Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

RVSec: Runtime Verification Methods for High Precision Detection of Cryptography API Misuse

Adriano Torres

Dissertation presented as a partial requirement for graduation in the Postgraduate Program In Informatics - PPGI

Supervisor
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2022

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

RVSec: Runtime Verification Methods for High Precision Detection of Cryptography API Misuse

Adriano Torres

Dissertation presented as a partial requirement for graduation in the Postgraduate Program In Informatics em - PPGI

Prof. Dr. Rodrigo Bonifácio de Almeida (Supervisor)
CIC/UnB

Prof. Dr. Leopoldo Teixeira
UFPE

Prof. Dr. Vander Alves
UnB

Prof. Dr. Ricardo Pezzuol Jacobi
Coordenador do Programa de Pós-graduação em Informática - PPGI

Brasília, 21 de february de 2022

Resumo

O uso incorreto de APIs de criptografia pode causar vulnerabilidades em software. Portanto, recentemente, foram propostas ferramentas baseadas em análise estática para detecção de mau uso. Estes detectores encontram diversos maus usos, mas diferem em suas capacidades e limitações, alem de não detectarem alguns bugs. Neste trabalho, investigamos verificação em tempo de execução (RV, de Runtime Verification em Inglês) - como uma alternativa baseada em análise dinâmica para detectar mau uso de crypto APIs. RV monitora execução de programas em relação a especificações formais, e há evidência da eficiência e eficácia do seu uso na deteccção de bugs em software.

Neste estudo empírico sobre a eficácia e eficiência da aplicação de análise dinâmica para detectar tais maus usos, nós propomos um um protótipo baseado em JavaMOP - uma implementação de Monitoring-Oriented Programming para Java - para realizar Verificação em Runtime (RV) de 22 classes da Java Cryptography Architecture (JCA).

Desenvolvemos nossas especificações através da tradução manual de 22 especificações presentes em CrySL - o estado da arte em detecção estática - para nosso contexto MOP. Após conduzir um estudo comparativo de RVsec com o estado da arte em análise estática, nos calculamos métricas de acurácia - precision, recall e F-measure - bem como custos de execução e correlação com cobertura de testes. Nossos resultados suportam RV como um complemento efetivo para analisadores estáticos, uma vez que os métodos aqui propostos apresentam precisão comparável, quando não superior, sem incorrer em custos de execução proibitivos.

Palavras-chave: Segurança, Criptografia, Verificação em Tempo de Execução, Análise de Programas, Engenharia de Software

Abstract

Incorrect usage of cryptographic (crypto) APIs can cause software security vulnerabilities, but developers often find it difficult to reason about those APIs. To automate the detection of misuse, static-analysis based crypto API tools have been proposed. These detectors find many misuses, but they differ in strengths and weaknesses, and miss bugs. We investigate runtime verification (RV) as a dynamic-analysis based alternative for crypto API misuse detection. RV monitors program runs against formal specifications and was shown to be effective and efficient for amplifying the bug-finding ability of software tests.

In this empirical study on the efficacy and efficiency of applying dynamic analysis to detect such misuses, we propose a prototype based on JavaMOP - a Java implementation of Monitoring-Oriented Programming - to perform Runtime Verification of 22 classes of the Java Cryptography Architecture (JCA).

We developed our specifications by manually translating 22 specifications from CrySL - a state-of-the-art static detector - into our MOP context. Upon conducting a comparative assessment of the methods using three benchmarks provided in the literature, we evaluated accuracy metrics - precision, recall and F-measure - as well as runtime overhead cost and correlation to coverage. Our results support RV as an effective complement to static analysers, as the methods herein proposed presented competitive, when not superior, accuracy metrics, without incurring in prohibitive runtime overhead.

Keywords: Security, Cryptography, Runtime Verification, Program Analysis, Software Engineering

Contents

List of Figures

List of Tables

Acronyms

AOP Aspect Oriented Programming.

API Application Programming Interface.

CFG Context-Free Grammar.

ERE Extended Regular Expression.

FSM Finite State Machine.

IDE Integrated Development Environment.

JCA Java Cryptography Architecture.

LTL Linear Temporal Logic.

MOP Monitoring Oriented Programming.

PTCaRet Past Time Linear Temporal Logic With Calls and Returns.

ptLTL Past Time Linear Temporal Logic.

RV Runtime Verification.

Chapter 1

Introduction

When storing and manipulating data that needs protection, developers often resort to some form of encryption. This is usually performed by integrating cryptography libraries into source code, which brings forth a new level of concern about security: making sure that developers make secure use of the APIs provided by such libraries. Previous studies suggest that the opposite is frequently the case, as misuses of such APIs by developers often introduce vulnerabilities into source code [1], [2], [3].

Cryptography systems are reportedly regarded as hard to engineer. When designing them, not only does one have to possess knowledge about several disciplines of Mathematics and Computer Science, but one also faces the engineering trade-offs involved in the creation of abstractions that are easy enough for developers to use, but that also implement techniques that are hard to break [4]. Even though there has been criticism to the fact that cryptography is often treated as "just another component" [5] to be plugged in during the software development lifecycle, many of the main programming platforms have developed cryptography APIs, and these often become widely used by developers [6].

Since previous research reports that the presence of low-level abstractions and lack of tooling to foster the correct use of crypto APIs are among the main reasons why developers struggle to make proper use of them [7] [8] [6], several static analysis methods have been developed to detect whether certain assumptions and preconditions are met by the code being analysed [9] [10].

In this empirical study, we propose an alternative approach to static methods for performing the detection of cryptography API misuses. Since the implications of applying a dynamic analysis approach for this domain have not been reported in the literature, we are motivated by the fact that Runtime Verification attempts to introduce a layer of formalism to the specifications against which the code under inspection is checked, without introducing all the complexity and overhead of purely formal approaches, such

as model checking and theorem proving. As an example, let us mention that in RV it is possible to define a finite state machine to specify the entire computation automaton, with all its states and transitions[1].

By leveraging open source implementations of Monitoring-Oriented Programming (MOP) [11] that allow for scalable Runtime Verification (RV) [12], [13], the methods and artefact herein proposed aim to detect crypto API misuses by actually executing the code and checking if specified assumptions about how the respective APIs is expected to be used are actually met by consumer code.

The objective of this research is thus to investigate the efficacy and effiency of the application of dynamic analysis methods, i.e., methods wherein the program under inspection is actually executed and monitored, and apply available RV technology as an alternative to the state-of-the-art literature and tooling, which base themselves on allow-list methods [9] for crypto API misuse detection, and whose core modules are based on static analysis. [14]. As we shall see throughout the text, even if it is possible to apply advanced methods and techniques [10] to arbitrarily enhance precision of static analysers, the engineering and runtime costs involved can make it difficult, if not impractical, to create scalable and extensible tools that enforce the underlying security assumptions behind the implementation of widely available cryptography APIs.

The main contributions we aim with this work are:

- A **prototype tool** that allows for an expressive way to define complex relationships between different components of a given cryptography architecture, which can be easily integrated into a project's code base to perform verification;
- An **empirical assessment** of the efficiency and efficacy of our RV methods for detecting misuses of crypto APIs. In our assessment, we are going to conduct a comparative study between of our artefact's (which we will call RVSec) **accuracy metrics**, namely precision and recall, between our results and those produced by state-of-the-art-static analysis methods, when all the tools are run against test benchmarks available in the literature. We also expect to be able to gather insight about complex scenarios that arise when such libraries are consumed, and establish guidelines for what kinds of relationships are better expressed/detected in static analysis, and vice-versa;
- **Strategies** for the development of specification files that leverage MOP concepts to verify code behaviour. We also provide a Test-Driven Development process for

[1]Another potential upside of applying RV, which shall not be explored in this work, is the possibility of altering program behaviour at runtime. If, for example, an RV analysis tool detects that a key has not been properly randomised prior to being used for encryption, such tool could, in principle, generate a proper key and perform secure ciphering instead.

translating specifications from CrySL, a definition language developed as part of the state of the art in static detection [9], into our MOP framework;

If feasible at scale, our approach can serve as the groundwork for the creation of software and tooling for specifying and validating the usage of crypto APIs, public or private. Although in this work we restrict outselves to the context of the Java Cryptography Architecture (JCA), the approach can be extended to other cryptography frameworks. Moreover, the methods herein applied make no restriction to the kind of code to be analysed, which means we can apply our methods to specify behaviour of non-cryptographic APIs as well.

Another potential implication of a dynamic analysis-based approach lies in the creation of safety net tools that prevent unsafe code from executing at all. As we shall see, one of the upsides of MOP-based Runtime Verification is that we can alter program behaviour at runtime, therefore avoiding the execution of identified API misues. Even though this is not going to be explored in our work, this is a potential upside of RVSec, as the actual program trace is available for manipulation during execution.

1.1 Research hypotheses

In this section, we present the major hypotheses to be investigated in this work. In their most general sense, they can be stated as:

- **R.H. 1**: Given the current advances in Runtime Verification literature and technology, it is possible to leverage Runtime Verification tooling to perform efficient Runtime Verification of cryptography API misuses at scale.
- **R.H. 2**: Provided that there is a positive answer to the aforementioned hypothesis, our second hypothesis is that the dynamic analysis approach can yield accuracy metrics that are comparable, if not superior, to those of the state of the art in crypto API misuse detection, which base themselves in static analysis.

1.2 Research objectives

The overarching problem we aim to solve in this research can be stated as that of investigating the possibility of creating a dynamic analysis framework whose metrics for accuracy and performance are comparable to the current state-of-the-art static analysis approaches for specifying correct usage of cryptography APIs. The following are some of the questions that will guide the researcher:

- Is it possible to express syntactic and semantic concepts used by static analysers for specifying the correct usage of crypto APIs in an MOP context?
- Can such approach efficiently detect misuse? In other words, how does the precision and recall of an RV-based detection approach compare to the state of the art?
- Do there exist correlations between test coverage and number of warnings reported? Does the number of monitors generated correlate to misuses detected?

1.3 Research Method

In order to to address these questions and validate or refute our hypotheses, we first present a brief overview of static analysis, the method at the core of most of the current state-of-the-art tooling to detect misuses of the Java Cryptography Architecture (JCA) as well as violations of other kinds of security rules, which we found during our literature review. Our review also includes an introduction to methods and tooling that form the basis of an integrated framework based on available open source projects that performs scalable Runtime Verification to decide on whether the code under inspection behaves as expected. This should lay the groundwork for the construction of our artefact.

Initially, we are going to investigate how one can apply dynamic analysis to detect misuses of a subset of the JCA APIs. This shall be done by creating specification files that leverage MOP concepts. The main framework we are going to use is JavaMOP [15], which synthethises monitors that allow us to react to method calls, thus allowing us to observe properties of the objects being manipulated. If, for instance, there is a call to the one-way hashing functionality provided by the `MessageDigest` class, i.e., if `MessageDigest.update()` is invoked, we can capture such an event and make assertions about its input types/values, whilst imposing constraints and relationships between the different objects that are used during a complete `MessageDigest` call sequence. In practice, we are going to consider CrySL [9], a domain-specific language created for the exact purpose of specifying the behavior of several classes of the JCA, as the starting point for the creation of the JavaMOP specs.

Upon completion of the aforementioned investigation, we are going to perform the empirical validation of our artefact by comparing its misuse detection capabilities, as well as performance overhead and costs, with its static analysis counterparts, so we can then see how our approach compares to static methods when the tools are run against benchmark test sets that were selected during our literature review.

Chapter 2

Background and Related Work

In this chapter, we discuss some of the literature on the current methods that are the most efficient at crypto API misuse detection. The artefacts produced by the works referenced in this chapter are the ones we are going to compare our framework to during our empirical assessment. We start with a brief introduction to static analysis, and then proceed to introducing CrySL [9] and CryptoGuard [10], two of the foremost tools that apply it to detect crypto API misuse.

Our literature review of the tooling and technology available for detecting crypto API misuse showed that the underlying paradigm upon which most of them operate is static analysis [14]. Our review on static detection started with [9] and [10], followed by the other works of the authors. For RV techniques, we started with [16]. Then we read, in chronological order, a series of RV papers, all published by the same group as the original one. This series of papers resulted in the creation of JavaMOP [15], the infrastructure upon which our tool is built.

2.1 Current State of the Art for Detecting API Misuse

In this section, we present a brief overview of what static analyis entails. We then explore in further detail how current state-of-the-art tooling and methods [9], [10] work. We conclude by describing some of the theoretical limitations of static analysis, as well as engineering challenges associated with creating accurate approximations of program behaviour statically.

2.1.1 Static Analysis Overview

Static analysis is a field of Computer Science wherein a program's behaviour is inferred and inspected without the program under consideration being executed. Software that performs static analysis usually require two types of input:

- Input code: a representation of the program we want to analyse [14];
- Specifications: sets of rules about the expected behaviour of certain parts of the program, such as function invocations, types, values and other properties of the parameters being passed, as well as sequences of calls. In general, any constraint that can be applied to the form of the input code can become a specification.

The output of such tools can also be of two general kinds:

- Code: Upon processing the input code against the specifications, static analysis tools can produce code that has been adapted to meet the specifications. One of the main examples of this is the refactoring functionality that is found in code editors. These take textual language (code) as input, matches it against that language's grammar rules (specifications), and produces suggestions that conform with the grammar;
- Textual or tabular data, such as warnings or violation reports, about the places where the specifications are not met. These can also be source code locations, line number, class names and so on.

Due to not executing the code under consideration, static analysis tools have to inspect input code against a predefined set of expected conditions. When a condition is not met, we say that a *violation* has occurred [17]. Therefore, it is fair to state that the overall objective of static analaysis tools is to process code against a specification set and search for violations, producing either reports for subsequent human action, or suggestions of versions of the code that are compliant.

In order to generate approximations of how the program would behave at runtime, static methods employ several different strategies. Customary approaches are[1]:

- *Data Flow Analysis*: Information about the program is derived and transmitted by representing the program as a graph whose nodes are elementary blocks, and whose edges represent how control and data might pass from one black to another. This is usually performed by extracting equations out of a program. Such method can also be expressed as a Constraint Based Analysis, in which certain conditions or inequations are derived out of the program, and then imposed onto its behaviour;

[1]This is by no means a formal introduction to the elements of static analysis. For an in-depth discussion, plese refer to [18]

- In *Abstract Interpretation*, traces that record information about events that take place during the computation are maintained. As such traces "contain sufficient information that we can extract a set of *semantically reaching definitions*" [18], Abstract Interpretation provides a more general theorical method for calculating analyses, and is not dependant on any given specification style;
- *Type and Effect Systems* are formalisms that describe properties - or states - of a program, as well as the effect of executing statements - or its transitions. *Annotated Type Systems* concerns itself with the states, whereas *Effect Systems* handles transitions;

Some examples of programs that perform static analysis and are frequently used by software engineers are:

- Integrated Development Environments, or IDEs, can analyse code as it is written, producing real time warnings about potential issues with the code. They can also suggest automated refactorings, such as method extraction, variable inlining, or any other transformation that can be applied to the program's internal representation;
- Compilers usually employ static analysis in more than one phase of the compilation process. During parsing, static analysis is employed to make sure the program's syntax is unanbiguous. This is usually performed by the creation and analysis of data structures that result from analysing source code, such as Abstract Syntax Trees - ASTs - and Control Flow Graphs, or CFGs. Static methods also take place during semantic analysis, such as when type checking is performed [19].

