Verifying Object Construction

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

Manli Ran
UC Riverside, USA
mran002@ucr.edu

Manu Sridharan
UC Riverside, USA
manu@cs.ucr.edu

Martin Schäf
Amazon Web Services, USA
schaef@amazon.com

Michael D. Ernst
U. of Washington, USA
mernst@cs.washington.edu

ABSTRACT

In object-oriented languages, constructors often have a combination of required and optional formal parameters. It is tedious and inconvenient for programmers to write a constructor by hand for each combination. The multitude of constructors is error-prone for clients, and client code is difficult to read due to the large number of constructor arguments. Therefore, programmers often use design patterns that enable more flexible object construction—the builder pattern, dependency injection, or factory methods.

However, these design patterns can be *too* flexible: not all combinations of logical parameters lead to the construction of well-formed objects. When a client uses the builder pattern to construct an object, the compiler does not check that a valid set of values was provided. Incorrect use of builders can lead to security vulnerabilities, run-time crashes, and other problems.

This work shows how to statically verify uses of object construction, such as the builder pattern. Using a simple specification language, programmers specify which combinations of logical arguments are permitted. Our compile-time analysis detects client code that may construct objects unsafely. Our analysis is based on a novel special case of typestate checking, *accumulation analysis*, that modularly reasons about accumulations of method calls. Because accumulation analysis does not require precise aliasing information for soundness, our analysis scales to industrial programs. We evaluated it on over 9 million lines of code, discovering defects which included previously-unknown security vulnerabilities and potential null-pointer violations in heavily-used open-source codebases. Our analysis has a low false positive rate and low annotation burden.

Our implementation and experimental data are publicly available.

CCS Concepts: • **Software and its engineering → Software verification**; **Automated static analysis**; **Data types and structures**.

Keywords: Pluggable type systems, AMI sniping, builder pattern, lightweight verification, Lombok, AutoValue

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380341

ACM Reference Format:
Martin Kellogg, Manli Ran, Manu Sridharan, Martin Schäf, and Michael D. Ernst. 2020. Verifying Object Construction. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380341

1 INTRODUCTION

This paper concerns verification of flexible object construction patterns in Java-like languages. Objects in such 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 $\{x, y, color\}$, but not $\{x, color\}$. 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*. 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[1]. Popular frameworks like Lombok [63] and AutoValue [15] 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 [31]. It was already a common design pattern in Smalltalk-80 [51]. Open-source projects that automatically generate builder classes are popular: Lombok has 8500 stars on GitHub, and AutoValue has 8200. The codebase of Amazon Web Services has over 769,000 uses of builders in non-test code, 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. 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 2.1 describes a security concern associated with improperly configured requests submitted to a public AWS API [45].

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.

Typestate analysis [61] may seem like a natural fit for verifying such a property, as it is capable of tracking changes to object state across different program points. However, setters can be invoked in any order, and accommodating all orders causes a blowup in the finite-state-machine representation used by typestate analyses. More seriously, typestate analysis can be difficult to scale to large programs, as it relies heavily on precise alias analysis [27].

Our key contribution is *accumulation analysis*, a special case of typestate analysis that can be performed modularly without an alias analysis. Verifying builder usage is an example of an accumulation analysis. An accumulation analysis is free to only do partial reasoning about aliasing, or no reasoning at all. Ignored aliases can cause imprecision and false positive warnings, but never unsoundness.

An accumulation analysis, then, can be expressed as a standard type system. We implemented our verifier, called the Object Construction Checker, as a *pluggable type system* [50] that estimates which methods have been called on an object. This formulation enables type-based verification of the builder pattern, which yields a number of advantages, including scalability, modularity, and understandability. As explained in section 7, accumulation analysis is applicable to problems beyond the builder pattern, such as dependency injection and some instances of typestate.

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

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

Figure 1: Vulnerable client code that does not properly construct a request to the DescribeImagesRequest API, resulting in a potential "AMI sniping" concern.

This paper describes the design and implementation of our type system and the Object Construction Checker. 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 AWS security vulnerability (section 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 contributions of our work are:

- the identification of three real-world problems stemming from unsafe object construction (section 2),
- accumulation analysis, a special case of typestate analysis that can be checked soundly without precise (section 3),
- an accumulation analysis for reasoning about unsafe object construction (section 4),
- an implementation of that analysis for Java (section 5), and
- an evaluation of the type system on the three problems presented in section 2 (section 6).

