Continuous Compliance

Martin Kellogg [1,*] Martin Schäf [2] Serdar Tasiran [2] Michael D. Ernst [1,2]
[1]University of Washington [2]Amazon Web Services
USA
kelloggm@cs.washington.edu,schaef@amazon.com,tasirans@amazon.com,mernst@cs.washington.edu

ABSTRACT

Vendors who wish to provide software or services to large corporations and governments must often obtain numerous certificates of compliance. Each certificate asserts that the software satisfies a compliance regime, like SOC or the PCI DSS, to protect the privacy and security of sensitive data. The industry standard for obtaining a compliance certificate is an auditor manually auditing source code. This approach is expensive, error-prone, partial, and prone to regressions.

We propose *continuous compliance* to guarantee that the codebase stays compliant on each code change using lightweight verification tools. Continuous compliance increases assurance and reduces costs.

Continuous compliance is applicable to any source-code compliance requirement. To illustrate our approach, we built verification tools for five common audit controls related to data security: cryptographically unsafe algorithms must not be used, keys must be at least 256 bits long, credentials must not be hard-coded into program text, HTTPS must always be used instead of HTTP, and cloud data stores must not be world-readable.

We evaluated our approach in three ways. (1) We applied our tools to over 5 million lines of open-source software. (2) We compared our tools to other publicly-available tools for detecting misuses of encryption on a previously-published benchmark, finding that only ours are suitable for continuous compliance. (3) We deployed a continuous compliance process at AWS, a large cloud-services company: we integrated verification tools into the compliance process (including auditors accepting their output as evidence) and ran them on over 68 million lines of code. Our tools and the data for the former two evaluations are publicly available.

CCS CONCEPTS

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

KEYWORDS

compliance, SOC, PCI DSS, FedRAMP, key length, encryption, hard-coded credentials, pluggable type systems

* Some of the work was performed while this author was an intern at AWS.

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416593

ACM Reference Format:
Martin Kellogg, Martin Schäf, Serdar Tasiran, and Michael D. Ernst. 2020. Continuous Compliance. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3324884.3416593

1 INTRODUCTION

A compliance regime like the PCI DSS [61], FedRAMP [35], or SOC [4] 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) [80]. PCI DSS certification assures card issuers that merchants will safely handle consumer credit card data [61]. Other card issuers have similar requirements [53, 71], and some U.S. states define non-compliance as a type of corporate negligence for which companies can be sued [56, 66].
- The U.S. government requires that cloud vendors be compliant with FedRAMP (Federal Risk and Authorization Management Program) [35, 78].
- Many customers of software providers expect a SOC (System and Organization Controls) report [4], which is used to evaluate how seriously potential vendors take security [2, 42].

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) [21].

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 paper focuses on requirements about the *source code*. Continuous compliance automates checking of these compliance requirements.

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) [28, 63]. A failed audit can cost millions of dollars more [34].

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 [62].

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 [77].

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

1.2 Our approach: continuous compliance

We propose *continuous compliance*, which runs a 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 [67] and continuous integration [16, 32].

Continuous compliance eliminates the need for manual audits for specific source-code controls, resolving the problems described in section 1.1. The marginal cost of an audit is negligible, because auditors accept the results of running the verifier. The opportunity for mistakes is smaller: our tools found dozens of findings of interest to compliance auditors that all prior approaches had missed, because the verifier checks the entire codebase. Regressions are caught immediately when they occur, when it is cheaper for developers to fix them [17]. 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.

1.3 Contributions

There are four main contributions of our work:

- a *conceptual* contribution: the recognition that source-code compliance is an excellent domain for the strengths and weaknesses of (some varieties of) formal verification.
- an *engineering* contribution: we designed and built five practical 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 [50, 52, 76]. 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 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 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 8.1). Both developers and compliance teams are now more receptive to formal methods than they were before: both AWS and the auditors

```
final GenerateDataKeyResult dataKey = kms
    .generateDataKey(new GenerateDataKeyRequest()
            .withKeyId(this.vaultKey)
            .withKeySpec(DataKeySpec.AES_256));
```

Figure 1: A sample of evidence that the nitor-vault program [79] only uses 256-bit keys to encrypt data in its source code.

have spoken publicly on how verification has improved their process [83]. Security and compliance teams also run verifiers on a significant fraction of code at the company on a regular schedule—the most recent run (section 8.2) scanned over 68 million lines of code and required only 23 type annotations.