Further elaboration on static analysis is outside the scope of this text. The reader should refer to [19], [20] and [18] for more detailed discussions on static analysis and its applications in cryptography.

2.1.2 Static Detection of Crypto API Misuses

This section uses an example to describe the syntax and semantics of CrySL, a domain-specific **Cr**yptography **S**pecification **L**anguage. Although it is not exhaustive, our example describes most of the concepts needed for understanding its specification files. For a complete formalization of the language, the reader should refer to [9].

CrySL - A DSL For Crypto API Specification

Krüger et al. proposed a definition language called CrySL [9] , which can serve as a building block for documentation, code generation, and program analysis. It provides

a lightweight syntax for specifying the correct usage of crypto APIs. The authors also introduced $CogniCrypt_{SAST}$, a CrySL compiler that generates the static analysis and allows us to check applications for compliance with CrySL's encoded rules. At the time of this writing, CrySL and its compiler are the most comprehensive and precise tools for detecting misuses of the JCA.

In the following listing, we provide an example of a simplified, yet functional, CrySL specification for Cipher, one of the core classes of the JCA. It is important to notice that writing a CrySL specification is not a trivial task, because it requires one to possess advanced knowledge of cryptography. Ultimately, writing a such a specification requires one to have command over not only about the JCA itself; one is also required to understand the underlying mathematical and computational concepts behind encryption, ciphering, randomisation, and the other techniques involved in cryptography [5]. For our purposes, CrySL's specifications serve as a paramount starting point, as they allow us to concentrate effort on expressing the relationships, constraints, requirements and guarantees in our dynamic analysis context, rather than devising and validating those in the first place.

```
SPEC java.security.Cipher

OBJECTS
        java.lang.String transformation;
        int encryptionMode;
        java.security.Key key;
        byte[] wrappedKeyBytes;
        java.security.Key wrappedKey;

EVENTS
        instantiateWithoutProvider: getInstance(transformation);
        instantiateWithProvideR: getInstance(transformation, _);
        Instantiate := instantiateWithoutProvider | instantiateWithProvideR;
        initialize: init(encryptionMode, key);
        wrap: wrappedKeyBytes = wrap(wrappedKey);

ORDER
        Instantiate+, initialize+, wrap

CONSTRAINTS
        encryptionMode in {1,2,3,4};
        alg(transformation) in {"AES"} => mode(transformation) in {"CBC", "GCM", "PCBC", "CTR"
            ↪ , "CTS", "CFB", "OFB"};

REQUIRES
        generatedKey[key, alg(transformation)];
```

```
    mode(transformation) in {"OAEPWithMD5AndMGF1Padding", "OAEPWithSHA-224AndMGF1Padding",
        ↪  ...} => preparedOAEP[paramSpec];

ENSURES
    generatedCipher[this] after initialize;
    wrappedKey[wrappedKeyBytes, wrappedKey];
```

Each section is defined by an uppercase word. The code listing above contains all of CrySL's mandatory sections, namely:

- The `SPEC` section, in which we indicate the class whose behaviour we want to specify;
- The `OBJECTS` is the place where we can declare the objects that will get manipulated at runtime, if the program is actually executed. As we will see below, we can also declare objects upon which we want to define preconditions and postconditions;
- `EVENTS` is where we bind names to method calls. In our example, `initialize` is bound to the `Cipher.init()` method. Notice that, since methods can be polymorphic, CrySL allows us to define conjunctions of events, like in the `Instantiate` event in our example file;
- The `ORDER` section is where we define an Extended Regular Expression (ERE), which defines sequences of events. The standard operators for regular expressions, such as the Kleene operators $+$ and $*$, also apply to this context;
- The `CONSTRAINTS` section is used to set preconditions to be met by some of the `OBJECTS` involved in the computations. In our case, for instance, we are restricting the values `encryptionMode` can assume, and we are also defining a mapping between the values of `alg(transformation)` and `mode(transformation)`[2]. In other words, we are stating that, if the algorithm to be used for ciphering is AES, then the respective mode must be one of the seven values defined in the constraint.
- The `REQUIRES` clause is used when one wants to specify inter-class dependencies. In our example above, from the Cipher specification file, we want to be able to make assertions about the *key* that we are going to use during the initialisation step. This means that this section is where we assert the behaviour of the other classes that are referenced by our Cipher spec, provided that those classes are being used correctly [9].

[2] `alg(transformation)` and `alg(transformation)` are auxiliary functions that extract the algorithm and the mode of the given transformation. For example, in `Cipher.getInstance("AES/CBC/PKCS5Padding")`, the transformation is "AES/CBC/PKCS5Padding", so the functions `alg` and `alg` return, respectively, "AES" and "CBC", which would configure a valid transformation in our case.

- Similarly to the above, in the `ENSURES` section, we specify the outcomes of the class under specification being used properly. In our example, we want to be able to *ensure* that a cipher object is available upon initialisation, as well to assert that the *key* that was provived has changed into a specific state.

Note that CrySL is a step toward a more declarative way of specifying the behaviour of cryptography libraries. Sets of specification files like the one above serve as input to $CogniCrypt_{SAST}$, which carries out the actual analysis, as well as the other tasks it can perform. However, the user is still required to understand Extended Regular Expressions (EREs). Also, as has been mentioned above, if users want to extend an existing specification, or create a new one, they must understand lower-level cryptography concepts. This emphasises our argument that the expertise provided by CrySL's authors is paramout to our research.

Upon running $CogniCrypt_{SAST}$ on 10001 Java projects, the authors reported that about 95% of the projects showed at least one misuse of the JCA APIs. Their results support $CogniCrypt_{SAST}$ as an effective tool for identifying JCA misuses. For a more detailed discusson on CrySL, its applications and results, please refer to [9].

CryptoGuard: Advanced Static Methods for False Positive Reduction

Trying to devise methods to address static analysis' inherent tendency to generate false positives, Rahaman et al. proposed CryptoGuard, a set of specialized program slicing algorithms whose common goal is to increase static analysis precision [10]. In essence, program slicing attempts to extract the minimal information that is necessary to specify a given behaviour, effectively reducing the cost of analysis [21].

By creating mappings between 16 security abstractions and concrete Java constructs, the authors were able to devise a set of slicing algorithms that significantly reduce the prevalence of false positives in the analysis. Based on empirical observations, the authors also performed several forms of systematic removal of irrelevant information, further increasing the perfomance and accuracy of *CryptoGuard* [10]. Another relevant contribution of this work was a benchmark test set, which shall be referenced during the empirical assessment of our artefact.

Because the process of creating the mapping and the respective algorithm for a given security rule involves expertise in advanced static analysis methods, the initial 16 rules are rather specific, attaining themselves mostly to the detection of predictable and/or constant values - such as keys, passwords, salts and seeds - as well as improperly randomised values. Nonetheless, the artifact performed very well, reducing false positive alerts by up to 80%[3] [10].

[3]The authors make no mention about CryptoGuard's recall in the original paper [10]

2.2 Limitations of Static Analysis

As mentioned above, since static analysis does not execute the code under inspection, it produces conservative approximations, hence producing an answer set that contains values which may not be correct. This is not an arbitrary design decision. It is, in fact, a constraint imposed to static analysis itself. Let us see why this is the case with an example. Consider the following piece of code:

```
isValid = validateInput(data);
if (isValid == true) {
    output = "success";
} else {
    output = "failure";
    handleFailure(data);
}
response = output;
```

By just reading the code without executing it, one can expect that, by the time we reach the last assignment statement, `output` can equal either `"success"` or `"failure"`. However, if the call to `handleFailure(data)` invokes code that never terminates, then it is also correct to state that the only value `output` can assume by the time we reach the last statement is `"success"`. However, one cannot, statically decide on whether `handleFailure(data)` terminates. Moreover, it has been shown that two poblems that are fundamental to static analysis are undecidable/uncomputable[4], "even when all paths are executable in the program being analyzed for languages with if statements, loops, dynamic storage, and recursive data structures." [22].

As such, both by constraint and design, static analysis is bound to produce a larger set of possibilities than the values that will ultimately be produced when the program executes. It is still important to stress, though, that static analysis is still a very effective way to investigate and assert properties about programs without running them, therefore avoiding most (if not all) of the costs associated with inspecting the program at runtime. From a software engineering perspective, static analysis is extremely useful in several steps of the development lifecycle, from the moment the engineer is writing the code and getting instant visual assistance, up until compilation and pre-deployment checks performed by continuous intgration pipelines.

Although the refinements proposed by CryptoGuard [10] are very efficient, they are also very specific and non-trivial to produce: not only does one have to carefully map a security specification to language-specific constructs, but one also needs to manipulate

[4]The two problems are, respectively, may-alias and must-alias, which are, in essence, problems that arise when one attempts to make guarantees about whether there exists aliasing between two memory references. For a thorough discussion on the topic, as well as proofs, please refer to [22]

slicing algorithms to enforce the specification. To make such an artefact reusable and extensible, it would be necessary to automate the production of slicing algorithms from input cryptographic specifications, which should take considerable effort. The authors even conclude their text by calling for such a tool, which would be “similar to what CrySL partially provides, but with much higher expressiveness, precision, and recall" [10].

In the next section, we introduce static analysis' counterpart, in which the program is actually executed, and properties or events are monitored at runtime. We also introduce two different programming paradigms, which form the core of the prototype proposed in this research.

Chapter 3

Scalable Runtime Verification

In this chapter we present two programming paradigms which are foundational to our work, namely: Aspect Oriented Programming (AOP) and Monitoring-Oriented Programming (MOP). Our systematic literature review on parametric Runtime Verification suggests that, from a performance standpoint, scalable Runtime Verification is approaching feasibility, which motivates our proposal of a framework for detecting crypto API misuse at runtime. In particular, we will see that the JavaMOP [13] implementation allows us to check code against specifications without incurring in prohibitive runtime costs, whilst keeping specification and error handling decoupled from the target code.

3.1 A Dynamic Analysis Approach

In contrast to static analysis, when inspecting programs dynamically, the program's code gets actually compiled and executed. When approached from this standpoint, analysis can be performed just as the execution stack is generated, and the execution is monitored via some kind of instrumentation [23]. In this section, we introduce some of the building blocks that allow the creation of a framework whose goal is to verify whether cryptography APIs are being used correctly.

Potential Downfalls of RV

Let us take a moment to discuss some potential drawbacks of a Runtime Verification approach. The foremost concern that arises when instrumenting code for observability is overhead, which can manifest itself in a few different ways, such as memory and CPU consumption, and execution time. A program that has its properties monitored at runtime is likely to consume extra resources. Therefore, careful consideration and experimentation must be given when assessing whether an RV approach is viable.

Another important aspect to be observed is tool and ecosystem maturity. For practical purposes, Runtime Verification is a relatively recent technique, dating to not much more than a decade from the time of this writing[1]. As such, the user of available tooling might face difficulties known to software in early stages, such as poor/nonexistent documentation, and bugs - both known and unknown - that are hard to understand and fix. To provide some examples, during our empirical assessment we came across several crashes caused by overflow issues when we started extending our specification files to contemplate entire event chains. This required us to inspect the internals of the monitor synthesiser - `rv-monitor` - to realise we needed to increase the virtual machine's stack size. We also came across other issues of the sort, and had to consult directly with the authors of JavaMOP to understand certain behaviours observed during the engineering of our prototype.

Let us now discuss two paradigms that, when properly combined, can produce an elegant way of generating monitors from event declarations.

3.1.1 Aspect Oriented Programming

Aspect Oriented Programming - AOP - is a programming paradigm whose primary concern is to modularize **cross-cutting** concerns, i.e., functionality exported from a module that often gets invoked accross several other modules of the application [27]. For an intuiton on what cross-cutting concerns are, consider the following scenarios:

- *Event persistence*: Logging is an example of functionality that can get invoked from virtually all other modules of an application. The use of logging applications usually involve coupling it to application logic, by inserting calls to the logger whenever we want to persist an event;
- Authentication: certain functionality of the application should only be carried out by authenticated/authorised users. Similarly to the case of the logger above, authentication on its own does not provide much use, and its applications usually take place by invoking it from many different parts of the software.

Both examples above should provide enough intuition into the problem that AOP is trying to solve: the fact that invoking code for cross-cutting functionality by traditional approaches inexorably introduces coupling between the module wherein calls to cross-cutting concern utilities take place, such as loggers and authenticators, and the consuming modules themselves.

[1] A quick Google scholar search on the string "Runtime Verification" yields its earlier relevant results at about 2001. When conducting our systematic literature review, we confirmed that its first practical developments happened at around this year [24, 25, 26]

AOP introduces a technique for expressing such cross-cutting concerns in a modular manner, consequently providing a way to separate components from aspects [27]. The elements of an AOP implementation are [28]:

- Identifiable points in the execution of the program - ways to inspect method calls, object manipulation, and exepction handling. These are usually called **join points**;
- Syntax for selecting join points. Having the aforementioned authentication example in mind, we want to be able to identify all the join points before which we want to ensure proper authentication has occured;
- Syntax for executing code whenever a join point code is run. Once we have identified all the join points that require authentication, we must be able to express that we want to perform authentication, or execute any block of code in general, before or after all such join points. These are commonly called the **advices**;
- Constructs to alter the static structure of the program. There are complex scenarios in which aspect oriented applications need to be able to alter the static structure of the system. For instance, introducing cross-cutting logging into several different classes is a kind of *inter-type* declaration, which needs access to the program's static structure. There are also cases where one wants to be able to detect or ascertain conditions or requirements to be met before or after the execution of certain methods. Such *weave-time* declarations constructs perform this task [28];
- Syntax for expressing cross-cutting concerns: It must be possible to express a cross-cutting concern in a self-contained, decoupled, and reusable way. Following our analogy, it must be possible to define a module where logging functionaliy happens, and to express that logging happens before/after a certain set of join points, instead of inserting a call to the logger module whenever we need it.

For the practical purposes of our project, we are going to use AspectJ [28], an AOP implementation available for Java. Please refer to the book for an extensive discussion.

3.1.2 Monitoring-Oriented Programming

In 2003, Chen and Ruso proposed a programming paradigm for checking conformance against specifications at runtime [11]. Their approach, which is language and formalism independent, paves the way for interesting applications, such as:

- Ways to increase program dependability and reliability by comparing program behaviour versus specified behaviour, and avoiding the execution of code that violates the specification;

- Extensions of programming languages that allow us to execute code when certain monitored events occur;
- A formal method which, relative to theorem proving and model checking, is lightweight, and whose primary concern, instead of performing static checks, is to not let the program misbehave at runtime;

In MOP, properties are specified using using specification formalisms, such as Regular Expressions and Finite State Machines, alongside code to be executed when such specifications are met or violated. [29] The way this is performed in an MOP framework is via the automatic generation of monitors from user-defined specifications, which can also be expressed in different formalisms.

Notice that, if we are able to express formal specifications as cross-cutting concerns to be enforced throughout an application, one can view MOP as an instance of AOP. Let us briefly examine an example to see why this is the case.

```
event invalidTransformation after(String transformation) returning(Cipher c):
     call(public static Cipher Cipher.getInstance(String)) &&
     args(transformation) &&
     condition(!isValid(transformation)) {
        System.out.println("Invalid transformation");
   }

event validTransformation after(...)

ere: validTransformation
```

The event definitions above use AspectJ syntax to express that, if the transformation string that is passed in to the method is not valid, then the specification has been violated and a warning must be raised. As shall be seen in the next sections, the fact that we are able to synthesise monitors from such aspects, therefore automatically generating agents that allow us to observe the program at runtime, is what motivates the interpretation of MOP as AOP.