The paper concludes with a discussion of applications of accumulation analysis beyond the builder patterns (section 7) and a discussion of related work (section 8).

2 UNSAFE OBJECT CREATION

To motivate our work, 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 (section 2.1), and buggy usage of Lombok-generated builders (section 2.2) and AutoValue-generated builders (section 2.3). Our approach soundly detects all the problems described in this section.

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

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

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. 1). This API (fig. 2) requires clients to carefully create requests to avoid a potential operational security risk [45].

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. 1 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" [45], 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. 1.

Misuse of the API must be prevented, even though a client-side coding concern is not ordinarily eligible for a CVE [46, 49]. 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" [9]. 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 [52]. Our sound static analysis is better: it does not depend on programmers to remember to use the best practice.

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

Figure 3: 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 4: A client of the `UserIdentity` builder defined in fig. 3, from the same project. This builder use will not cause a run-time exception, because all fields whose type is `@NonNull` have been set.

2.2 Lombok builders

Lombok [66] 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[2], which includes the class in fig. 3. Figure 4 is an example of such code, from java-webauthn-server's included demo. As defined, this code 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. 4 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 [6, 16–18, 29, 38, 39, 44, 48, 53]. 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" [65].

2.3 Google AutoValue

AutoValue [12] 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 [15], which contain run-time checks to ensure that when `build()` is called on

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

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 [59]. Further, it can be difficult to discover which properties have default values and which need to be set by a client, complicating builder usage [47]. And, library upgrades can lead to run-time failures when properties in AutoValue types become required.[3]

3 MODULAR ACCUMULATION ANALYSIS

This section describes how verifying object construction is an instance of an *accumulation analysis*, a special case of typestate analysis that can be computed soundly without performing alias analysis.

When a builder's finalizer is called, every required logical argument must have been supplied to the builder. Our analysis maintains a compile-time estimate of which arguments have been provided. More specifically, our implementation estimates what methods have been called on every object. This compile-time estimate can only increase. At a call to the finalizer, if the receiver object might not satisfy the finalizer's specification, our tool issues an error.

A typestate system permits the type of an object to change as a result of operations in the program, so it is a natural candidate for expressing which logical arguments have been provided to a builder. 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`.

We define an accumulation analysis as a program analysis where the analysis abstraction is a monotonically increasing set, and some operation is legal only when the set is large enough—that is, the estimate has accumulated sufficiently many items. Accumulation analysis is a special case of typestate analysis in which (1) the order in which operations are performed does not affect what is subsequently legal, and (2) the accumulation does not add restrictions; that is, as more operations are performed, more operations become legal.

For builders, each typestate stands for a different set of logical arguments that have been provided so far. The finalizer operation is permitted in all typestates whose set is a superset of the required logical arguments. Builders therefore satisfy the definition of accumulation analysis.

We have devised a *modular* typestate analysis, for the special case of an accumulation analysis. An arbitrary typestate analysis requires alias analysis for soundness. Suppose that two `OpenedFile` references `f1` and `f2` might refer to the same file object. Calling `f1.close()` must change the estimate of the type of `f2`, or else the analysis would permit the program to perform the possibly illegal operation `f2.read()`.

This problem does not arise for an accumulation analysis, which can soundly disregard aliasing. Suppose that `a1` and `a2` are must-aliased, and their estimate of logical arguments supplied is $\{x, y\}$. The operation `a1.z()` changes `a1`'s estimate to $\{x, y, z\}$. The valid operations on the old type are a subset of the valid operations on the new type. It would be sound to update the estimate of `a2`'s type, but it is not necessary: the old estimate for `a2` remains valid, but imprecise. This imprecision might lead to false positive warnings. In our case studies, we observed a need to track aliasing created by fluent method returns to avoid false positives (section 4.3); we observed no other false positives due to aliasing.

Ignoring aliases does not mean ignoring side effects. Whenever a side effect, such as an assignment, might change the object that an expression evaluates to, the refined estimate for that expression is discarded, and the analysis uses its specification (that is, its declared type) instead.

The secondary reason that our analysis does not require whole-program analysis is that our analysis checks rather than infers method specifications. Even so, our implementation requires programmers to write few annotations, and these annotations serve as valuable machine-checked documentation. If a user wished to eliminate the source-code annotations, whole-program inference could do so without requiring a heavyweight alias analysis.