2 COMPLIANCE CERTIFICATION WORKFLOW

Section 2.1 describes the state-of-the-art approach for compliance certification of source-code properties, and section 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 [61].

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.

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. 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.

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 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 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.

```
Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
cipher.init(Cipher.ENCRYPT_MODE, mySecretKey);
return cipher.doFinal(message);
```

Figure 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 [1].

3 CONTINUOUS COMPLIANCE CONTROLS

We have implemented verification tools for the following controls.

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) [19], 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`.

3.2 Cryptographic algorithms

Another common requirement in compliance regimes is the use of *strong cryptographic algorithms* [61]. Figure 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 [1, 10].

AWS had previously written a lexical analysis to validate uses of cryptographic APIs, but it was not sufficient. In figs. 1 and 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.

3.3 Web requests

PCI DSS requirement 4.1 [61] 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.

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 [5, 57], 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.

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 [61]. 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.

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.

4 TECHNICAL APPROACH

In order to satisfy the requirements of section 1.2, we designed dataflow analyses to perform verification.

4.1 Dataflow analysis via typechecking

We chose to implement each analysis as a type system. 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 [30], and the proof extends directly to all the type systems in this paper (proofs omitted for space).

Our implementation uses the Checker Framework [59], a framework for building pluggable typecheckers for Java. Our implementation handles all Java features, including polymorphism. It performs local type inference within a method body, so developers write annotations only at method boundaries, where they serve as machine-checked documentation.

As with any type system, every assignment to a variable must be from an expression whose type is a subtype of the variable's declared type. For example, when a formal parameter type has a qualifier, it is a type error if any call site's argument does not satisfy the given property.

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:

- single integer value.
- single string value.
- sets 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"`.
- integer ranges, including unbounded ones.
- estimates of array lengths, and sets of them.
- user-defined enumerated types, and sets of them.
- regular expressions to represent sets of strings, and sets of regular expressions so users do not need to write disjunctions within regexes.

A traditional constant propagation and folding analysis [81] handles the first two features. We use an enhanced constant folding and interval analysis that handles the third and fourth features [23]. We use an array indexing analysis that handles the fifth feature [44]. We made numerous bug fixes and enhancements to the existing tools to improve precision. We designed and implemented the last two features (sections 4.3 and 4.4).

Table 1: Examples of annotations from [23] 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

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 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 [23, 44]; our extensions fit in naturally.

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.

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 [68]:

```
class Cipher {
  static 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 [69]. Standard (but not regular) features such as backreferences make even regex *matching* NP-hard [26]. Precise subtyping for regular expression types [33, 37] 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):

$$\frac{S_1 \subseteq S_2}{\texttt{@MatchesRegex(}S_1\texttt{) String} <: \texttt{@MatchesRegex(}S_2\texttt{) String}}$$

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):

$$\frac{\exists i, j: \quad s_i\texttt{.matches(}r_j\texttt{)}}{\begin{array}{c}\texttt{@StringVal(\{}s_1,\ldots,s_m\texttt{\}) String} \\ <: \texttt{@MatchesRegex(\{}r_1,\ldots,r_n\texttt{\}) String}\end{array}}$$

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.

4.5 Type inference

We implemented a whole-program type inference tool [24] 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.

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: it has the same interface to the verifier as a human annotator. 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 VERIFYING COMPLIANCE CONTROLS

This section details the verification tools we built to verify that Java programs are compliant with the controls in section 3. Our framing of the problem made it simple to express and implement the dataflow analyses.

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 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 3.1. The specifications for the other libraries of section 3.1 are similar but simpler; fig. 3 is the largest.

```
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 3: Our full specification for AWS KMS. These library annotations guarantee that the keys KMS generates are 256 bits or more.

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.

5.3 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"`.

5.4 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. An accumulation analysis [45], a restricted form of typestate analysis [73] that does not require a whole-program alias analysis for soundness, can verify that all required methods are called on a builder. We used the implementation from [45] and wrote specifications for `Bucket` and `BucketProps`.

Our enhanced constant value analysis handles the latter.

5.5 Hard-coded credentials

We implemented a dataflow analysis (similar to taint tracking [39]) 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 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`.

`@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.

5.5.1 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 3.5). The inference tool (section 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.

6 CASE STUDY ON OPEN-SOURCE SOFTWARE

To permit reproduction, we open-sourced our tools [6, 7, 18, 58] and applied them to open-source software. The scripts and data used for sections 6 and 7 are available at https://doi.org/10.5281/zenodo.3976221.