The two paradigms also have in common the fact that code that happens to be generated by these methods needs to be integrated into the code under inspection. In fact, both predict such need [27], [11]. Thefore, if feasible, such a system can yield a productive combination of testing and formal methods [30].

From a software engineering perspective, MOP implementations are expected to provide abstractions for several different logic engines, so as to spare the developer from having to know how to translate a chosen formalism into a finite state machine, therefore abstracting away a relatively hard problem. This means that, so long as they know

how to express program state and behaviour using the target logic [11] [31], MOP implementations will handle monitor synthetization. They do so by having several different code generators, one for each of the the different logics that are supported by the actual implementation. The generators themselves take as input the specified behavior. In a subsequent step, the monitors are automatically synthesised and integrated.

3.2 Runtime Verification

The main contributions of [11] are the overall architecture of an MOP framework, as well algorithms that implement engines for past time and future time linear temporal logics (ptLTL) and for extended regular expressions ERE.

Let us recall that, in Runtime Verification, the program is checked at runtime against its specified behaviour. This, combined with the fact that MOP frameworks abstract away the internals of the formalisms used, can have important implications for the software engineering practitioner, some of which being:

- It allows developers who know how a given API works to focus on the easier task of creating and validating specifications, relative to that of translating the specifications into state machines and monitors;
- Once a specification is validated, developers should have a high confidence that the property/behaviour that has been specified is being enforced throughout the several parts of the software wherein that property or behaviour can be manipulated/triggered, irrespective of the complexity of the system [30]. This is because, when the RV tool is run, we will know that the code is being executed, and that properties will not go unnoticed by the program, unless the specification itself is not currently designed to detect a known issue. This is because, in contrast to the static methods outlined in Chapter 2, in RV, the program is actually run against the specification.

In contrast with what we have observed about static analysis [18], in a runtime verification scenario, false positives are not necessarily inherent to the design. This is, again, because we are sure that the program is being executed, instead of approximations being generated and analysed. For that, false positives do not necessarily have to be present in RV.

In static analysis, thus, the presence of a positive can mean one of three things:

- The violation is a true positive;
- The violation is a false positive, because the approximation generated for the program, when analysed, raised a warning where there should not be one;

- The violation is a false positive, because the specification is incorrect.

In contrast, in RV, the presence of false positives must necesarily mean the specification is incorrect, because no approximation is generated, and the program is actually executed and monitored, leaving fewer room for doubt.

As we have seen, refining static analysis tools to reduce false positives introduces considerable technical overhead, and it can yield refniments that are too context-sensitive for practical purposes [10], which reinforces the need for an expressive and extensive means for specifying behaviour.

3.2.1 JavaMOP: Scalable RV in Practice

Following the conceptualisation of an MOP framework proposed by Chen and Ruso [11], JavaMOP was introduced for efficient runtime verification [30]. JavaMOP implements all three formalisms present in [11], as well as the following [13]:

- Finite State Machines - FSM
- Context-Free Grammars - CFG
- Simultaneous support for past and future operators for LTL
- Past Time Linear Temporal Logic With Calls and Returns - PTCaRet

Let us take a look at an adapted specification file to develop a high level understanding of what JavaMOP does under the hood.

```
package mop;

RandomStringPasswordSpec(String str) {
    event vo after(Object obj) returning(String s):
       call(public static String String.valueOf(Object)) &&
       args(obj) &&
       condition(RANDOMIZED_STRING === true) {
          // tell consumers the string is properly randomised
    }

    event gb after(String s) returning(char[] chars):
      call(public char[] String.toCharArray()) &&
      target(s) &&
      condition(RANDOMIZED_STRING === true) {
        // tell consumers the string is properly randomised
    }

    ere : vo gb
```

```
    @match {}
}
```

As can be seen, JavaMOP, specification files are created with the `.mop` extension, and we have to include the `mop` package. We use a class-like syntax to declare a specification, by assigning it a name, and defining parameter types. The events are declared using standard AspectJ syntax. Internally, JavaMOP combines such events into a multi-threaded monitoring system. The generated code is responsible for creating the final state machine which will specify the call sequence, and the monitor manager also handles concurrency internally. In this process, JavaMOP creates an intermediate `.java` file, which is ultimately converted into an AspectJ (`.aj`) file. As a final - and optional - step, JavaMOP allows us to create an agent from the AspectJ file. This `.jar` file can be set via the command line when executing a test suite, basically informing the Java Virtual Machine that the generated agent is going to be used.

The `ere` section works exactly as it does in CrySL: it is an Extended Regular Expression that allows us to define sequences of events. In contrast to CrySL, though, JavaMOP also provides plugins for other formalisms, such as Finite State Machines (FSM) and Control Flow Graphs (CFG) [13].

For some of the supported formalisms, we are allowed to use the `@match` and `@fail` clauses, which allow us to execute code, or to communicate whether the execution conformed to the specification.

Leveraging the aforementioned parallel between formal specifications and aspects, the creators of JavaMOP implemented the automatic synthetisation of monitors, effectively enabling the possibility of creating “raw MOP specifications" [30], thereby leveraging AspectJ’s native compiler and weaver to generate code that represents the monitors. Extensive validation of its effectiveness showed that JavaMOP frequently incurs in acceptable runtime overhead that is no greater than 10%, therefore making it a potential candidate for runtime verification at scale [13].

3.2.2 Concluding Remarks

The literature review of the theory, methods and tooling available for efficient Runtime Verification [13] [11] [12], [30], [25], [24], [26] suggests that it is in fact possible to construct an artefact for runtime verification of cryptography API specificactions. In the next chapter, we outline the main construction steps, as well as the experimental setting we designed in order to conduct the empirical assessement of our method.

Chapter 4

Research Outline and Method

This chapter outlines the process employed in the creation of the MOP specifications that will serve as input to our crypto API Runtime Verification tool. Having CrySL specifications as a starting point, we are going to see how the respective JavaMOP specs are created. We also informally discuss mappings between CrySL and JavaMOP constructs, and leave the automation of the translation process as future work. The chapter concludes with a workflow view of the architecture or our tool, alongside a brief discussion of the abstractions we introduced that allowed us to express the proper expected behaviour of JCA classes.

4.1 Test-Driven Translation of Specifications From CRySL to JavaMOP

CrySL [9] provided a set of 49 specifications for how JCA APIs should be used. As an initial step in the development of our specifications, we are going to translate a subset of the specifications provided by CrySL. Our decision on which ones to translate was primarily based on how frequently the respective JCA class was used. A secondary criterion was the prevalence of violations, as reported by CrySL and CryptoGuard.

Note that, even though we are opting to translate specifications generated by CrySL due to its authors' expertise in cryptography and, specifically, in the JCA, we are not strictly required to do so. This is a design decision that was made in order to attain ourselves to the overall question of determining whether Runtime Verification is an efficient method for detecting crypto API misuses, whilst still having a trustworthy source of rules for the correct usage of JCA's classes. So long as one knows how the APIs are expected to be used, one could perform the test-driven process outlined below without referring to external sources. Provided that we have test sets to serve as ground truth, we can apply

TDDD to create our own specifications. In addition, our approach should be flexible enough to allow us to adapt our specifications in the (unlikely) case where we disagree with some of CrySL's guidelines.

4.1.1 Translating CrySL Specifications into JavaMOP

In this section, we present CrySL's entire specification for the `MessageDigest` class, which shall be used to illustrate the incremental, test-driven development process we applied for the creation its equivalent JavaMOP spec. Our process consisted in the standard red-green-refactor iterative approach to writing software [32], which basically consists of three phases:

- Create a test case and run the program. Since there is no implementation to be tested, the test will fail;
- Implement code that meets the behaviour expected by the test. This means writing code that either throws an expected error, or that represents a valid execution;
- Refactor the code as necessary;

This process is repeated with each of the test cases available in the test suite. In our case, the initial development of our specifications was made possible by test sets provided by Bodden et al [9]. This enabled us to gradually increase the complexity of our specifications, whilst ensuring that we would not introduce bugs into them. Let us now turn to our example.

```
SPEC java.security.MessageDigest

OBJECTS
        java.lang.String algorithm;
        byte preInputByte;
        byte[] preInput;
        int preOffset;
        int preLen;
        java.nio.ByteBuffer preInputByteBuffer;
        byte[] input;
        int offset;
        int len;
        byte[] output;

EVENTS
        g1: getInstance(algorithm);
        g2: getInstance(algorithm, _);
```

```
        Get := g1 | g2;

        u1: update(preInputByte);
        u2: update(preInput);
        u3: update(preInput, preOffset, preLen);
        u4: update(preInputByteBuffer);
        Update := u1 | u2 | u3 | u4;

        d1: output = digest();
        d2: output = digest(input);
        d3: digest(output, offset, len);
        DWOU := d2;
        DWU := d1 | d3;
        Digest := DWU | DWOU;

ORDER
        Get, (DWOU | (Update+, Digest))+

CONSTRAINTS
        algorithm in {"SHA-256", "SHA-384", "SHA-512"};
        length[preInput] >= preLen + preOffset;
        preOffset >= 0;
        preLen > 0;
        length[output] >= len + offset;
        offset >= 0;
        len > 0;

ENSURES
        generatedMessageDigest[this] after Get;
        digested[output, _];
        digested[output, input];
```

Expressing CrySL's constructs in JavaMOP

In order to express that we want to specify the behaviour of the `MessageDigest` class, we use a constructor-like syntax in JavaMOP:

```
package mop;
import java.security.MessageDigest;
MessageDigestSpec(MessageDigest digest) {
    //events, objects, call sequence, @match/@fail handlers
}
```

In JavaMOP, the class declaration defines which JCA class is to be specified, thereby having the same meaning as CRySL's SPEC section. In our case, the type of the ar-

gument that is passed to the specification file defines that the `digest` object from the `MessageDigest` class is going to be monitored. This is why JavaMOP is considered a parametric runtime verification framework. Now let us proceed to incrementally creating our MOP spec.

Notice that, in its `EVENTS` section, CrySL allows us to specify conjunctions of events, such as Get := g1 | g2;, `Update := u1 | u2 | u3 | u4`; and `Digest := DWU | DWOU`. This allows for a relatively clean way of defining the extended regular expression in the `ORDER` section[1].

Since CrySL provides this neat way of expressing conjunctions, even though its $ORDER$ section reads relatively simple, it does in fact generate several possible different call sequences. When developing its JavaMOP couterpart, for each possible path, one is required to create test cases that exert that path. Here are a few examples of sequences of method calls that conform with CrySL's specification for MessageDigest:

- $g1 \rightarrow d2$;
- $g1 \rightarrow u1 \rightarrow d1$
- $g2 \rightarrow u2 \rightarrow u2 \rightarrow d1$;
- $g2 \rightarrow u3 \rightarrow d3$

Let us pick the first example sequence above of a valid call sequence, in which only two events are involved, build a partial specification, and apply TDD to give us confidence that this partial specification works properly.

The simplified JavaMOP spec for `MessageDigest` that should check whether $g1 \rightarrow d2$ works as specified is the following:

```
package mop;
import java.security.MessageDigest;
import java.util.List;
import br.unb.cic.mop.eh.*;
import br.unb.cic.mop.ExecutionContext;
import br.unb.cic.mop.ExecutionContext.Property;

MessageDigestSpec(MessageDigest digest) {

    List<String> algorithms = Arrays.asList("SHA-256", "SHA-384", "SHA-512");
    MessageDigest md = null;
    String currentAlgorithmInstance = "";
```

[1]Relative to the JavaMOP framework, this was an advantage we observed in expressiveness, as trying to create such conjunctions from outside the ERE itself in JavaMOP would cause errors whose trace made it difficult to replicate these conjunctions as present in CrySL. For that reason, our JavaMOP EREs ended up more explicit, handling the conjunctions inside the ERE itself, and looking more complicated.

```
                                                                                    13
    event g1 after(String alg) returning(MessageDigest digest):                    14
        call(public static MessageDigest MessageDigest.getInstance(String))        15
        && args(alg) && condition(algorithms.contains(alg.toUpperCase())) {         16
            md = digest;                                                            17
            currentAlgorithmInstance = alg;                                         18
        }                                                                           19
                                                                                    20
    event d2 after(MessageDigest digest) returning(byte[] out):                    21
        call(public byte[] MessageDigest.digest(byte[])) &&                         22
        target(digest) {                                                            23
            if (!algorithms.contains(currentAlgorithmInstance.toUpperCase())) {     24
                ErrorCollector.instance().addError(new ErrorDescription(ErrorType.  25
                    ↪ UnsafeAlgorithm, "MessageDigestSpec", "" +
                __LOC,"expecting one of {SHA-256, SHA-384, SHA-512} but found " +    26
                    ↪ currentAlgorithmInstance + "."));
            }                                                                       27
            ExecutionContext.instance().setProperty(Property.DIGESTED, out);        28
        }                                                                           29
                                                                                    30
        ere : g1 d2                                                                 31
                                                                                    32
@fail {                                                                             33
    ErrorCollector.instance().addError(new ErrorDescription(ErrorType.              34
        ↪ InvalidSequenceOfMethodCalls, "MessageDigestSpec",  "" + __LOC));
    ExecutionContext.instance().unsetObjectAsInAcceptingState(md);                  35
    __RESET;                                                                        36
    }                                                                               37
                                                                                    38
@match {                                                                            39
    ExecutionContext.instance().setObjectAsInAcceptingState(md);                    40
    }                                                                               41
}                                                                                   42
```

The `args()` function binds the `String alg` object, therefore allowing us to establish **condition**s about it, which, when met, allow us to execute custom blocks of code. Notice also that AspectJ's syntax provides us with the `returning` directive, basically meaning that a successful call to the method in question ought to return the expected type. This is also in alignment with the first statement of the ENSURES section in the CrySL spec, which expects a call to `MessageDigest.getInstance()` to actually return a digest object.

The simplest test cases that test whether the call sequence is valid are shown below. Please note that this is by no means an extensive test set. We could, for instance, change the type of the input to be digested, or use a different (potentially invalid or unsafe)

algorithm - such as SHA-1 - when instantiating the digest object. Moreover, since our ERE specifies that exactly one call to each method should happen, inserting redundant calls should also raise errors.

```
@Test
    public void messageDigestSimplesPathSuccess() throws NoSuchAlgorithmException {
        // proper g1 -> d2 sequence
        byte[] input = "StringToBeEncrypted".getBytes();
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        byte[] out = md.digest(input);
        Assertions.hasEnsuredPredicate(out);
        Assertions.mustBeInAcceptingState(md);
    }

    @Test
    public void messageDigestSimplesPathFailure() throws NoSuchAlgorithmException {
        // improper g1 -> d2 sequence: missing call to d2
        byte[] input = "StringToBeEncrypted".getBytes();
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        Assertions.hasEnsuredPredicate(out);
        Assertions.mustBeInAcceptingState(md);
    }
```

In order to validate whether all the predicates involved in the encryption are valid, we introduced another validation class called *Assertions*, which extends the `org.junit.Assert` class. As its name suggests, we use it to ensure that properties we expect to be met by the end of the computation do in fact hold. The first type seen in the test cases above - *hasEnsurePredicate* - serves the function of making sure that CrySL's equivalent `ENSURES` clauses are met, while *mustBeInAcceptingState* refers to whether the call sequence was valid or not. In the two aforementioned examples, the first test would finish gracefully, whilst the second would fail, since the `md` object is **not** going to be in an accepting state, since the call sequence was invalid.

