Yu, Zhenbang Chen, Ji Wang, Zhendong Su, and Wei Dong. Symbolic verification of regular properties. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 871–881. IEEE, 2018.

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

[323] Hailong Zhang, Haowei Wu, and Atanas Rountev. Automated test generation for detection of leaks in Android applications. In *International Workshop on Automation of Software Test (AST)*, pages 64–70, 2016.

[324] Lu Zhang and Chao Wang. Runtime prevention of concurrency related type-state violations in multithreaded applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 1–12, 2014.

[325] Xin Zhang, Mayur Naik, and Hongseok Yang. Finding optimum abstractions in parametric dataflow analysis. *ACM SIGPLAN Notices*, 48(6):365–376, 2013.

[326] Yufeng Zhang, Zhenbang Chen, Ji Wang, Wei Dong, and Zhiming Liu. Regular property guided dynamic symbolic execution. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 643–653. IEEE, 2015.

[327] Zhiqiang Zuo. Personal communication, 2021.

[328] Zhiqiang Zuo, John Thorpe, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang, Guoqing Harry Xu, Linzhang Wang, and Xuandong Li. Grapple: A graph system for static finite-state property checking of large-scale systems code. In *EuroSys*, pages 1–17, 2019.

[329] Reinier Zwitserloot. "Mandatory" fields with @Builder. `https://github.com/rzwitserloot/lombok/wiki/FEATURE-IDEA:-%22Mandatory%22-fields-with-@Builder`, 2018. Accessed 12 August 2019.

[330] Reinier Zwitserloot and Roel Spilker. Project Lombok. `https://projectlombok.org/`, 2019. Accessed 19 April 2019.