Because of its special properties, an accumulation analysis can be expressed as an ordinary flow-sensitive type system—it does not require a full typestate analysis. Our implementation is a pluggable type system, layered on top of a host language.

A pluggable type system decorates each basetype from the host programming language with a *type qualifier* that mixes in more information. Our implementation is for Java, whose type annotation syntax expresses a qualifier as a symbol preceded by `@`. For example, `@NonNull String` and `@Nullable String` are types. Our type system uses the `@CalledMethods` type qualifier. An example Java variable declaration is `@CalledMethods({"setX", "setY"}) PointBuilder b;`.

4 A TYPE SYSTEM FOR BUILDERS

This section presents our type system that guarantees required methods are always invoked on builder objects. 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` (section 4.1).
- `build` must have a specification to indicate that both `title` and `author` must have been called on its receiver (section 4.2).

Two facts allow us to treat the object construction problem with builders as an accumulation analysis:

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

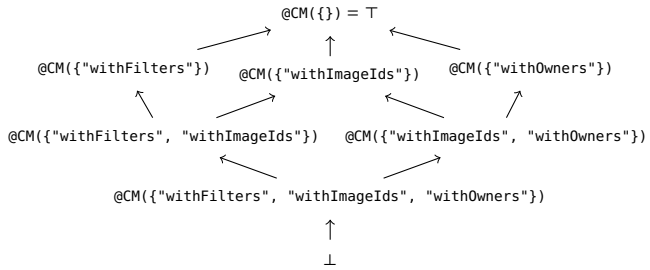

Figure 5: A type qualifier represents which methods have been called. "`@CM`" stands for `@CalledMethods`, for brevity. If an expression's type has qualifier `@CalledMethods({"withFilters", "withOwners"})`, then the methods `withFilters` and `withOwners` have definitely been called on the expression's value. Arrows represent subtyping relationships. Section 4.1 formalizes the subtyping relationship. The diagram shows a part of the type hierarchy; the full hierarchy is a lattice of arbitrary size.

- The **order** in which the client calls the setters is not important to enforce the specification on the finalizer.
- The analysis only **accumulates** method calls: it is always safe to forget that a method has been called on an object, even if it may be imprecise.

Because an accumulation analysis can verify that a client of a builder provides all required arguments, we can use a modular, flow-sensitive, pluggable type system to solve it, as detailed in the remainder of the section.

4.1 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 5 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, the Object Construction Checker verifies the project. This also guarantees that each implementation of `getRequest()` does call `withOwners()`, since the Object Construction Checker verifies, not trusts, each annotation.

4.2 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.

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.

4.3 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 4.1 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. This lack of alias reasoning can lead to false positives, as discussed in section 3.

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);
}
```

We verify `@This` annotations by ensuring the corresponding methods always return `this`.[4]

Given a call $e.m()$, the inference of section 4.1 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.[5] 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"})`.

4.4 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`) ⊓ @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 $\mid (S) \mid S \wedge S \mid S \vee S$

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

4.4.1 Using @CalledMethodsPredicate to specify the AWS API. As a practical example, the specification for the AMI sniping example (section 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.

4.4.2 Subtyping for disjunctive types.

`@CalledMethods(A)` ⊑ `@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)` ⊑ `@CalledMethodsPredicate(Q)`
If ¬(P⇒Q) is unsatisfiable.

`@CalledMethodsPredicate(P)` ⊑ `@CalledMethods(A)`
If ¬(P⇒Q) is unsatisfiable, where Q is the conjunction of the methods in A.

4.5 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 it 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")`.

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

[5] 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.

4.6 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.

5 IMPLEMENTATION

We implemented the Object Construction Checker for Java atop the Checker Framework [50]. Our implementation is 1,397 non-comment, non-blank lines of code.

The current version of our tool is available at https://github.com/kelloggm/object-construction-checker. The version of the tool used for the evaluation in section 6, including the open-source portion of our scripts and data, is publicly available at https://doi.org/10.5281/zenodo.3634993.

5.1 Limitations

Our 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 [22, 40, 50] and can be run together with the Object Construction Checker. Or, a user could use a different analysis (e.g., NullAway [5]). 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 the Object Construction Checker. Consider the AMI sniping example in section 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 [19] 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 handle guaranteeing 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.

Table 1: Detection of AMI sniping vulnerabilities.

	Open source	Closed 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 6: A true positive AMI sniping concern in Netflix's Simian-Army project.

6 EVALUATION

Our evaluation aims to answer these research questions:

- **RQ1:** Is the Object Construction Checker sufficiently scalable and effective to find previously-unknown AMI sniping attacks in real-world programs (section 6.1)?
- **RQ2:** Is the Object Construction Checker useful to programmers when they work with frameworks that provide flexible builders at the cost of compile-time checking (section 6.2)?

6.1 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 1. The Object Construction Checker 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 in section 6.3.2.

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. 6. 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.

Both false positives in the open-source experiments (cases where our type system 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.

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

public static class StartRegistrationOptionsBuilder {
   public static class MandatoryStages {
      private final StartRegistrationOptionsBuilder builder = new
           StartRegistrationOptionsBuilder();

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

Figure 7: 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 it can be safely deleted.

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.

6.2 Usefulness to programmers

There are two ways that programmers interact with the Object Construction Checker:

- 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 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 the Object Construction Checker 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 (section 6.3).
- 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 (section 6.4).

6.3 Case studies

The case studies (table 2) demonstrate the costs and benefits of onboarding an existing project. We sampled the projects from GitHub by searching for projects with significant builder usage that could compile with our infrastructure, preferring more popular projects where 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.

6.3.1 *Lombok.*

Code to force order of initialization. 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