The `ORDER` section maps one to one to our *ere*[2]. Let us examine the entire event definitions for the two events from the specification above to introduce the abstractions we implemented to provide equivalent meaning for the rest of *CrySL*'s constructs.

Note that, in order to express the guarantees - CrySL `ENSURES` clause - provided by the class being specified, we introduced the *ExecutionContext* class, a singleton oject which alllows us to set and remove properties about the objects, as well as to keep track

[2]Note that, since JavaMOP implements several different formalisms, which all translate to an internal state machine, it allows us to express event orders in ways other than EREs. For example, we could have used fsm for Finite State Machine, `cfg` for Control Flow Graph, or any of the formalisms supported by JavaMOP [13]

of and communicate such properties throughout the execution. We explain its purpose and functionality further in the architecture section.

After we define the `events`, we bind names to them: `g1` and `d1` in our example. We are able to specify whether we want to perform checks `before` or `after` a `MessageDitest` object is successfully returned. The arguments passed in are the objects we want to monitor, which means that, in this event, we are interested in monitoring the `String alg` that is passed in to `MessageDigest.getInstance()`. The `args()` construct performs such binding for us. The `condition()` construct allows us to insert Java statements, which provides the ability to express requirements, properties or states, thus carrying the same meaning as CrySL's `REQUIRES`. The curly braces following the condition define a block of code in which we can execute custom code, allowing us to collect, save and propagate context information, and potententially (though not explored in this work) alter program behaviour at runtime.

In the case of EREs, JavaMOP supports both the `@match` and `@fail` clauses, which are used, respectively, to execute code whenever there is a complete match, or as soon as an unexpected event occurs[3]. Notice also that, when `@fail` is reached, we use another abstraction of our own creation, called the *ErrorCollector*, which allows us to not only catch method calls that do not conform with the ERE that defines the call sequence, but also to raise errors when incorrect types and values are used.

Enhancing the MOP specifications

Even though we described a translation of a relatively simple call sequence, the overall process of enhancing the specs to encompass more paths follows the same systematic process:

1. Break the ERE down into all[4] the reasonable call sequences it can produce;
2. For each call sequence to be specified, we write test cases that are expected to pass and fail;
3. Run the tool against the tests, and make any necessary adjustments exposed during the execution;

[3]Note the *___RESET* statement here, which is a low-level operation to reset JavaMOP's `rv-monitor` to its initial state. For further discussions on JavaMOP's special variables, as well as for reference on which formalisms support which clauses, please refer to https://web.archive.org/web/20171214085644/http://fsl.cs.illinois.edu/index.php/JavaMOP4$_{S}yntax$.

[4]Strictly speaking, this would not be possible when the + and * operators are present, as the number of calls to the same method can be arbitrarily large. We simplify in this case, and do not repeat method calls in our tests more than two times.

4. Once confident that a path has been properly tested, enhance the ERE to contain a new path. Repeat until all the paths defined by the `ORDER` ERE have been covered.

Automating the Translation - An Informal Discussion

The systematic process outlined above raises an important question, which shall only be informally addressed in this text, but which we believe to be an important topic for future work:

- *Is it possible to automate the translation of CrySL specifications into JavaMOP?*

Our experience developing the 22 MOP specifications that we have at the time of this writing suggests that, without any adaptations, it would be difficult to perform automatic translations. However, the set of auxiliary modules, internal functionality, data structures and abstractions that we developed to augment the JavaMOP framework allowed us to express most CrySL's directives in our MOP context. The most relevant ones for this discussion are:

- *ExecutionContext*: Originally created to provide means to express CrySL's `ENSURES` and `REQUIRES` clauses, this data structure allows us to store and propagate information throughout the execution. This module allows us to set and remove properties of the objects used at runtime. It is via *ExecutionContext*, for example, that we record and communicate about the state of a key, thus allowing us to know whether it has been properly randomised or not. In conjunction with the JavaMOP agent that is generated, this class also assists us in deciding if the execution sequence is valid. At each stage of the computation, if the agent observes a violation of the expected order of method calls, *ExectionContext* is requested to register such violation[5];
- *ErrorCollector*: As execution unfolds, one or more violations may occur. This singleton object is frequently used inside the bodies of the blocks of code that are executed when events are captured, allowing us to report errors exactly as they occur;
- *CipherTransformationUtil*: Since the instantiation methods in the Cipher class usually accept a transformation string - as opposed to simple algorithm strings, for most other classes - we had to introduce this utility module so as to have a way to

[5]The need for *ExecutionContext* in this case is due to the internals of rv-monitor, the engine used by JavaMOP to synthethise monitors. Since we want to be able to know all the errors during the entire flow of execution of a given class being specified, we need to ___RESET rv-monitor whenever we find an invalid call sequence. In this case, *ExecutionContext* makes sure this information is not lost during the resets.

express the functions `alg()`, `mode()` and `pad()`, which are extensively used in the `CONSTRAINTS` and `ENSURES` sections of CrySL's specification for Cipher.

- *Assertions*: This module, which inherits from JUnit's `Assert` class, contains functions that decides on whether a call sequence is valid, whether predicates defined in CrySL's `REQUIRES` sections hold, and whether errors have been raised throughout the execution.
- A *Logger* that reports the results into tabular form. The outputs produced by the *Logger* module are `.csv` files that contains information about which specification was violated, the kind of violation, and information about the location of the violation. Such data will be paramount for the empirical assessment of our framework.

Although we have not engaged in automating the translation, we applied the systematic process outlined above, together with our helper modules, and we successfully developed our 22 specifications. Nonetheless, it is important to mention that we only translated about half of all the CrySL specifications available. Therefore, even though our visual inspection of the files we did not translate seem to pose no threat to our current method, it would be irresponsible to affirm that such translations can be automated before we provide strong evidence that they are in fact possible. Moreover, due to the inherent challenges involved in the development of compilers and translators [33] we are unable to make statements about the difficulty of developing such transpiler. Below are code snippets that provide mappings between constructs in the two languages/frameworks. Again, these should not be treated as exact one-to-one relationships between them; instead, the reader should use them as a starting point, and we encourage a more thorough analysis before proceeding to creating an automatic translator.

Defining the class under specification is straightforward in both contexts:

```
//CrySL
SPEC java.security.MessageDigest

//JavaMOP
package mop;
import java.security.MessageDigest;
MessageDigestSpec(MessageDigest digest) { ... }
```

The `CONSTRAINTS` section is handled in JavaMOP by leveraging AspectJ's `condition()`, inside an event declaration:

```
//CrySL
CONSTRAINTS
    algorithm in {"SHA-256", "SHA-384", "SHA-512"};

```

```
    EVENTS
        g1: getInstance(algorithm);

//JavaMOP
package mop;
import java.security.MessageDigest;
MessageDigestSpec(MessageDigest digest) {
    List<String> validAlgorithms = Arrays.asList("SHA-256", "SHA-384", "SHA-512");
    MessageDigest md = null;
    String currentAlgorithmInstance = "";

event g1 after(String alg) returning(MessageDigest digest):
    call(public static MessageDigest MessageDigest.getInstance(String))
    && args(alg) && condition(validAlgorithms.contains(alg.toUpperCase())) {
        ...
    }
}
```

Even though CrySL provides an explicit separation between the `OBJECTS` and the `EVENTS`, in JavaMOP, we have to combine the ideas, and so we have to express both combined insided an event declaration, as per AspectJ syntax. Let us take a look at an example.

```
//CrySL
OBJECTS
    byte[] preInput;
    int preOffset;
    int preLen;
    ...

EVENTS
    u3: update(preInput, preOffset, preLen);
    ...

//JavaMOP event
event u3 after(byte[] preInput, int preOffset, int preLen, MessageDigest digest):
 call(public int MessageDigest.digest(byte[], int, int)) &&
 args(preInput, preOffset, preLen) &&
 target(digest) {
   ExecutionContext.instance().setProperty(Property.DIGESTED, preInput);
}
```

As can be seen, the objects that are used by the method under specification are declared inside the `after()` directive. The `call()` construct specifies what is the method to be monitored. The arguments of the method - `MessageDigest.digest` in our case -

take only the types, and the binding is handled by `args()`. Here, the necessity of the `ExecutionContext` singleton becomes more evident, as we would have no native way in JavaMOP to `ENSURE` that the input byte array has been properly digested. Let us know use a fictitious event declaration to examine how to communicate when a CrySL constraint is not met.

```
//CrySL
OBJECTS
java.lang.String alg;
...

EVENTS
// Since this is an invalid event, it is not listed in CrySL

CONSTRAINTS
    alg in {"SHA-256", "SHA-384", "SHA-512"};

//JavaMOP event
event g3 after(String alg) returning(MessageDigest digest):
      call(public static MessageDigest MessageDigest.getInstance(String))
      && args(alg) && condition(!validAlgorithms.contains(alg.toUpperCase())) {
        ErrorCollector.instance().addError(new ErrorDescription(ErrorType.UnsafeAlgorithm,
            ↪ "MessageDigestSpec", "" + __LOC,
         "expecting one of {SHA-256, SHA-384, SHA-512} but found " + alg + "."));
           currentAlgorithmInstance = alg;
    }
...
@fail {
       ErrorCollector.instance().addError(new ErrorDescription(ErrorType.
           ↪ InvalidSequenceOfMethodCalls, "MessageDigestSpec",  "" + __LOC));
       ExecutionContext.instance().unsetObjectAsInAcceptingState(md);
       __RESET;
    }

@match {
       ExecutionContext.instance().setObjectAsInAcceptingState(md);
}
```

The `g3` event represents the instatiation of an invalid, or unsafe, algorithm. For example, if we call `MessageDigest.getInstance("SHA-1")`, this event will be triggered, which will, in turn, cause our `ErrorCollector` to report that a violation has occurred. Since this event is technically invalid - due to the use of an unsafe algorithm - it is not present in CrySL's `ORDER` section. This is where the match and fail handlers come in handy, as

they allow us to either report an `InvalidSequenceOfMethodCalls`, or to in fact accept the computation, in the case of a match.

This concludes our informal discussion on how to establish mappings between the constructs in CrySL's main directives into JavaMOP. We would like to emphasise that these are only **guidelines** used by our group of researchers in the development of our MOP specs. It is also important to stress that The topic of formally mapping the constructs and creating an automatic translator is subject to a research endeavour of its own, and is left as future work.

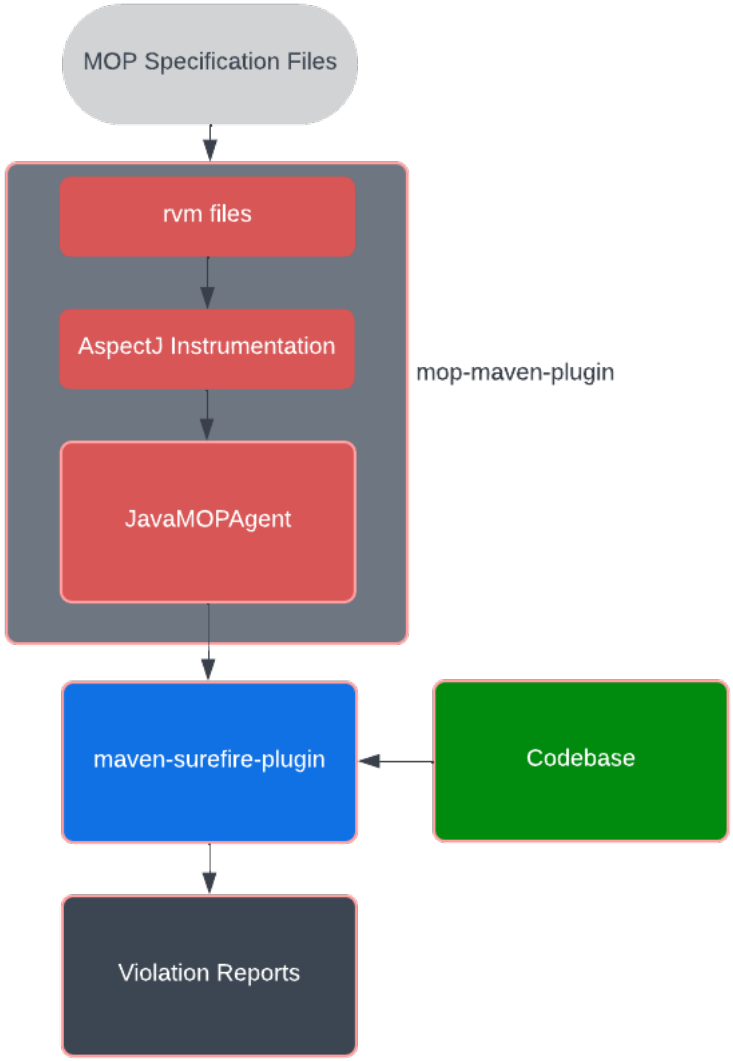
4.2 Architecture Overview

Figure 4.1 depicts the major components, as well as the overall flow of information for RVSec. Let us briefly describe each component.

- As observed in the previous section, the input to our program is a set of MOP specification files using JavaMOP, which leverages AspectJ syntax for pointcuts and advices;
- The $mop - maven - plugin$ is a set of utility functions that abstracts away some of the low level, intermediate steps required by JavaMOP, such as the creation of the monitor files and their subsequent merging, the creation of the monitor libraries, and the creation of a JavaMOP agent. As shown by the picture, Runtime Verification Monitor files serve as input to generate a single AspectJ file, wherein each monitor is translated to an AspectJ advice and body. Finally, its last step consists in the generation of the JavaMOPAgent, a `.jar` file;
- The aforementioned agent is then supplied to $maven - surefire - plugin$, an open source plugin that offers utilities for assisting during the testing lifecycle. In particular, this plugin allows us to easily customise some of the Java Virtual Machine settings, and it also permits us to specify agents to be using throughout the execution;

It shall be noted that, since the generation of the agent is decoupled from the programs represented by the green box in the picture, the agent can be reused arbitrarily, only needing update when the MOP specifications are updated. This means that, once we are confident that the specifications are correct, we are able to run the test cases against them before production code is deployed, thus introducing one more layer of validation into the integration and deployment pipeline.

Figure 4.1: RVSec's pipeline



The output produced by the *Logger* module is a *csv* that contains information about which specification was violated, the kind of violation, and information about the location of the violation. Such data will be paramount for the empirical assessment of our framework, which is outlined next.

Chapter 5

Empirical Assessment

5.1 Evaluation of Empirical Data

This chapter presents both quantitative and qualitative discussions regarding the data that was collected during our empirical assessment. We seek to provide answers to four research questions, which shall be outlined throughout the chapter. We also provide examples of cases where the different tools produced different results, along with design decisions that we employed in order to increase the accuracy metrics of RVSec. We discuss some hypotheses as to why discrepancies in the results arise, and conclude by summarizing our findings, as well as some potential threats to the validity of our approach.

5.2 Experiment Outline

This section introduces the questions that will guide the researchers in the evaluation of the empirical study herein proposed. After eliciting the research questions that are relevant to our discussion, we will describe the expriment setting, as well as the methods we will employ in order to validate or refute our hypotheses.