6.1 Methodology

For each API mentioned in section 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 [43] 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 2. The most

Table 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
Crypto. Algos.	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%	

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 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 inspected each warning issued by each verifier, and classified it 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.

6.2 Findings

Table 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.
- Most warnings were true 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 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 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.

6.3 Example compliance violations

Figure 4 shows two examples of compliance violations:

(a) An HSM (Hardware Security Module) simulator [38] uses the DES encryption algorithm. An HSM is a physical device used for managing encryption keys. Practical brute-force attacks against DES were public knowledge as early as 1998 [31].

```
if (sCommand.contains("#S#>")) {
  SecretKey sk_w_key_VNN = new SecretKeySpec(b_w_key_VNN, "DES");
  ...
}
```

(a) 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");
```

(b) An example use of a hard-coded key.

Figure 4: Example compliance violations our checkers found.

(b) A command-line email client [70] uses a hard-coded key. 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.

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 [65] have developed a microbenchmark set of misuses of cryptography, which they call CryptoAPIBench [3]. 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.

7.1 Tools compared

We compared our verifier to four state-of-the-art tools that detect misuses of cryptographic APIs.

- SpotBugs [25] is the successor of FindBugs [8], 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 (SpotBugs$_{\mathrm{D}}$), and version 3.1.12 configured with the ruleset from the SWAMP [74], which contains additional security bug patterns (SpotBugs$_{\mathrm{S}}$). For both versions, we only enabled warnings in the `SECURITY` category.
- Coverity [13] 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$_{SAST}$ [47] is a tool that checks user-written specifications (in the custom CrySL language) consisting of typestate

Table 3: Comparison of tools for finding misuses of cryptographic APIs, on relevant parts of CryptoAPIBench.

	SpotBugs$_D$	SpotBugs$_S$	Coverity	CogniCrypt	CryptoGuard	This paper
Original Labeling						
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 Labeling						
Precision	-	0.69	1.0	0.79	1.0	0.97
Recall	0.0	0.32	0.38	0.61	0.88	1.0

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 [65] 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 [22].

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) [40]. 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 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.

7.2 Results

Table 3 shows the results of the comparison. Precision and recall are defined identically to CryptoAPIBench [3]. Our numbers differ from [3] 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, as safe.
- 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 [14]. CryptoAPIBench labels 11 calls to `Cipher.getInstance("RSA")` that use the default padding as safe.

[1]The tools also have other capabilities, but our evaluation focuses on this aspect of their functionality.

```
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: Code from a service with a code path that could have been used to generate a 128-bit key.

The "compliance labeling" in table 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.

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 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.

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.
- 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.

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 [54].

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. 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 [83].

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.

[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 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

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 5.1 and 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 4 categorizes each package into one of four categories:

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 intraprocedural, 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. 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 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.

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.

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 [41], and social pressure is an important factor influencing whether security tools are adopted by practitioners [82]. 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 [49] 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 task 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

[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.

interested in impact in practice to investigate other stakeholders in the correctness of software besides the developers themselves.

11 RELATED WORK

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 [55]. 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 [11, 36, 72, 76]. 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 [64] 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 [15, 51, 52].

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 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) [52]. They do not mention sound verification. Ullah et al. describe a framework for building an automated cloud security compliance tool [76]. 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 [50].

Formal methods like process modeling have been applied to compliance problems, especially in safety-critical domains such as railway [12] and automotive systems [9]. The COMPAS project [27] 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 [46]. Tran et al. developed a framework for expressing compliance requirements in a service-oriented architecture [75]. 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. Pavlova et al. developed a technique for inferring JML annotations that encode security policies of JavaCard applets [60]. 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 [29] toolkit for statically verifying that Android apps did 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% or more) Java applications that use cryptography *misuse* it [20], and most (>80%) security vulnerabilities related to cryptography are due to improper usage of cryptographic APIs [48]. CogniCrypt$_{SAST}$ [47] is a technique based on synchronized push-down systems for finding unsafe uses of cryptography APIs. CryptoGuard [65] is a heuristic-based tool based on program slicing for finding unsafe uses of cryptography APIs. We compare to both in section 7.

12 CONCLUSION

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 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 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 that goal.

Acknowledgments: We thank the developers and compliance officers at AWS who participated and provided feedback, especially Zaanjana Sreekumar. Thanks to Sean McLaughlin for his help with this project. Thanks to Ranjit Jhala for comments on a draft of this paper.