```
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 8: 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 9: Example AutoValue builder code, adapted from google/error-prone, that sets default values.

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 7 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 were able to 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. 8. 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 [23]. The Lombok authors advocate using `null` to indicate an optional value when using Lombok builders [58], 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 code in this way.

6.3.2 *AutoValue.*

Need for annotations. The most common code pattern requiring manual annotation was setting of default values when creating a builder [13]. Figure 9 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.

Table 2: Verifying uses of the builder pattern. Throughout, "LoC" is lines of non-comment, non-blank Java code. "Annos." is number of manually-written annotations to specify existing methods. "TPs" is true positives. "FPs" is false positives, where the Object Construction Checker could not guarantee that the call was safe, but manual analysis revealed that no run-time failure was possible.

Project	Framework	LoC	Finalizer calls	LoC added	LoC removed	Annos.	TPs	FPs
Yubico/java-webauthn-server	Lombok	7,153	42	52	426	48	0	3
javagurulv/clientManagementSystem	Lombok	5,134	65	0	0	0	0	0
google/error-prone	AutoValue	74,180	9	0	0	2	0	2
googleapis/gapic-generator	AutoValue	49,054	442	2	0	58	1	1
google/nomulus	AutoValue	71,627	95	0	0	23	0	8

```
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.java"));
[...]
return packageInfo.build();
```

Figure 10: Excerpt of real bug discovered in googleapis/gapic-generator by the Object Construction Checker.

AutoValue users have discussed the difficulty of finding which properties have default values when the above pattern is used [47]. 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` (section 4.5). We believe these annotations in particular are useful documentation, as it was non-obvious in many such cases why the code was safe.

In the future, we plan to extend the Object Construction Checker to suggest these annotations to users.

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

Bug found. The Object Construction Checker found a defect in googleapis/gapic-generator (fig. 10). 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!" [60] For the one false positive in gapic-generator, a non-trivial global invariant ensures the relevant property is always set.

False positives. The Object Construction Checker 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 yet automatically support, like manually writing a builder's `build()` method with delegation to a generated `autoBuild()` method [14]. We plan to add support for such patterns in the future.

6.4 User study

To further explore the usefulness of the Object Construction Checker, we undertook a small user study.

6.4.1 Participants. Each participant was employed as a software engineer, regularly uses Java, and was familiar with Lombok. Participants were not familiar with our tool. We recruited 6 participants; all were at the same level but worked on different teams.

6.4.2 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 6.3. Participants started with a fully-annotated codebase that type-checks with the Object Construction Checker; 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.

6.4.3 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?

6.4.4 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."

6.5 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 Object Construction Checker, since they were familiar with Lombok but not with our work.

7 BEYOND BUILDERS

This paper has shown how a modular accumulation analysis can verify objects constructed via the builder pattern are well-formed. We see promise in applying accumulation analysis to other types of object construction, and to typestate properties more generally.