Since our experiment entails a comparative study between RVSec and the state-of-the-art static analysis methods available for crypto API detection, we provide the data generated by our experiments, and we also discuss the limitations, advantages and drawbacks of each of the frameworks analysed during the experiments. Throuhgouht the discussion, we provide examples where some tools fail whilst others thrive, and, when possible, provide insights as to why this is the case.

5.2.1 Experiment Objectives and Research Questions

Our empirical study aims to assess both the **accuracy metrics** of RVSec when compared to CogniCrypt and CryptoGuard, as well as the potential **implications** of applying runtime verification methods to the detection of cryptography APIs misuse. We are also interested in measuring the costs of our approach. The next few paragraphs describe the specific questions we are going to investigate during our experiment. The following sections describe the procedures and methods applied to collect data to generate insights about the questions.

The overarching questions we seek to answer in our empirical assessment are:

- **R.Q.1** - *Accuracy Metrics*: How does RVSec compare to state-of-the-art tools for static detection of crypto API misuse in terms of accuracy metrics? In particular, how does the *recall* and *precision* of RVSec compares to those of CrySL and CryptoGuard, when the three tools are run on the same input code?
- **R.Q.2** - *Performance*: Is there any significant increase in execution time or memory consumption when running test suite code that has been instrumented with our JavaMOPAgent?
- **R.Q.3** - *Correlation to coverage*: Is there a statistically significant correlation between the coverage of test cases and the number of violations reported by the analysis tools?
- **R.Q.4** - *Comparison*: What are the relative strengths and weaknesses of RV, compared to static crypto API misuse detectors?

In order to produce answers to the first three research questions, we are going to perform quantitative assessments, whereas we are going to follow a more qualitative approach when trying to answer RQ4.

A legitimate concern that is always present in dynamic analysis is that RV techniques may incur in high overhead costs, especially when compared to static approaches. So answering RQ2 provides a sense of the cost of using RV for crypto API misuse detection. Moreover, since RV can only detect violations parts of code that are actually executed, answering RQ3 allows us to see the impact of code coverage on the accuracy of RVSec for detecting crypto API misuses. Finally, answering RQ4 helps us to see ways to further improve RVSec for use in crypto API misuse detection, and to investigate the potential synergy that could result from combining static and dynamic detection of crypto API misuses in the future. We conclude our qualitative assessment with some example techniques we used to increase RVSec's precision and recall.

5.2.2 Study Setting

Upon completion of the test-driven development of our MOP specifications, we are going to execute RVSec, CrySL, and CryptoGuard against three benchmark test sets, whilst collecting data and metrics that we expect to help answer the research questions outlined above.

Experiment Design and Preliminary Threats to Validity

Since benchmarks serve as ground truth for validating observations and measurements, we also reviewed the literature available searching for benchmarks provided by different research groups. The three benchmarks to be used in our experiments are:

- ***MASCBench*** is a collection of 30 small Java programs that isolate crypto API misuses introduced via domain-specific mutation operators [34]. The benchmark comprises *minimal* versions of programs with crypto API misues coming from 13 mutated open source Android apps and four sub-systems of Apache Qpid Brokerj. These minimal versions contain only 512 lines of Java code.
- ***SmallCryptoAPIBench*** consists of 187 test cases (3735 source lines of code in total) that cover simple and complex scenarios of crypto API uses and misuses. *SmallCryptoAPIBench* is a smaller version of *CryptoAPIBench*, a handcrafted benchmark that Afrose et al. designed to compare the performance of static analysis tools in detecting crypto API misuses [35]. We removed from *CryptoAPIBench* 16 test cases not related to the JCA library—hence the name *SmallCryptoAPIBench*.
- ***ApacheCryptoAPIBench*** is a dataset of real crypto API misuses collected from Apache project modules [35] (Table 5.1 presents more detailed information about them). According to the available ground truth for this benchmark, there are a total of 74 crypto API misuses in the Apache modules of Table 5.1. Table 5.1 also highlights the specific revision of each project we used in our assessment, as well as the number of test cases available, the size of the projects (in terms of source lines of code), the number of RVSec monitors generated and the number of events the monitors capture during the execution of the projects' test suites.

We use all three benchmarks in search for answers to RQ1 and RQ4. To do so, we integrate RV into the build process of the small Java programs in *MASCBench* and *SmallCryptoAPIBench*. Our integration uses reflection to invoke the programs while also using a JavaMOP Java agent to monitor the executions for crypto API misuses. The output from our pipeline is a set of violations that signal the detection of crypto API misuses. We also export *MASCBench* and *SmallCryptoAPIBench* as `.jar` files that can

be analyzed by CogniCrypt and CryptoGuard, and then computed precision and recall using ground truth data that are available from *MASCBench* and *SmallCryptoAPIBench*. We discarded from ou analysis programs in these benchmarks whose vulnerabilities are not due to the incorrect use of the JCA API (e.g., programs that use the HTTP protocol instead of HTTPS when instantiating URIs) or those that do not contain an explicit `main` method. Also, we fixed several test cases in *MASCBench* and *SmallCryptoAPIBench* that raised runtime exceptions—for instance, when a test case refers to an invalid key store, we had to replace it with a reference to a valid one.

We used the developer-provided build configurations in the 10 original Apache projects of *ApacheCryptoAPIBench*, which we used to answer all four research questions. Unfortunately, *ApacheCryptoAPIBench* only describes the Maven artifact's non-stable version of the programs (i.e., `SNAPSHOT` versions). Since there can be different revisions (e.g., a Git commit ID) of a non-stable version, we checked out and built the most recent revision for a given non-stable version, and we display our results in the second column of Table 5.1. We needed to implement small fixes to some projects to make the build succeed. These changes involved either commenting out pieces of code that did not compile, or repairing some library dependencies. We will make these fixes available as supplementary material. Once all the issues listed above were fixed, we ran the entire test suites of these projects 10 times each in two configurations: (a) **without** RVSec, and (b) **with** RVSec. For each test-suite execution, we recorded the total execution time, the test case coverage, and (for the second configuration) the crypto API misuses detected. We also ran CogniCrypt and CryptoGuard on the `.jar` files from these projects to collect statically detected crypto API misuses. We could not fix the build process of two (Taverna Workbench and TomEE Container) out of the 10 Apache modules in the original *ApacheCryptoAPIBench*. For this reason, we discarded these two modules from our analysis.

Project	Module	Revision	TCs	SLOC	RVSec Monitors	Events	Misuses
ActiveMQ Artemis	artemis-commons	5ab187b	110	11 737	14	73	15
Directory Server	apacheds-kerberos-codec	155bd94	376	42 185	135	2379	19
ManifoldCF	mcf-core	9573dc4	5	21 281	39	744	3
DeltaSpike	deltaspike-core-impl	d95abe8	155	13 515	31	3954	2
Meecrowave	meecrowave-core	3780f1c	19	6788	23	8998	3
Spark	spark-core_2.11	9ff1d96	2045	164 335	123	181548	27
Tika	tika-core	6f33bae	222	23 207	6	294	0
Wicket	wicket-util	dbd86d9	237	20 220	52	405	5

Table 5.1: Information about *ApacheCryptoAPIBench* artifacts used in our analysis, including the number of crypto API misuses according to the original ground truth

Notes on Limitation of Benchmarks

Before we continue the analysis of the data generated by the experiments we conducted, it is important to be transparent about the potential threats the researcher faces when using benchmarks as ground truth data to validate or refuse hypotheses. Below is a non-exhaustive list of factors that can hinder our ability to produce consistent and trustworthy results:

- **Bias**: Although we, as scientists, strive to be objective in our assessments, as humans, we are prone to bias [36]. As Nobel Prize Laureate Daniel Kahneman States, *"When people believe a conclusion is true, they are also very likely to believe arguments that appear to support it, even when these arguments are unsound."* [37]. When designing benchmarks to serve as ground truth for crypto API detection, researchers will naturally carry their biases into the projects. In practice, this means that, not only are researchers prone to inserting bias into the creation of the detection tools, but they are also - consciously or not - likely to create test benchmarks that are in accordance with such biases, therefore increasing the chance that confirmation bias is present in the test sets.
- **Disagreements on specifications**: Although in many cases the JCA defines the proper use of the classes and interfaces it makes available to engineers, its documentation is not always explicit about certain rules, leading researchers to establish their own rules for how a certain API is expected to be use. A notorious example that was exposed by our research is exposed by the specification of the `PBEKeySpec` class. Whereas the researchers behind the CrySL project define that its constructor must take 10000 as the minimum value of the `IterationCount` parameter, the authors of CryptoGuard consider 1000 enough. As such, when all the tools are run against the different benchmarks, there can arise discrepancies in the reports. This is also compounded by the aforementioned biases that may be introduced into the benchmarks;
- **Human Error**: Since we do not yet have reliable ways to automate the process of creating benchmarks, researchers must engage in the laborious process of defining and validating benchmark data. Therefore, even with processes like coss-validation and repeated reviews, errors can be present in ground truth data, especially as the data sets get larger.

5.2.3 Experiment Procedure

In this section, we describe the methods to be employed during our experiment, with the goal of generating data and insight about our research questions.

To answer RQ1, we compute the Precision, Recall, and F-measure of RVSec, CogniCrypt, and CryptoGuard on *MASCBench*, *SmallCryptoAPIBench*, and *ApacheCryptoAPIBench*. Prior work used these metrics to compare static detectors of crypto API misuses [9, 10]. Given the number of true positives (TP) and false negatives (FN) in our assessments, these metrics are given by the following equations.

$$Precision = \frac{TP}{TP + FP} \tag{5.1}$$

$$Recall = \frac{TP}{TP + FN} \tag{5.2}$$

$$F - Measure = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{5.3}$$

To measure RV overhead (and to answer RQ2), we ran the test suites 20 times for each project in *ApacheCryptoAPIBench* (10 repetitions for each configuration). That is, we computed the average time of 10 executions using the **with** RVSec configuration, but we also performed 10 executions using the **without** RVSec configuration, for every project in *ApacheCryptoAPIBench*. Considering TRV and $TBase$ the average time to run the test cases **with** and **without** RVSec, we estimate the overhead using the Eq(5.4):

$$Overhead = 100 \times \frac{TRV - TBase}{TBase} \tag{5.4}$$

Notes on limitations of our approach for answering R.Q.2: We opted to only conduct overhead analysis for the real-world projects that we analysed from *ApacheCryptoAPIBench*. This was mainly due to the following reasons:

- Running the suite on the small set of cases provided by *MASCBench* took almost no time. Therefore, the measurements would be too noisy and unrealistic, thus providing no real insight;
- Although *SmallCryptoAPIBench* contained a lot more test cases, the intricacies of `rv-monitor` - a low level library used by JavaMOP to synthetise monitors - was very sensitve to the environment (operating system and Java Virtual Machine) upon which it was running, which, after some point in the development of our work, prevented the author from running RVSec against the benchmark. This forced us to resort to the assistance of other contributors in our research group;

- Each round of execution on the 8 projects from *ApacheCryptoAPIBench* that built succesfully would take several hours to produce its output. The 20 runs executed in total - 10 with RVSec's agent enabled and the other 10 regular runs - took more than 14 hours to run. In order to spare the efforts of the contributor who was able to run the tools against all benchmarks, we opted to prioritise the real world projects;
- Our tool was run on a regular user's computer, which was running several other processes which might have introduced variance into the measurements of the 10 runs performed with instrumentation. Since the runs would take several hours to complete, we had to restrict ourselves to a shorter number of runs, which made it difficult to migitate the noise we observed during the experiment;

Finally, to answer RQ3, we measured statement coverage, branch coverage, and method coverage in the *ApacheCryptoAPIBench* projects using the JaCoCo tool. We combined these coverage metrics with Precision and Recall to carry out the Spearman Correlation test [38]. Our goal is to investigate how code coverage impacts the accuracy of RV for detecting crypto API misuses.

Upon collection of the aforementioned metrics, as well as the cross-validation and analysis of the results, we will then be in a position to examine the shortcomings of our approach. As explained in previous sections, we are particularly interested in finding situations which would be difficult to express using our framework.

Finally, the knowledge, data and insights generated during the research and development of RVSec will provide information about useful heuristics and strategies for specifying crypto API behaviour via runtime verification.

5.2.4 RQ1: Accuracy

Accuracy results for *MASCBench*. Table 5.2 shows that RVSec achieves higher Precision, Recall, and F-measure than CogniCrypt and CryptoGuard on *MASCBench*. As shown in the "FP" column, none of the three tools reports a false-positive. So, they all have 100% Precision. However, RVSec produces a smaller number of false-negatives for *MASCBench*—two false negatives—while CogniCrypt and CryptoGuard reported eight and nine false negatives respectively. The two false negatives that RVSec reports were due to an issue in our specification, which was not correctly capturing an event in the RVSec version used in the experiments. Differently, false-negatives in CogniCrypt and CryptoGuard seems more related to limitations in their static analysis components, as shall be seen in Section 5.3.1.

Tool	TP	FP	FN	Precision	Recall	F-measure
RVSec	26	0	2	1	0.92	0.96
CogniCrypt	20	0	8	1	0.71	0.83
CryptoGuard	19	0	9	1	0.67	0.80

Table 5.2: Accuracy results for *MASCBench*

Accuracy results for *SmallCryptoAPIBench*. Table 5.3 shows that RVSec also achieves higher Precision, Recall, and F-measure than CogniCrypt and CryptoGuard on *SmallCryptoAPIBench*. Seven (out of eight) false negatives that RVSec missed relate to the use of hard-coded passwords for loading key stores (see lines 3 and 6). Failure to handle hard-coded strings is a limitation of our approach—we cannot check at runtime whether a variable has been initialized using a hard-coded string constant. An inspection of the CrySL specifications shows they use two types of check to detect hard-coded, or string values, as in the following examples:

- `neverTypeOf[password, java.lang.String];`
- `notHardCoded[password];`

Even though the first check is trivial to perform in JavaMOP, the second one leverages static analysis to decide on whether the password is in fact hard coded. The latter has yet to be reproduced in RVSec. The investigation of the possitility of expressing this in RV is left for future work.

We exemplify and qualitatively analyse the false positives and false negatives from all three tools in Section 5.3.2.

Tool	TP	FP	FN	Precision	Recall	F-measure
RVSec	122	7	8	0.94	0.94	0.93
CogniCrypt	110	29	20	0.79	0.84	0.81
CryptoGuard	114	18	17	0.86	0.87	0.86

Table 5.3: Accuracy results for *SmallCryptoAPIBench*

Accuracy results for *ApacheCryptoAPIBench*. We identified some threats while conducting the precision and recall comparison of RVSec and the static detectors for *ApacheCryptoAPIBench* [35]. First, we could not find the specific commits that were used to curate *ApacheCryptoAPIBench*, since only unstable versions—`SNAPSHOT` versions in Maven lingo—are specified. Many commits can map to a `SNAPSHOT`, so it is hard to find the specific commit that matches the one that the *ApacheCryptoAPIBench* authors use. Second, the provided ground truth misses essential information that we need for comput-

```java
public class ProgramExample {
  public static void main(String args[]) throws Exception {
    String key = "password";
    KeyStore ks = KeyStore.getInstance("JKS");
    URL cacerts = new File("testInput-ks").toURI().toURL();
    ks.load(cacerts.openStream(), key.toCharArray());
  }
}
```

Figure 5.1: Example of using a hard-coded password for loading a key store (which is considered insecure). RVSec does not detect this type of issue, while both CogniCrypt and CryptoGuard do detect.

ing Precision and Recall. For example, many *true positives* in *ApacheCryptoAPIBench* do not specify the Java class in which a crypto API misuse occurred.