REFERENCES

[1] 2002. FIPS PUB 140-2, Security Requirements for Cryptographic Modules. U.S.Department of Commerce/National Institute of Standards and Technology.
[2] Bhargav Acharya. 2016. Why Cloud Providers Need a SOC Report. https://www.schellman.com/blog/why-cloud-providers-need-a-soc-report. Accessed 28 March 2019.
[3] Sharmin Afrose, Sazzadur Rahaman, and Danfeng Yao. 2019. CryptoAPI-Bench: A Comprehensive Benchmark on Java Cryptographic API Misuses. In *2019 IEEE Cybersecurity Development (SecDev)*. IEEE, 49–61.
[4] AICPA. 2017. 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. Accessed 1 February 2019.

[5] Amazon Web Services, Inc. 2006. Amazon S3. https://aws.amazon.com/s3/. Accessed 17 April 2020.

[6] aws-kms-compliance-checker Developers. 2020. awslabs/aws-kms-compliance-checker. https://github.com/awslabs/aws-kms-compliance-checker. Accessed 11 August 2020.

[7] awslabs/aws-crypto-policy-compliance-checker Developers. 2020. awslabs/aws-crypto-policy-compliance-checker. https://github.com/awslabs/aws-crypto-policy-compliance-checker. Accessed 11 August 2020.

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

[9] Ghada Bahig and Amr El-Kadi. 2017. Formal verification of automotive design in compliance with ISO 26262 design verification guidelines. *IEEE Access* 5 (2017), 4505–4516.

[10] Elaine B. Barker, William C. Barker, William E. Burr, W. Timothy Polk, and Miles E. Smid. 2007. *SP 800-57. Recommendation for Key Management, Part 1: General (Revised).* Technical Report. Gaithersburg, MD, United States.

[11] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. 2010. State of the art: Automated black-box web application vulnerability testing. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 332–345.

[12] Cinzia Bernardeschi, Alessandro Fantechi, Stefania Gnesi, Salvatore Larosa, Giorgio Mongardi, and Dario Romano. 1998. A formal verification environment for railway signaling system design. *Formal Methods in System Design* 12, 2 (1998), 139–161.

[13] 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.

[14] Daniel Bleichenbacher. 1998. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1. In *Annual International Cryptology Conference*. Springer, 1–12.

[15] Sören Bleikertz, Carsten Vogel, Thomas Groß, and Sebastian Mödersheim. 2015. Proactive security analysis of changes in virtualized infrastructures. In *Proceedings of the 31st annual computer security applications conference*. ACM, 51–60.

[16] Grady Booch. 1991. *Object Oriented Design with Applications*. Benjamin/Cummings.

[17] KA Briski, Poonam Chitale, Valerie Hamilton, Allan Pratt, B Starr, J Veroulis, and B Villard. 2008. Minimizing code defects to improve software quality and lower development costs. *Development Solutions. IBM. Crawford, B., Soto, R., de la Barra, CL* (2008).

[18] bucket-complaince-checker Developers. 2020. kelloggm/bucket-compliance-checker. https://github.com/kelloggm/bucket-compliance-checker. Accessed 11 August 2020.

[19] Matthew Campagna. 2015. Aws key management service cryptographic details.

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

[21] PCI Security Standards Council. 2020. Qualified Security Assessors. https://www.pcisecuritystandards.org/assessors_and_solutions/qualified_security_assessors. Accessed 14 April 2020.

[22] CryptoGuard Developers. 2020. CryptoGuardOSS/cryptoguard. https://github.com/CryptoGuardOSS/cryptoguard/commit/2898b5b5ec25d94bbedda271638385c0fa6e0c9c. Accessed 26 April 2020.

[23] Checker Framework Developers. 2019. Constant Value Checker. https://checkerframework.org/manual/#constant-value-checker. Accessed 10 August 2019.

[24] Checker Framework Developers. 2020. Whole-program Inference. https://checkerframework.org/manual/#whole-program-inference. Accessed 17 April 2020.

[25] SpotBugs Developers. 2020. SpotBugs. https://spotbugs.github.io/. Accessed 24 April 2020.

[26] Mark Jason Dominus. 2001. Perl regular expression matching is NP-hard. https://perl.plover.com/NPC/.

[27] Schahram Dustdar. 2010. 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. Accessed 4 April 2019.

[28] Stacy English and Susannah Hammond. 2018. Cost of Compliance 2018. https://legal.thomsonreuters.com/content/dam/ewp-m/documents/legal/en/pdf/reports/cost-of-compliance-special-report-2018.pdf. Accessed 26 February 2019.

[29] 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. 2014. 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*. Scottsdale, AZ, USA, 1092–1104.