7.1 Setters for multiple fields

As presented in this paper, the accumulation analysis assumes that every setter operates on disjoint fields. If this assumption is violated, then the accumulation analysis should accumulate the set of fields rather than the method calls. This is how the dependency injection analysis of section 7.2 works. In Lombok and AutoValue builders, there is a one-to-one correspondence between methods called and fields set, so the accumulation analysis can accumulate the set of methods called, as a proxy for the fields set.

7.2 Dependency injection

Like the builder pattern, dependency injection is a way of creating objects that is more flexible and expressive than constructors, but also more error-prone. For example, in a framework like Guice [34], there are multiple ways to provide a logical argument:

- A class provides a single logical argument via the `@Provides` annotation on a method.
- A call to `bind(requiredArgument).to(provider)` behaves like `@Provides` in that it provides a value, but that value is obtained from elsewhere than the current class.
- Each call to `this.install(someClass)` within `configure()` provides the receiver with every logical argument of `someClass`. This may provide multiple logical arguments.
- The values required by a class are typically its logical constructor arguments. However, its superclass may impose requirements, which the class must also satisfy. The superclass may also provide values, relieving the subclass of that requirement.

The exit of `install` is the finalization point.

We believe that these, and other features of Guice and its ilk, can all be expressed as an accumulation analysis over logical arguments.

7.3 Typestate

A typestate analysis is an accumulation analysis (section 3) if the set of legal operations only grows as an object transitions through typestates. That is, if typestate TS_2 is reachable from TS_1, then $enabled(TS_2) \supseteq enabled(TS_1)$. More generally, the properties of TS_2 are stronger than those of TS_1.

Our modular accumulation analysis cannot handle cyclic typestate graphs (discounting self-loops). Doing so requires an alias analysis, and an imprecise alias analysis may lead to an unacceptable number of false positives. Few typestate examples require a cyclic graph, especially in well-structured code. `File` or `Socket` objects, e.g., are rarely closed and re-opened: new objects are created instead.

Similarly, few real-world typestate problems have complex ordering restrictions on operations. Real-world problems are often of the simple form "Always call m before n," involving a single operator (e.g., requiring a call to an initializer method). The requirement can be a longer sequence, e.g., "Call m1, then m2, and then m3." Our type system can handle such cases with an `@CalledMethods` annotation at each intermediate method to enforce the ordering.

8 RELATED WORK

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, this issue motivates some language design choices such as named and default parameters in languages like Python. The closest works are tools that generate interfaces that enforce the mandatory stages pattern (section 6.3), and only permit calls to finalizers from interfaces which have all mandatory fields set. Examples include the AutoValue Step builder [57] and the Jilt library [56]. Type-safe builders can also be encoded using phantom types [28] or in the Scala type system [26]. Recent work shows how to generate a fluent API encoding a deterministic context-free language in Java while preserving type safety [33], 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 predefined order, or both; none of them can be applied to legacy code without modifying it. Our analysis neither requires programmers to rewrite their builders nor requires methods be called in a particular order or exponentially-many classes.

Others have addressed problems in object construction. Types have been used in functional languages to enforce that unmarshalling objects is safe [35]. Specialized analyses for languages that permit mix-ins or aspects to enforce that objects under construction are not provided with conflicting method definitions also exist [8].

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 [62] 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 [37]. Delayed [25] and mask [54] 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. Their approach is designed around the internal state of an object, while ours uses externally visible properties (i.e. method calls) that correspond to how clients will actually use an object.

Typestate: Our type system can be viewed as a limited form of typestate [61] in which objects can only accumulate method calls. This limited form can be efficiently implemented without an expensive, potentially imprecise alias analysis. Our system also permits only downward refinement, whereas full typestate systems permit arbitrary changes to state. Our approach is simpler, but, as we have shown, is sufficient for the problem of constructing well-formed objects.

Fähndrich and Leino defined heap-monotonic typestates [24], which have similarities to our notion of accumulation and can also be verified without alias analysis. Their work defines heap-monotonic typestate systems as those in which "statically observable object invariants only become stronger as objects evolve." It then formalizes heap monotonicity within a general system for specifying typestates, transitions, and object invariants. Our focus is only on checking correctness of client code, not enforcing data structure invariants. Hence, we can define accumulation analysis purely in terms of available operations on an object (section 3), without formalizing its internal invariants. The Fähndrich-Leino system cannot express properties where methods may be invoked in an arbitrary order, like the required methods property for builders. Finally, the Fähndrich-Leino system was not implemented and evaluated.