Despite these threats, the *ApacheCryptoAPIBench* benchmark can still be a valuable source of data about how RVSec compares with the static detectors on real-world open-source projects where the static detectors were previously evaluated. So, our decision was to manually inspect all warnings the three tools generate for *ApacheCryptoAPIBench*. We consolidate the observations counting multiple warnings for the same object in a given class/method as a unique occurrence. This decision is necessary because RVSec, CryptoGuard, and CogniCrypt report warnings with different granularity. This procedure leads to a total of 192 warnings in our dataset, which were taken into account in our decision to create our own ground truth for this benchmark. Although creating a new ground truth might have introduced additional threats into our research, we are confident that this decision avoids several inconsistences we found in the previous *ApacheCryptoAPIBench* ground truth.

Note that, among the 192 warnings in our dataset, 44 warnings come from code in external dependencies. We excluded these warnings from our analysis because it is not trivial to validate these cases manually. RVSec also reports 14 warnings from unit tests. Since neither CogniCrypt nor CryptoGuard evaluate the artifact packages of the programs (`.jar` files) containing unit tests, we also removed these 14 warnings from our analysis. In the end, our curated dataset contains 134 warnings, as we summarize in Table 5.4.

Tool	Full Data Set	Curated Data Set
RVSec	64	40
CogniCrypt	72	58
CryptoGuard	56	36

Table 5.4: Summary of the warnings CogniCrypt, CryptoGuard, RVSec report for the *ApacheCryptoAPIBench*

After curating our dataset and ground truth, we compute the metrics Precision, Recall, and F-measure. Table 5.5 presents the results, which reveals that RVSec presents again a superior Precision. Nonetheless, when we consider both Precision and Recall, CogniCrypt presents the better performance (according to the F-measure estimate).

Tool	TP	FP	FN	Precision	Recall	F-Measure
RVSec	39	1	12	0.97	0.76	0.85
CogniCrypt	48	10	3	0.82	0.94	0.88
CryptoGuard	20	14	31	0.58	0.39	0.47

Table 5.5: Accuracy results for *ApacheCryptoAPIBench*

The Venn Diagram of Figure 5.3 helps to explain these results. First, since we built the ground truth from the set of warnings that at least one tool (RVSec, CogniCrypt, or CryptoGuard) generates, the true positive set (TP) with 51 warnings has intersection with at least one tool. The set of TP warnings all three tools identify contains 17 elements (33.33% of the elements in TP); while, the sets of TP warnings that at least two tools report contains 39 elements (76.47% of the elements in TP).

Note that, even though CryptoGuard implements a smaller number of rules, even in comparison with RVSec, it finds 58% of the true crypto misuses. Nonetheless, the smaller number of rules reflects into the CryptoGuard Recall (0.39). Besides that, even without improving the test suite of the projects, RVSec missed only 12 out of the 48 true positives that CogniCrypt report. Our manual analysis reveal that these false negatives are due to (a) the lack of test cases necessary to reveal the issues and (b) the lack of RVSec specifications for JCA classes that are not frequently used (`SecretKeyFactory` and `TrustManagerFactory`). This result suggests that RVSec is an alternative to static crypto API misuse detectors, even without the need to improve the test suite of real programs.

Conversely, RVSec reports one false positive, while CogniCrypt and CryptoGuard reports 10 and 14 false positives respectivelly. There is no intersection among the sets of false positives the tools report. Thirteen false positives from CryptoGuard come from a rule that considers the `java.util.Random` class insecure. Nonetheless, instances of this Java class are often used in non-cryptographic contexts. In fact, after a careful analysis, we identified that no instance of the `java.util.Random` class in the *ApacheCryptoAPIBench* leads to a software vulnerability. We marked these examples as false positives. Had we discarded these warnings from our analysis, the Precision of CryptoGuard for this benchmark would be significantly higher (0.95). We keep these warnings here because previous research incorrectly regards usage of the `java.util.Random` as true positives, which have misled previous results already published [39, 35].

```
public void parse(
            InputStream stream, ContentHandler handler,
            Metadata metadata, ParseContext context)
            throws IOException, SAXException, TikaException {
  Cipher cipher = Cipher.getInstance(transformation);

  Key key = context.get(Key.class);

  AlgorithmParameters params = context.get(AlgorithmParameters.class);
  SecureRandom random = context.get(SecureRandom.class);

  if (params != null && random != null) {
    cipher.init(Cipher.DECRYPT_MODE, key, params, random);
  } else if (params != null) {
    cipher.init(Cipher.DECRYPT_MODE, key, params);
  } else {
    cipher.init(Cipher.DECRYPT_MODE, key);
  }
  super.parse(
    new CipherInputStream(stream, cipher),
    handler, metadata, context);
}
```

Figure 5.2: Code snippet from the Tika project. In this case, a `Cipher` is being just prepared to future usage.

The 10 false positives CogniCrypt report involves tricky situations, which required significant amount of time to inspect and confirm. Our conclusion is that, in general, they appear not due to a bug in a CrySL rule or CogniCrypt implementation, but instead due to contextual information. For instance, we found in the Tika project pieces of code that just prepare a Java `Cipher` class for use by extensions of the project, not making explicit calls to methods such as `update` or `doFinal`. See an example in Figure 5.2. Currently, CogniCrypt reports an error in such a situation, even though we understand that deferring such calls is the intention of the developer. We mark these situations as false positives in our ground truth.

The results so far bring several findings, which we summarize bellow.

Finding 1

RVSec leads to a higher accuracy values when considering *MASCBench* and *SmallCryptoAPIBench*—two handcrafted benchmarks. RVSec also leads to the higher Precision in *ApacheCryptoAPIBench*, though with a lower Recall than CogniCrypt.

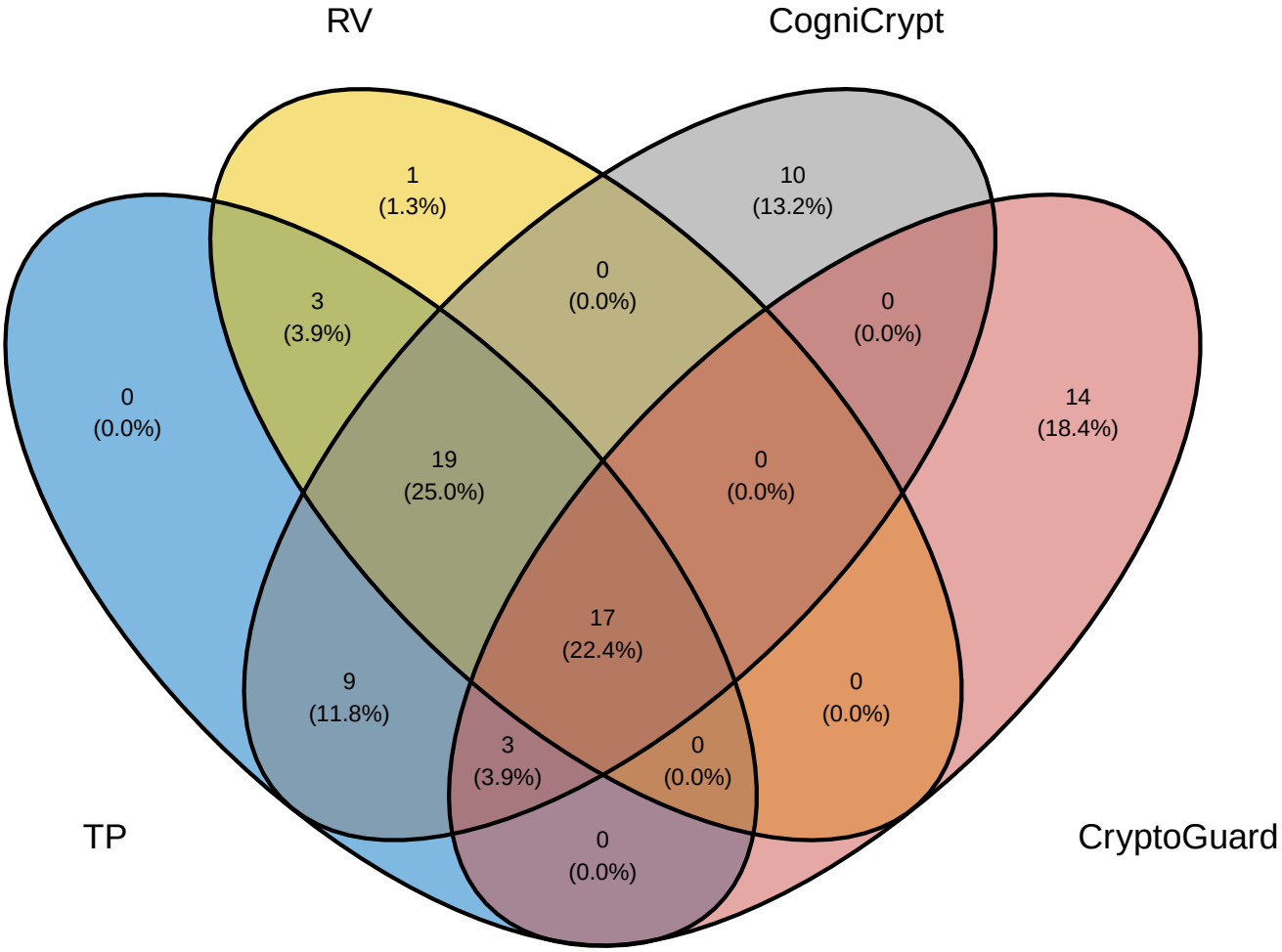


Figure 5.3: Venn Diagram summarizing the crypto API misuse each tool reports

Finding 2

RVSec and CogniCrypt report 36 true positive warnings in common in *ApacheCryptoAPIBench* (70% of the true positives in the benchmark), even though they use distinct techniques for detecting crypto API misuses.

Finding 3

Seven out 12 RVSec missed warnings (false negatives) in *ApacheCryptoAPIBench* are due the lack of test cases that would be necessary to reveal the misuse using RVSec. The remaining ones are due to specifications that we did not translate

from CrySL.

When we compare the results of our analysis with what had been presented in recent papers [39, 35], which also use *ApacheCryptoAPIBench*, we found a discrepancy with respect to the performance of CryptoGuard. This difference might be partially explained considering that:

- We removed from our analysis two projects from *ApacheCryptoAPIBench* that we were not able to build: Taverna Workbench and TomEE Container—the aforementioned research work take these projects into account during their analysis.
- We removed from our analysis warnings reporting crypto API misuses that occurr in external dependencies, since we were not able to manually validate these warnings. It is unclear in the specifications how the authors of *ApacheCryptoAPIBench* confirmed these misuses. We could not validate that using the original ground truth because several misuses there do not specify the class that originate the issue.
- CryptoGuard reports a specific misuse for every call to the `java.util.Random()` constructor, regardless the respective random instance being used in a cryptographic context or not. The authors of *ApacheCryptoAPIBench* consider all these warnings *true positives* originally. After our manual analyis, we conclude that none of these instances are being used for cryptography and then we mark these warnings as *false positives*. This leads to a higher number of false positives related to CryptoGuard.
- We build our *ApacheCryptoAPIBench* ground truth from the after a manual analysis of the warnings all tools generate. As such, our set of true positives include cases that are not present original ground truth dataset. This might also lead to a higher number of false negatives than reported before.
- Moreover, we could run CogniCrypt in all artifacts of *ApacheCryptoAPIBench*, differently from what had been originally reported in a preprint version of Zhang et al. work [39]. We contacted the authors of the paper, and they confirmed our findings for CogniCrypt and (hopfully) they have fixed part of their findings in the camera version of their paper.

5.2.5 RQ2: RV Overhead

Comparing a dynamic analysis like Runtime Verification with static methods brings forth the discussion the overhead of RV for crypto API misuse detection. Furthermore, using RV to simultaneously monitor many specifications like we do is known to be more costly than monitoring a single one specification [40, 41, 42]. Table 5.6 shows the times for

building the the *ApacheCryptoAPIBench* projects without (the "TBase (s)" column) and with (the "TRV (s)" column) RV. It also shows the RV overhead (the "Overhead (%)" column), computed as discussed in Section 5.2.2. Time units are measured in seconds. Note that we ran the buid process in each project 10 times each, with and without RV, and computed the average times TRV and TBase.

Project	TRV (s)	TBase (s)	Overhead (%)
Directory Server	21.30	15.00	42.00
ActiveMQ Artemis	39.80	35.90	10.86
ManifoldCF	23.90	22.00	8.64
DeltaSpike	47.10	39.80	18.34
Meecrowave	48.40	34.40	40.70
Spark	1319.70	1115.40	18.32
Tika	28.00	25.10	11.55
Wicket	24.00	15.30	56.86

Table 5.6: Overhead results for *ApacheCryptoAPIBench*, considering the average time of 10 executions of the test suites for each project and configuration (with and without RV).

In summary, RV overhead on these projects ranges from 8.64% (ManifoldCF) to 56.86% (Wicket), with an average overhead of 25.90% and median overhead of 18.32%. Considering Spark, which has the highest original execution time among all the projects, the overhead is 18.30%—roughly 22 minutes **with** RV and roughly 18 minutes **without** RVSec. We think that RV overhead on these projects might be acceptable. Further, we only measured RV overhead on one revision for each project because our focus is on comparing with static analysis based approaches. Still, recent evolution-aware techniques were proposed that reduce RV runtime overhead by up to $10x$ - averaging $5x$ - when RV is utilized across several stages of a project, e.g., during continuous integration or regression testing [43, 44]. Therefore, it is withing reason to think that using evolution-aware RV to detect crypto API misuses as software evolves could produce even lower runtime overheads than the ones shown in Table 5.6.

Figure 5.4 shows a correlation matrix between six metrics: number of test cases (TCs), source lines of code (SLOC), number of RVSec monitors generated (Monitors) and events captured (Events) while running the test cases, crypto API misuses (Misuses), and Overhead. Note that there is a small, negative correlation between the Overhead and the TCs (-0.09), SLOC (-0.12), and Events (-0.16) metrics. There is also a moderate positive correlation between the Overhead and the number of RVSec generated monitors (0.32). Since the number of RVSec specifications is the same across all projects, we were actually expecting that the total number of test cases (TCs), Monitors or Events could explain the RVSec overhead. Our correlation matrix suggests the contrary.

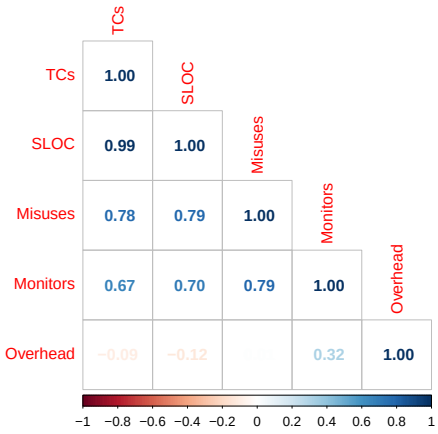


Figure 5.4: Correlation matrix between the Overhead for building the projects introduced by RV and other properties of the projects

Finding 4

There is a small correlation between the overhead RVSec introduces in the time to execute the test cases of the projects and metrics such as number of test cases, SLOC, Misuses and Events, and a small-to-moderate correlation between overhead and the number of Monitors.