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

[31] Electronic Frontier Foundation. 1998. Cracking DES: Secrets of encryption research, wiretap politics and chip design.

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

[33] Alain Frisch and Luca Cardelli. 2004. Greedy regular expression matching. In *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004*. Turku, Finland, 618–629.

[34] Dan Fritsche and Bhavana Sasne. 2015. Whitepaper: The Costs of Failing a PCI-DSS Audit. https://www.hytrust.com/wp-content/uploads/2015/08/HyTrust_Cost_of_Failed_Audit.pdf. Accessed 18 March 2019.

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

[36] Hossein Homaei and Hamid Reza Shahriari. 2019. Athena: A framework to automatically generate security test oracle via extracting policies from source code and intended software behaviour. *Information and Software Technology* 107 (2019), 112–124.

[37] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. 2005. Regular Expression Types for XML. *ACM Transactions on Programming Languages and Systems* 27, 1 (January 2005), 46–90. https://doi.org/10.1145/1053468.1053470

[38] hsm-simulator Developers. 2019. gjyoung1974/hsm-simulator. https://github.com/gjyoung1974/hsm-simulator/blob/432b2b6e9fd63936347293743e54a8e572367fda/src/com/goyoung/crypto/hsmsim/commands/crypto/GenerateVISAWorkingKey.java. Accessed 5 May 2020.

[39] Wei Huang, Yao Dong, and Ana Milanova. 2014. Type-based taint analysis for Java web applications. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 140–154.

[40] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *ICSE 2013, Proceedings of the 35th International Conference on Software Engineering*. San Francisco, CA, USA, 672–681.

[41] Elena Karahanna, Detmar W Straub, and Norman L Chervany. 1999. Information technology adoption across time: a cross-sectional comparison of pre-adoption and post-adoption beliefs. *MIS quarterly* (1999), 183–213.

[42] Audrey Katcher. 2019. Understanding How Users Would Make Use of a SOC2 Report. https://www.rubinbrown.com/soc2_user_document_111710.pdf. Accessed 28 March 2019.

[43] Martin Kellogg. 2020. do-like-javac. https://github.com/kelloggm/do-like-javac. Accessed 24 April 2020.

[44] 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.

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

[46] Natallia Kokash and Farhad Arbab. 2008. Formal behavioral modeling and compliance analysis for service-oriented systems. In *International Symposium on Formal Methods for Components and Objects*. Springer, 21–41.

[47] 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.

[48] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. 2014. Why does cryptographic software fail?: a case study and open problems. In *Asia-Pacific Workshop on Systems*. ACM, 7.

[49] Gary T Leavens, Albert L Baker, and Clyde Ruby. 2006. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes* 31, 3 (2006), 1–38.

[50] Sebastian Lins, Stephan Schneider, and Ali Sunyaev. 2018. Trust is good, control is better: Creating secure clouds by continuous auditing. *IEEE Transactions on Cloud Computing* 6, 3 (2018), 890–903.

[51] Suryadipta Majumdar, Yosr Jarraya, Momen Oqaily, Amir Alimohammadifar, Makan Pourzandi, Lingyu Wang, and Mourad Debbabi. 2017. Leaps: Learning-based proactive security auditing for clouds. In *European Symposium on Research in Computer Security*. Springer, 265–285.

[52] Suryadipta Majumdar, Taous Madi, Yosr Jarraya, Makan Pourzandi, Lingyu Wang, and Mourad Debbabi. 2018. Cloud Security Auditing: Major Approaches and Existing Challenges. In *Symposium on Foundations & Practice of Security*.

[53] Mastercard. 2017. 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. Accessed 18 March 2019.

[54] Suzanne Millstein. 2016. Implement Java 8 type argument inference. https://github.com/typetools/checker-framework/issues/979. Accessed 17 April 2020.

[55] Deepti Mishra and Alok Mishra. 2009. Simplified software inspection process in compliance with international standards. *Computer Standards & Interfaces* 31, 4 (2009), 763–771.

[56] MS 325E.64. 2007. Access Devices; Breach of Security. Minnesota Statutes (2018): Chapter 325E, Section 64.

[57] Jacob Nemetz and Brett Lieblich. 2019. Dash Compliance Automation — S3 Security Controls. https://www.dashsdk.com/docs/aws/hipaa/amazon-s3/. Accessed 8 April 2020.

[58] no-literal-checker Developers. 2020. kelloggm/no-literal-checker. https://github.com/kelloggm/no-literal-checker. Accessed 11 August 2020.