Modular typestates using access control abstractions have been proposed [10]. Their system handles arbitrary typestate properties but forces programmers to reason about aliasing. Typestate specifications can be converted to working Java programs and checked if all objects with typestates are linear [41]. Using a mix of static typestate checking and dynamic typestate checks, arbitrary typestate properties can be enforced [11]. Our approach is more targeted but entirely static. Fully typestate-oriented languages have also been proposed [3], but they cannot be applied to legacy code.

Gradual typestate [64] is an unsound technique that inserts runtime checks where the static analysis cannot prove a fact. The program crashes if it attempts to perform an unsafe operation. Gradual typing is of no benefit in our context, since incorrect operations already lead to a crash. The goal of our static verification is to avoid such crashes.

There has been significant work on inferring the correct typestate model for a program based on its implementation. Both static [21, 32, 36] and dynamic [4, 20, 30, 43] approaches to this problem exist. For the builder case, the correct specification is readily apparent: all required methods must be called. Our approach is therefore complementary to these: it is concerned with efficiently enforcing properties, not inferring them.

Static Analysis for Security: Several static analyses exist that are designed to detect security problems. Coverity is a heuristic bug-finding tool that is commercially available and heavily used in industry [7] that can find some security vulnerabilities. CogniCrypt [42] and CryptoGuard [55] are tools for finding unsafe uses of cryptographic APIs; CogniCrypt is based on abstract interpretation, while CryptoGuard is based on program slicing. None of these tools contains rules for finding image sniping attacks.

9 CONCLUSION

Flexible object construction via the builder pattern is superior to manually writing constructors for complex classes in most ways. However, it has one glaring flaw: it permits any combination of logical arguments, so malformed objects that would never have been possible if all constructors were written by hand become possible. These malformed objects can lead to run-time errors or, worse, security vulnerabilities—adding a dramatic cost in bugs to the readability and flexibility benefits of builders.

We have proposed a limited form of typestate checking that only tracks which methods have been invoked on an object that verifies that legacy code using builders never produces malformed objects. Our system requires few code changes or annotations, scales to real-world Java programs, and warns programmers at compile-time about possible violations with few false positives. It found real security bugs and enthused the programmers that tested it. With our system, programmers gain all the flexibility and readability of the builder pattern, without the risk of malformed objects.

ACKNOWLEDGMENTS

Thanks to Max Willsey, Talia Ringer, Chandrakana Nandi, Don Bailey, Andy Warfield, Chriss Stephens, and the anonymous reviewers for their comments on an earlier versions of this paper.

REFERENCES

[1] 2007. *OOPSLA 2007, Object-Oriented Programming Systems, Languages, and Applications*. Montreal, Canada.
[2] 2008. *ICSE 2008, Proceedings of the 30th International Conference on Software Engineering*. Leipzig, Germany.
[3] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-oriented programming. In *OOPSLA Companion: Object-Oriented Programming Systems, Languages, and Applications*. Orlando, FL, USA, 1015–1022.
[4] Rajeev Alur, Pavol Černý, P. Madhusadan, and Wonhong Nam. 2005. Synthesis of interface specifications for Java classes. In *POPL 2005: Proceedings of the 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Long Beach, CA, USA, 98–109.
[5] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. 2019. NullAway: Practical type-based null safety for Java. In *ESEC/FSE 2019: The ACM 27th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Tallinn, Estonia, 740–750.
[6] Chris Beams. 2014. @Builder should require invoking methods associated with final fields. https://github.com/rzwitserloot/lombok/issues/707. Accessed 20 August 2019.
[7] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75.
[8] Lorenzo Bettini, Viviana Bono, and Silvia Likavec. 2005. Safe and flexible objects. In *SAC 2005: Proceedings of the 2005 ACM Symposium on Applied Computing*. Santa Fe, NM, USA, 1258–1263.
[9] Jeremy Bicha and Nancy Alvine. 2018. CVE-2018-15869: –owners flag isn't mandatory. https://github.com/aws/aws-cli/issues/3629. Accessed 5 June 2019.
[10] Kevin Bierhoff and Jonathan Aldrich. 2007. Modular typestate checking of aliased objects, See [1], 301–320.
[11] Eric Bodden. 2010. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *ICSE 2010, Proceedings of the 32nd International Conference on Software Engineering*. Cape Town, South Africa, 5–14.
[12] Kevin Bourrillion and Éamonn McManus. 2019. AutoValue. https://github.com/google/auto/tree/master/value. Accessed 14 August 2019.