5.2.6 RQ3: Correlation of Coverage with RV Accuracy

We compute four metrics related to the test suites of the *ApacheCryptoAPIBench* projects: Total Number of Test Cases (TCs), average Instruction Coverage (IC), average Branch Coverage (BC), and average Method Coverage (MC)—the last three coverage metrics using the Java Code Coverage Library (JaCoCo), which we integrated into the build process of the *ApacheCryptoAPIBench* projects. After running the test suites, JaCoCo exports test coverage measurements for each class of a project and we then compute the average coverages (i.e., instruction, branch, and method level coverage) as we present in Table 5.7.

We then explore the correlation between the test suite metrics and the F-measure metric. Our idea is to identify if there is any correlation between test coverage and accuracy. The correlation matrix in Figure 5.5 summarizes the result, showing that there is a small and negative correlation between the number of test cases and instruction coverage with the accuracy metric (F-measure). This lack of correlation might be due to the fact that crypto API misuses concentrate in a small number of classes. That is, the

Apache Module	TCs	IC	BC	MC
Directory Server	376	47.54	2.29	66.60
ActiveMQ Artemis	110	11.22	6.76	14.29
ManifoldCF	5	3.62	0.00	5.07
DeltaSpike	155	33.33	6.78	69.79
Spark	2045	8.64	1.06	17.46
Tika	222	23.42	12.34	29.74
Wicket	237	16.88	12.16	24.47

Table 5.7: Summary of the test suite metrics

JaCoCo outputs report that the test suite of all projects covers 5706 classes. Nonetheless, the crypto API misuses in all projects are present in only 20 classes (0.35% of the total classes covered in the test suites).

Finding 5

Since crypto API misuses concentrate in a small number of classes of projects, there is no correlation between test coverage and the RVSec accuracy.

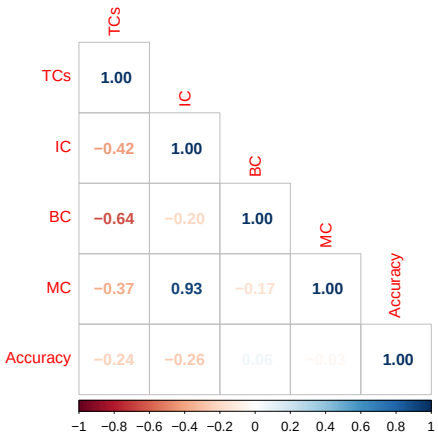


Figure 5.5: Correlation matrix between the test suite metrics and F-measure

5.3 Qualitative Analysis

We discuss the reasons for false positives and false negatives reported by RV, CogniCrypt, and CryptoGuard, based on the quantitative results obtained from our experimens. We also discuss some limitations of RV for crypto API misuse detection, and identify problems

that future work should solve for RV to become better at detecting crypto API misuses. We also describe a scenario that is common within the JCA, which our RV approach was able to handle, but that static analysers struggled with.

5.3.1 False Negatives in *MASCBench*

Figure 5.6 shows an example crypto API misuse that neither CogniCrypt nor CryptoGuard detected, but which RVSec detected. There, `Cipher` is instantiated by calling the `getAlgorithm` method of class `KeyGenerator`. In the example, `getAlgorithm` returns the `String "AES"`— since `keygen` is instantiated using a call to `KeyGenerator.getInstance("AES")`. But, instantiating the `Cipher c` in this way is similar to calling `Cipher.getInstance("AES")`, which specifies just the cipher algorithm, and not its operation mode and padding. The vulnerability occurs because the default mode and padding configuration for `AES` is `ECB/PKCS5Padding`, which might result in disclosing of sensitive information [**?**]. So, creating a `Cipher` as shown is insecure, but CogniCrypt and CryptoGuard do not detect this misuse. Differently, if one passes the `"AES"` string to the `Cipher.getInstance` method directly, instead of calling `KeyGenerator.getAlgorithm()`, both tools detect the misuse.

```
public class CipherExample09 {
  public static void main(String[] args) {
    try{
      KeyGenerator keygen = KeyGenerator.getInstance("AES");
      SecretKey key = keygen.generateKey();
      Cipher c = Cipher.getInstance(keygen.getAlgorithm()); /* error */
   /* possible patch:
    * Cipher c = Cipher.getInstance("AES/CBC/PKCS5Padding");
    */
      c.init(Cipher.ENCRYPT_MODE, key);
      c.doFinal("something".getBytes());
    }catch(Exception e ){
      e.printStackTrace();
    }
  }
}
```

Figure 5.6: Program in *MASCBench* that yields false-negatives in CogniCrypt and CryptoGuard

The remaining false-negatives in CogniCrypt are due to its inability to detect (a) usages of insecure message digest algorithms whose names are mutated in the calling program (e.g., `"md5".toUpperCase()`); and (b) initializing a `SecureRandom` class with hard-coded values (instead of random values). Together, these cases account to four and three CogniCrypt false negatives, respectively—out of eight false negatives.

```
public class CipherExample05 {
  private String cipherName = "AES/GCM/NoPadding";

  public CipherExample05 methodA() {
    cipherName = "AES/GCM/NoPadding";
    return this;
  }

  public CipherExample05 methodB() {
    cipherName = "DES";
    return this;
  }

  public String getCipherName(){
    return cipherName;
  }

  public static void main(String[] args) throws Exception {
    String name = new CipherExample05().methodA().methodB().getCipherName();
    /* error: DES is not secure */
    Cipher c = Cipher.getInstance(name);
    runCipher(c);
  }

  public static void runCipher(Cipher c) throws Exception {
    /* error: DES is not secure */
    Key key = KeyGenerator.getInstance("DES").generateKey();
    c.init(Cipher.ENCRYPT_MODE, key);
    byte[] cipherText = c.doFinal("password".getBytes());
  }
}
```

Figure 5.7: Example of CryptoGuard false negative for *MASCBench*

Five false-negatives in CryptoGuard occur in the scenarios that use multiple method calls to initialize a `Cipher` class. Figure 5.7 presents an example, where the `Cipher c` instance is initialized using insecure "DES" algorithm. Note that CryptoGuard does not report any issue with the test case of Figure 5.7. Although CogniCrypt also misses the first error in this test case (Line 21), it correctly detects the second one (Line 27). Extending CryptoGuard and CogniCrypt with a more advanced interprocedural data flow analysis might reduce these interprocedural false-negative scenarios in both tools. For instance, FlowDroid is able to detect source-sink flows using different method calls and string manipulations [45].

5.3.2 Analyzing *SmallCryptoAPIBench* results

RV's False Negatives and False Positives. Seven (out of eight) RVSec's false negatives in *SmallCryptoAPIBench* are due to the use of hard-coded passwords for loading

key stores. According to CWE-798, this is a severe threat since "*hard-coded credentials typically create a significant hole that allows an attacker to bypass the authentication that has been configured by the software administrator*" [46]. Specifically, it is very difficult to write a specification that allows RVSec to check at runtime if the string being used as a password was hard-coded at initialization or not (see Lines 14 and 16 in Figure 5.8). The recommended best-practice is to retrieve the passwords from a external and private database . Additional techniques, such as taint analysis or string analysis may need to be used together with RVSec to detect whether passwords have been hard-coded. Four out of seven RVSec's false positives are due to a NIST that requires a number of 10 000 iterations to securely use password based encryption (via `PBEParameterSpec` objects). However, the *SmallCryptoAPIBench* ground truth considers the use of *at least* 1000 iterations to be safe. So, these false positives are due to a mismatching between a recent NIST specification and choices made in curating the ground truth data set that we used.

```
public class PredictableKeyStorePassword {

  URL cacerts;
  PredictableKeyStorePassword pksp = new PredictableKeyStorePassword();

  public static void main(String args[]) throws Exception {
    pksp.go();
  }

  public void go() throws Exception {
    String type = "JKS";
    KeyStore ks = KeyStore.getInstance(type);
    cacerts = new File("input-ks").toURI().toURL();
    String defaultKey = "password";
    /* error: defaultKey is hard-coded */
    ks.load(cacerts.openStream(), defaultKey.toCharArray());
  }
}
```

Figure 5.8: Example of a false-negative from RVSec

False Positives in CogniCrypt and CryptoGuard. Eighteen CogniCrypt false positives (out of 29) and all CryptoGuard false positives happen in the *SmallCryptoAPIBench* path-sensitive programs and are due to the over-approximations both tools employ. Figure 5.10 shows an example. There, `choice` is initialized to `2`, so the condition in line 6 is always true and the secure `SHA-256` algorithm is always used on line 10, as expected. But the over-approximations that CogniCrypt and CryptoGuard employ make them flag lines 8 as insecure, and raise a warning.

This raises an interesting observation regarding a potential advantage of RV in the context of crypto API misuse: when there exists this two-step process of instantiation and

```
// g1 and g2 instatiate valid algorithms, but g3 is not
// Don't raise a warning immediately when an invalid algorithm is instantiated
String currentAlgorithmInstance = "";
event g3 after(String alg) returning(MessageDigest digest):
  call(public static MessageDigest MessageDigest.getInstance(String))
  && args(alg) && condition(!algorithms.contains(currentAlgorithmInstance.toUpperCase())) {
        currentAlgorithmInstance = alg;
  }

//It's inside the events that actually consume the algorithm that we catch UnsafeAlgorithm
event update after(MessageDigest digest):
  call(void MessageDigest.update(..)) && target(digest) {
        if (!algorithms.contains(currentAlgorithmInstance.toUpperCase())) {
                ErrorCollector.instance().addError(new ErrorDescription(ErrorType.
                    ↪ UnsafeAlgorithm, "MessageDigestSpec", "" +
        __LOC,"expecting one of {SHA-256, SHA-384, SHA-512} but found " +
            ↪ currentAlgorithmInstance + "."));
        }
  }

// Allow an invalid instatiation to occur, so long as it is followed by a valid one
ere : (g3* g1 | g3* g2) (d2 | (update+ (d1 | d2 | d3)))+
```

Figure 5.9: Pattern for reducing false positives in algorithm instantiation

then actual use of an algorithm - as exemplified in `MessageDigest`'s `getInstance(...)` → `update(...)` - static analysers might raise a warning just as it sees the instantiation of an unsafe/invalid algorithm, even if a proper instatiation takes place afterwards. Since the instantiation of the algorithm itself its not a problem; it is only when the method that actually applies the instantiated algorithm is called that a warning should be raised.

Since this was common throughout many of the JCA classes (7 out of the 22 specifications we created), we developed a pattern that allowed us to properly address this condition. Figure 5.9

False Negatives in CogniCrypt and CryptoGuard. CogniCrypt reported (a) four false negatives related to the *SmallCryptoAPIBench*'s `MessageDigest` inter-procedural examples of *SmallCryptoAPIBench* that use algorithms that are not recommended anymore (i.e., MD5 or SHA1) and (b) thirteen false negatives related to the use of predictable seeds used for generating random numbers. These are the main sources of CogniCrypt false negatives for the *SmallCryptoAPIBench*. Differently, the use of the type string in credentials and predictable keys are the main source of false negatives for CogniCrypt. The use of strings credentials is not recommended because strings are immutable values that reside in the heap until garbage collection. This amplifies that attack surface.

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class BrokenHashABPSCase1 {
  public static void main (String [] args) throws Exception {
    String name = "abcdef";
    int choice = 2;
    MessageDigest md = MessageDigest.getInstance("SHA1");
    if(choice>1)
      md = MessageDigest.getInstance("SHA-256"); @\label{ps-example-create-instance}@
    md.update(name.getBytes());
    System.out.println(md.digest());
  }
}
```

Figure 5.10: Path sensitive example that leads to false positives in both CogniCrypt and CryptoGuard

5.3.3 Comparison using *ApacheCryptoAPIBench*

In general, over-approximation by static analysis can lead to false positives, and low-quality test suites in RV can lead to false negatives. Since we do not augment the test suites in *ApacheCryptoAPIBench*, we expect that RVSec may not find as many warnings as CogniCrypt and CryptoGuard. Our manual analysis in the *ApacheCryptoAPIBench* results leads to some interesting findings. We discuss some of them here.

CryptoGuard reports a many more issues in Meecrowave than RVSec and CogniCrypt. We manually inspected these warnings and found that raw24 out of 25 warnings that CryptoGuard reports in Meecrowave are from one statement, shown in Figure 5.11. In that statement, CryptoGuard reports a warning for each byte in the byte array parameter of `SecretKeySpec`'s constructor. RVSec and CogniCrypt report just one warning for the byte array. The reason for the warning form all three tools is that the byte array argument is required to be generated using a random number generator. Failure to use a random number generator allows reverse engineering attacks.