[59] 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.

[60] Mariela Pavlova, Gilles Barthe, Lilian Burdy, Marieke Huisman, and Jean-Louis Lanet. 2004. Enforcing high-level security properties for applets. In *Conference on Smart Card Research and Advanced Applications*. Springer, 1–16.

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

[62] PCI Security Standards Council. 2018. PCI DSS v3.2.1 Template for Report on Compliance. https://www.pcisecuritystandards.org/documents/PCI-DSS-v3_2_1-ROC-Reporting-Template.pdf. Accessed 4 April 2019.

[63] Ponemon Institute LLC. 2011. The True Cost of Compliance. https://www.ponemon.org/local/upload/file/True_Cost_of_Compliance_Report_copy.pdf. Accessed 3 April 2019.

[64] Sazzadur Rahaman, Gang Wang, and Danfeng Yao. 2019. Security Certification in Payment Card Industry: Testbeds, Measurements, and Recommendations. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 481–498.

[65] 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.

[66] RCW 19.255.020. 2010. Liability of processors, businesses, and vendors. Revised Code of Washington, Title 19, Chapter 19.255, Section 19.255.020.

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

[68] Martin Schaef. 2019. Example of how to whitelist crypto algorithms. https://github.com/awslabs/aws-crypto-policy-compliance-checker/blob/master/stubs/javax.crypto.astub. Accessed 11 August 2020.

[69] Helmut Seidl. 1990. Deciding Equivalence of Finite Tree Automata. *SIAM J. Comput.* 19, 3 (June 1990), 424–437. https://doi.org/10.1137/0219027

[70] sendmail Developers. 2015. NewSaigonSoft/sendmail. https://github.com/NewSaigonSoft/sendmail/blob/e31d9a86c7f863c59fc51d5fd2c1b60cc4586faf/src/main/java/com/newsaigonsoft/sendmail/SecurePassword.java. Accessed 5 May 2020.

[71] Square, Inc. 2017. PCI Compliance: What You Need to Know. https://squareup.com/guides/pci-compliance. Accessed 18 March 2019.

[72] Philipp Stephanow and Christian Banse. 2017. Evaluating the performance of continuous test-based cloud service certification. In *International Symposium on Cluster, Cloud and Grid Computing*. IEEE Press, 1117–1126.

[73] 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.

[74] The SWAMP Team. 2020. Welcome To The SWAMP. https://continuousassurance.org/. Accessed 24 April 2020.

[75] Huy Tran, Ta'id Holmes, Ernst Oberortner, Emmanuel Mulo, Agnieszka Betkowska Cavalcante, Jacek Serafinski, Marek Tluczek, Aliaksandr Birukou, Florian Daniel, Patricia Silveira, et al. 2010. 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*. IEEE, 407–414.

[76] Kazi Wali Ullah, Abu Shohel Ahmed, and Jukka Ylitalo. 2013. Towards building an automated security compliance tool for the cloud. In *Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 1587–1593.

[77] Ciske van Oosten, Anne Turner, Cynthia B. Hanson, Dyana Pearson, Ronald Tosto, and Andi Baritchi. 2018. Verizon 2018 Payment Security Report. Accessed 26 February 2019.

[78] Steven VanRoekel. 2011. Security Authorization of Information Systems in Cloud Computing Environments. https://www.fedramp.gov/assets/resources/documents/FedRAMP_Policy_Memo.pdf. Accessed 31 January 2019.

[79] vault Developers. 2020. NitorCreations/vault. https://github.com/NitorCreations/vault/blob/3c3ec65879c82bb353b4cf4d22898abb0b7b578f/java/src/main/java/com/nitorcreations/vault/VaultClient.java. Accessed 8 May 2020.

[80] VISA, Inc. 2017. Data Security Compliance Requirements for Service Providers. https://usa.visa.com/dam/VCOM/download/merchants/data-security-compliance-service-providers.pdf. Accessed 31 January 2019.

[81] Mark N Wegman and F Kenneth Zadeck. 1991. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 2 (1991), 181–210.

[82] Jim Witschey, Olga Zielinska, Allaire Welk, Emerson Murphy-Hill, Chris Mayhorn, and Thomas Zimmermann. 2015. Quantifying developers' adoption of security tools. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 260–271.

[83] Chad Woolf, Byron Cook, and Tom McAndrew. 2019. Automate Compliance Verification on AWS Using Provable Security. https://www.youtube.com/watch?v=BbXK_-b3DTk. Accessed 25 August 2020.