[13] Kevin Bourrillion and Éamonn McManus. 2019. AutoValue: How do I specify a default value for a property? https://github.com/google/auto/blob/master/value/userguide/builders-howto.md#default. Accessed 14 August 2019.

[14] Kevin Bourrillion and Éamonn McManus. 2019. AutoValue: How do I validate property values? https://github.com/google/auto/blob/master/value/userguide/builders-howto.md#-validate-property-values. Accessed 14 August 2019.

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

[16] Jan Brodda. 2018. Comment on "Mark fields as required for Builder". https://github.com/rzwitserloot/lombok/issues/1043#issuecomment-405509087. Accessed 12 August 2019.

[17] Christian Brunotte. 2016. Mark fields as required for Builder. https://github.com/rzwitserloot/lombok/issues/1043. Accessed 20 August 2019.

[18] João Campos. 2018. Comment on "Mark fields as required for Builder". https://github.com/rzwitserloot/lombok/issues/1043#issuecomment-389344262. Accessed 12 August 2019.

[19] Checker Framework [n.d.]. *The Checker Framework Manual: Custom pluggable types for Java*. http://CheckerFramework.org/.

[20] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. 2006. Mining object behavior with ADABU. In *WODA 2006: Workshop on Dynamic Analysis*. Shanghai, China, 17–24.

[21] Guido de Caso, Victor Braberman, Diego Garbervetsky, and Sebastian Uchitel. 2013. Enabledness-based program abstractions for behavior validation. *ACM Transactions on Software Engineering and Methodology* 22, 3 (July 2013), 25:1–25:46.

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

[23] Michael D. Ernst. 2016. Nothing is better than the Optional type. https://homes.cs.washington.edu/~mernst/advice/nothing-is-better-than-optional.html.

[24] Manuel Fähndrich and K. Rustan M. Leino. 2003. Heap Monotonic Typestates. In *IWACO 2003: International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*. Darmstadt, Germany.

[25] Manuel Fähndrich and Songtao Xia. 2007. Establishing object invariants with delayed types, See [1], 337–350.

[26] Rafael Ferreira. 2008. Type-safe Builder Pattern in Scala. http://blog.rafaelferreira.net/2008/07/type-safe-builder-pattern-in-scala.html. Accessed 15 August 2019.

[27] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology* 17, 2, Article Article 9 (2008), 34 pages.

[28] Matthew Fluet and Riccardo Pucella. 2005. Practical datatype specializations with phantom types and recursion schemes. In *ML 2005: Proceedings of the 2005 workshop on ML*. Tallinn, Estonia, 211–237.

[29] Fredrik Friis. 2016. Calling final builder step without providing required arguments. https://github.com/rzwitserloot/lombok/issues/1202. Accessed 20 August 2019.

[30] Mark Gabel and Zhendong Su. 2008. Symbolic mining of temporal specifications, See [2], 51–60.

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

[32] Dimitra Giannakopoulou and Corina S. Păsăreanu. 2009. Interface generation and compositional verification in JavaPathfinder. In *FASE 2009: Fundamental Approaches to Software Engineering*. York, UK, 94–108.

[33] Yossi Gil and Ori Roth. 2019. Fling — A fluent API generator. In *ECOOP 2019 — Object-Oriented Programming, 33rd European Conference*. London, UK, 13:1–13:25.

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

[35] Grégoire Henry, Michel Mauny, Emmanuel Chailloux, and Pascal Manoury. 2012. Typing unmarshalling without marshalling types. In *ICFP 2012: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*. Copenhagen, Denmark, 287–298.

[36] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2005. Permissive interfaces. In *ESEC/FSE 2005: Proceedings of the 10th European Software Engineering Conference and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. Lisbon, Portugal, 31–40.

[37] Laurent Hubert, Thomas Jensen, Vincent Monfort, and David Pichardie. 2010. Enforcing secure object initialization in Java. In *ESORICS 2010: Proceedings of the 15th European Symposium on Research in Computer Security*. Athens, Greece, 101–115.

[38] jax. 2015. Required arguments with a lombok @Builder. https://stackoverflow.com/questions/29885428/required-arguments-with-a-lombok-builder. Accessed 20 August 2019.

[39] Arash Kamangir. 2019. 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. Accessed 20 August 2019.

[40] Martin Kellogg, Vlastimil Dort, Suzanne Millstein, and Michael D. Ernst. 2018. Lightweight verification of array indexing. In *ISSTA 2018, Proceedings of the 2018 International Symposium on Software Testing and Analysis*. Amsterdam, Netherlands, 3–14.

[41] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. 2016. Typechecking Protocols with Mungo and StMungo. In *PPDP '16: Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*. Edinburgh, UK, 146–159.

[42] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. 2018. CrySL: An extensible approach to validating the correct usage of cryptographic APIs. In *ECOOP 2018 — Object-Oriented Programming, 32nd European Conference*. Amsterdam, Netherlands, 10:1–10:27.

[43] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. 2008. Automatic generation of software behavioral models, See [2], 501–510.

[44] Bennett Lynch. 2019. [FEATURE] @StepBuilder. https://github.com/rzwitserloot/lombok/issues/2055. Accessed 20 August 2019.

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

[46] MITRE. 2018. Inclusion decisions for CVE Numbering Authority (CNA) rules. https://cve.mitre.org/cve/cna/rules.html#Appendix_C_inclusion_decisions.

[47] Kevin Most. 2019. Allow default values to be set on AutoValue builders in property default impls. https://github.com/google/auto/issues/704. Accessed 14 August 2019.

[48] Atsushi Nakagawa. 2017. Feature: Allow fields to be specified only via builder's constructor. https://github.com/rzwitserloot/lombok/issues/1303. Accessed 20 August 2019.

[49] Andrej Nemec and Riccardo Schirone. 2018. awscli: Allows loading of an undesired AMI by setting similar image properties. https://bugzilla.redhat.com/show_bug.cgi?id=1623095.

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

[51] ParcPlace Systems. 1990. *ObjectWorks\Smalltalk Release 4 Users Guide*. Mountain View, CA, USA.

[52] Scott Piper. 2018. Investigating Malicious AMIs. https://summitroute.com/blog/2018/09/24/investigating_malicious_amis/. Accessed 5 June 2019.

[53] Mohit Punjabi. 2018. FindBugs detecter for NonNull Lombok builder attributes. https://stackoverflow.com/questions/51324922/findbugs-detecter-for-nonnull-lombok-builder-attributes. Accessed 20 August 2019.

[54] Xin Qi and Andrew C. Myers. 2009. Masked types for sound object initialization. In *POPL 2009: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Savannah, Georgia, USA, 53–65.

[55] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. 2019. 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*. London, UK, 2455–2472.

[56] Adam Ruka. 2017. 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. Accessed 15 August 2019.

[57] Kamil Sopko. 2019. auto-value-step-builder. https://github.com/sopak/auto-value-step-builder. Accessed 14 August 2019.

[58] Roel Spilker. 2015. Answer to Stack Overflow question titled "Optional in Lombok". https://stackoverflow.com/a/31674917. Accessed 21 August 2019.

[59] Joshua Spoerri. 2019. Fail fast for lack of default. https://github.com/google/auto/issues/554. Accessed 14 August 2019.

[60] Manu Sridharan. 2019. Possible missing packageInfo property in JavaSurfaceTransformer. https://github.com/googleapis/gapic-generator/issues/2892.

[61] Robert E. Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering* SE-12, 1 (January 1986), 157–171.

[62] Alexander J. Summers and Peter Müller. 2011. Freedom before commitment: A lightweight type system for object initialisation. In *OOPSLA 2011, Object-Oriented Programming Systems, Languages, and Applications*. Portland, OR, USA, 1013–1032. http://doi.acm.org/10.1145/2048066.2048142

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

[64] Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. 2011. Gradual Typestate. In *ECOOP 2011 — Object-Oriented Programming, 25th European Conference*. Lancaster, UK, 459–483.

[65] Reinier Zwitserloot. 2018. "Mandatory" fields with @Builder. https://github.com/rzwitserloot/lombok/wiki/FEATURE-IDEA:-%22Mandatory%22-fields-with-@Builder. Accessed 12 August 2019.

[66] Reinier Zwitserloot and Roel Spilker. 2019. Project Lombok. https://projectlombok.org/. Accessed 19 April 2019.