```
private final SecretKeySpec key = new SecretKeySpec(new byte[]{
  (byte) 0x76, (byte) 0x6F, (byte) 0xBA, (byte) 0x39, (byte) 0x31,
  (byte) 0x2F, (byte) 0x0D, (byte) 0x4A, (byte) 0xA3, (byte) 0x90,
  (byte) 0x55, (byte) 0xFE, (byte) 0x55, (byte) 0x65, (byte) 0x61,
  (byte) 0x13, (byte) 0x34, (byte) 0x82, (byte) 0x12, (byte) 0x17,
  (byte) 0xAC, (byte) 0x77, (byte) 0x39, (byte) 0x19}, "DESede");
```

Figure 5.11: Code fragment for which CryptoGuard generates 24 warnings in Meecrowave

Other interesting cases appear in ActiveMQ Artemis, ManifoldCF, and Tika, for which CogniCrypt generates a higher number of warnings. In ActiveMQ Artemis, we manually confirmed that all tools found issues in the same regions of code, though CogniCrypt

generates more warnings than RVSec and CryptoGuard. An additional warning from CogniCrypt relates to the use of the class `SecretKeyFactory`, for which we do not have a corresponding RVSec specification. In ManifoldCF, CogniCrypt reports three warnings related to the `Cipher` class that RVSec does not report due to a lack of test coverage. Additionally, CogniCrypt also reports warnings related to the `SecretKeyFactory` class. Finally, a manual analysis of Tika and ManifoldCF reveals that RVSec did not find any misuse due to low test coverage. These findings about the impact of coverage on RV validate our results on coverage and RV effectiveness in Section 5.2.6.

We explored the use of the Randoop [47] tool to generate tests for these two projects. In the case of ManifoldCF, RVSec identified 14 warnings after enriching its test suite with Randoop-generated test cases. Since the vulnerable Tika code using JCA primitives are implemented in abstract Java classes, the test cases Randoop generate did not covers the vulnerable code. As such, RVSec did not report any issue in Tika, even after generating additional test cases using Randoop. These results using Randoop suggest that we can reduce false negatives in RVSec using test case generator tools.

5.4 Discussion

5.4.1 Lessons Learned

In a previous work, Owolabi et al. explored the use of runtime verification for bug-finding [48]. Their results suggest that runtime verification leads to a large number of false positives, mainly due to the low quality of available abstract specifications. Here we learned from evidence that, starting from a previously validated set of specifications in CrySL, RVSec is effective to identify crypto API misuses with a high precision—97% when considering the more realistic *ApacheCryptoAPIBench*. We also learned that whole project code coverage is not a requirement for using RVSec to detect crypto API misuses. Since the code using cryptographic primitives concentrates in a few classes, we can benefit from (automatically) generating test cases for a relatively small subset of system's components. Even without enriching the test suite, RVSec missed only 12 crypto API misuses from *ApacheCryptoAPIBench* (out of 51 misuses in total). This is a promising result, in particular when we consider that existing research suggests that dynamic analysis might generate many false negatives due to a possible lack of test cases. Our preliminary study using Randoop to generate test cases for *ApacheCryptoAPIBench* seems also promising, since we found a reduction in false negatives. In a future work, we want to explore extensions to test cases generation tools that might benefit from RVSec specifications.

The costs of our approach

As has been discussed throughout the chapter, an outstanding concern of dynamic analysis methods is the overhead that gets introduced into the process. However, other factors play into the overall cost of the construction and evaluation of our tool. Here is a summary of other factors we observed that also play into the costs of developing and executing RVSec:

- **Developing specifications is hard**. Even though we derived most of the knowledge about the expected behaviour of the JCA classes from CrYSL CrySL, the task of writing and valitaing specs if far from trivial. We had to perform extensive iteration, manual inspection and testing to ensure our JavaMOP extension worked as intened;
- **Writing AspectJ code** is also not an easy task. Not only does the language have peculiar syntax and semantics, but it also lacks static analysis tools to assist in their development. Furthermore, the error messages produced by its engine are not always easy to interpret and correct. This slowed down the development of our specs, and allowed some bugs to creep into them during the empirical assessment;
- **`rv-monitor`**'s internals presented challenges to our project. Alongside the aforementioned environment sensitivity, we had to perform tweaks in its settings as the number of events observed, and therefore monitors created, increase. We had to increase the Java Virtual Machine stack size to accommodate for this. We also had to remove one of the projects from *ApacheCryptoAPIBench* from our experiment, because the instrumented code would not finish its execution, even when left running for over ten hours.

5.4.2 Threats to Validity

Our study only considers violations for the correct usage of the JCA library. Therefore, a limitation of our work is that we did not explore whether or not RVSec would be effective in detecting crypto API misuses of non-JCA libraries. Nonetheless, previous studies have used RVSec (and in particular the JavaMOP implementation of RVSec) to find violations in APIs that go beyond crypto libraries. For this reason, we believe that the approach we detail here could also be used to check incorrect usage of crypto APIs other than JCA.

We use three distinct benchmarks in our research: *MASCBench*, *SmallCryptoAPIBench*, and *ApacheCryptoAPIBench*. The programs in the first two benchmarks have been designed with the sole goal of comparing static analysis tools that detect crypto API misuses. We reused these benchmarks almost as is, even though we had to fix several bugs that do not allow us to execute the programs—something necessary to explore the capabilities of

the RVSec approach. Although, these benchmarks are useful to understand the limits of the tools on detecting crypto API misuses, we found several fictitious cases that would hardly appear in real systems. We are still confident that *MASCBench* and *SmallCryptoAPIBench* are useful for better understand where the tools might fail, but we cannot generalize the results related to these benchmarks for real systems.

Instead, *ApacheCryptoAPIBench* contains several crypto API misuses from real systems. These misuses violate recommendations from organizations such as National Institute of Standards and Technology (NIST), German Federal Office for Information Security (BSI), and Open Web Application Security Project (OWASP). Although those violations have also originated critical CVE/CWE security warnings, we did not investigate how the developers of these systems perceive the warnings reported by RVSec, CogniCrypt, and CryptoGuard. This is a limitation of our study, but we understand that this kind of validation involves a different research that we plan to conduct in a near future.

We found some issues in the ground truth of *MASCBench* and *ApacheCryptoAPIBench*. We decided to remove some program examples for *MASCBench* and keep its ground truth definitions. Differently, we generated a new ground truth after manually checking all warnings RVSec, CogniCrypt, and CryptoGuard report for *ApacheCryptoAPIBench*. We believe that our ground truth for *ApacheCryptoAPIBench* is more reliable and allow a fair comparison between the RVSec, CogniCrypt, and CryptoGuard—which report crypto API misuses using different granularity. We have already contacted the authors of *ApacheCryptoAPIBench* explaining our concerns, and we will share all assets we develop during our research for further validation.

Chapter 6

Concluding Remarks

Our primary motivation for this research is the fact that, not only are cryptographic systems hard to engineer, but they are also hard to use properly [5], and yet they are essential for data security and integrity. In a scenario where we expect an increasing demand for software developers - coupled with increasing demands for security and privacy - there is an increased likelihood that not all engineers are going to be educated on the mathematical underpinnings and assumptions behind crypto APIs, therefore calling for the need to provide tooling that supports them to secure data correctly.

Although we are proposing an alternative method to already effective static analysers available in the literature, by no means are we proposing a replacement; instead, we are looking to combining them, such that each approach compensates for defficiencies in the other.

In this study, we leveraged the JavaMOP implementation, as well as CrySL's validated specification files to create our own specs, which we then used to check at runtime if the APIs of the Java Cryptography Architecture were being used properly. Our tool used the Maven build system to integrate with the projects' test suites, and we streamlined the process of using a JavaMOP agent to monitor target code.

The empirical assesment we conducted consisted of running CrySL, CryptoGuard, and RVSec against benchmark test sets, as well real as in real projects, whilst measuring both accuracy metrics and runtime costs. We also conducted correlation studies between test coverage and number of monitors. Our results support RV as an effective tool for crypto API misuse detection. Overall, RV displayed better accuracy than CrySL and CryptoGuard: 8–16, 6–25, and 7-1 percentage points higher precision, recall, and F-measure, respectively. The overhead we observed for the 10 open source projects we monitored is between 9.9% and 54%.

6.1 Contributions

The following is a summary of the main contributions of our work:

- Development of abstractions that build upon the JavaMOP framework that allowed us to express the expected behaviour of crypto APIs;
- Automation of the process of generating monitors and the JavaMOP agent;
- Seamless integration between Maven testing infrastructure and RVSec verification runs;
- Translations of 22 CrySL specification files into our MOP context;
- Guidelines on how to express CrySL's constructs in RVSec;
- Data that supports RVSec as a potentially useful tool for detecting misuse of JCA classes;
- Discussions with the authors of the static analysers which resulted in the review and improvement of ground truth data present in a benchmark test proposed by the tool's authors.

6.2 Limitations and Future Work

Since this was primarily an exploratory study of the implications of applying dynamic analysis towards crypto API misuse detection, as we developed our prototype, we came across a few potential useful consequences that may stem from it. Our empirical assessment also exposed some flaws and limitations of our approach. In this section we mention practical improvements that can applied to RVsec, as well as ideas that arose during its construction.

Improvement of the Current Specs: Our empirical assessment exposed some flaws in some of the 22 specs we developed. For example, our `CipherInputStream` spec had its events referring to `CipherOutputStream` class instead. This prevented us from properly capturing misuses of the former, in case they were present in the programs we tested. Morevore, a careful inspection of the pattern outlined in figure 5.9 shows that, regardless of an invalid algorithm being instatiated or not, our EREs are written in such a way that we *always* reach the event where we capture an invalid instantation, due to a null check that will always be matched in line 6 of figure 5.9, therefore increasing the path to be traversed by at least one step every time. This is a one-line correction that can be applied to all the specs in which the pattern was applied. One could also devote a more careful

examination to the problem of expressing the fact that a string has not been hardcoded in RVSec, since this was a remarkable case where static analysers showed an advantage.

Translation of the entire set of CrySL specs: We only translated about half of the specifications that CrySL provided for the JCA, therefore leaving a lot of them out. As our empirical assessment showed, there was at least one class that was present in the benchmark sets that was not present in RVSec. This caused a reduction in the accuracy metrics of our tool. If we aim for a more thorough comparison between the two approaches, it would be desirable to have a complete set of specification files in RVSec. This can be accomplished in two ways:

- Reproducing the procedures and patterns we applied for the development of the 22 RVSec specs used in this research, and performing manual translations;
- Automating the translation from CrySL to RVSec. Though more laborious, this process could yield several benefits. For instance, if we have an automatic translator, one could translate several revisions of CrySL's codebase, and make more trustworthy comparisons. Automation would also make it easier to update our specs as CrySL's ones change, or when new ones are added.

Enhancement of benchmark data: Our empirical assessment exposed flaws in the benchmark test programs that we used. There were cases where the tests did not look realistic, as well as cases where ground truth data was incorrectly labeled. Not only does this present a threat to the validity of RVsec, but it also threatens the results of any tool that is run using such benchmarks. We believe the reduction of bias during the creation of the test cases is paramount, and believe automatic generation of test cases can assist in this regard.

Application of Recent Advancements in RV: Recent evolution-aware techniques proposed by Owalabi et al. were proposed that can reduce RV runtime overhead by an average factor of 5 - and by 10 in the most favorable cases - when RV is utilized across several stages of a project, e.g., during continuous integration or regression testing [43, 44]. Therefore, using evolution-aware RV to detect crypto API misuses as software evolves could incur in even lower runtime overheads than the values shown in Table 5.6.

Real World Verification at Scale: As mentioned throughout the text, one of the phases of our empirical assessment consisted in running RVSec successfully on 8 out of 10 Apache projects we selected , so as to assess runtime overhead. Extending this to a larger scale, whilst also applying stricter procedures for measuring the overhead, as well as collecting other metrics, such as memory consumption. Although we ran RVSec 10 times for each project, we did so in a machine running several other background processes, which could create noise in the measurements. Thus, extending the number of runs to a

larger number could reduce the variance observed, and give a better grasp of the actual overhead.

Our results ultimately lead us to conclude our work with the belief that, since there are cases where static analysers detect that RV does not, and vice-versa, combining both approaches into a testing pipeline could yield a productive synergy, simultaneously creating the potential for preventing more bugs from reaching production, as well as providing developers with greater knowledge about how crypto APIs should be used.

Reference

[1] Lazar, David, Haogang Chen, Xi Wang, and Nickolai Zeldovich: *Why does cryptographic software fail? a case study and open problems.* In *Proceedings of 5th Asia-Pacific Workshop on Systems*, APSys '14, New York, NY, USA, 2014. Association for Computing Machinery, ISBN 9781450330244. https://doi.org/10.1145/2637166.2637237. 1

[2] Chatzikonstantinou, Alexia, Christoforos Ntantogian, Georgios Karopoulos, and Christos Xenakis: *Evaluation of cryptography usage in android applications.* In *Proceedings of the 9th EAI International Conference on Bio-Inspired Information and Communications Technologies (Formerly BIONETICS)*, BICT'15, page 83–90, Brussels, BEL, 2016. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ISBN 9781631901003. https://doi.org/10.4108/eai.3-12-2015.2262471. 1

[3] Egele, Manuel, David Brumley, Yanick Fratantonio, and Christopher Kruegel: *An empirical study of cryptographic misuse in android applications.* In *Proceedings of the 2013 ACM SIGSAC Conference on Computer amp; Communications Security*, CCS '13, page 73–84, New York, NY, USA, 2013. Association for Computing Machinery, ISBN 9781450324779. https://doi.org/10.1145/2508859.2516693. 1

[4] Ferguson, Niels, Bruce Schneier, and Tadayoshi Kohno: *Cryptography Engineering: Design Principles and Practical Applications.* Wiley Publishing, 2010, ISBN 0470474246. 1

[5] Schneier, Bruce: *Cryptography is harder than it looks.* IEEE Security Privacy, 14(1):87–88, 2016. 1, 8, 57

[6] Nadi, Sarah, Stefan Krüger, Mira Mezini, and Eric Bodden: *"jumping through hoops": Why do java developers struggle with cryptography apis?* In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 935–946, 2016. 1

[7] Acar, Yasemin, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L. Mazurek, and Christian Stransky: *Comparing the usability of cryptographic apis.* In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 154–171, 2017. 1

[8] Fischer, Felix, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl: *Stack overflow considered harmful? the impact of copy amp;paste on android application security.* In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 121–136, 2017. 1

[9] Kruger, S., J. Spath, K. Ali, E. Bodden, and M. Mezini: *Crysl: An extensible approach to validating the correct usage of cryptographic apis*. IEEE Transactions on Software Engineering, 47(11):2382–2400, nov 2021, ISSN 1939-3520. 1, 2, 3, 4, 5, 7, 9, 10, 20, 21, 38

[10] Rahaman, Sazzadur, Ya Xiao, Ke Tian, Fahad Shaon, Murat Kantarcioglu, and Danfeng Yao: *CHIRON: deployment-quality detection of java cryptographic vulnerabilities*. CoRR, abs/1806.06881, 2018. `http://arxiv.org/abs/1806.06881`. 1, 2, 5, 10, 11, 12, 18, 38

[11] Chen, Feng and Grigore Roşu: *Towards monitoring-oriented programming: A paradigm combining specification and implementation*. Electronic Notes in Theoretical Computer Science, 89(2):108–127, 2003, ISSN 1571-0661. `https://www.sciencedirect.com/science/article/pii/S1571066104810454`, RV '2003, Run-time Verification (Satellite Workshop of CAV '03). 2, 15, 16, 17, 18, 19

[12] Luo, Qingzhou, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian Florin Serbanuta, and Grigore Rosu: *Rv-monitor: Efficient parametric runtime verification with simultaneous properties*. In *Proceedings of the 14th International Conference on Runtime Verification (RV'14)*, volume 8734 of *LNCS*, pages 285–300. Springer, September 2014. 2, 19

[13] Jin, Dongyun, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Roşu: *Javamop: Efficient parametric runtime monitoring framework*. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1427–1430, 2012. 2, 13, 18, 19, 25

[14] Chess, B. and G. McGraw: *Static analysis for security*. IEEE Security Privacy, 2(6):76–79, 2004. 2, 5, 6

[15] Chen, Feng and Grigore Roşu: *Java-mop: A monitoring oriented programming environment for java*. In Halbwachs, Nicolas and Lenore D. Zuck (editors): *Tools and Algorithms for the Construction and Analysis of Systems*, pages 546–550, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg, ISBN 978-3-540-31980-1. 4, 5

[16] Meredith, Patrick and Grigore Roşu: *Runtime verification with the rv system*. Volume 6418, pages 136–152, January 1970. 5

[17] Marcilio, Diego, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz, and Gustavo Pinto: *Are static analysis violations really fixed? a closer look at realistic usage of sonarqube*. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 209–219, 2019. 6

[18] Nielson, Flemming, Hanne R. Nielson, and Chris Hankin: *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010, ISBN 3642084745. 6, 7, 17

[19] Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman: *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006, ISBN 0321486811. 7

[20] Rival, Xavier and Kwangkeun Yi: *Introduction to static analysis: an abstract interpretation perspective*. Mit Press, 2020. 7

[21] Weiser, Mark: *Program slicing*. IEEE Transactions on Software Engineering, SE-10(4):352–357, 1984. 10

[22] Landi, William: *Undecidability of static analysis*. ACM Lett. Program. Lang. Syst., 1(4):323–337, dec 1992, ISSN 1057-4514. https://doi.org/10.1145/161494.161501. 11

[23] Meredith, Patrick and Grigore Roşu: *Runtime verification with the rv system*. Volume 6418, pages 136–152, January 1970. 13

[24] Havelund, Klaus, Grigore Rosu, and Daniel Clancy: *Java pathexplorer: A runtime verification tool*. In *International Space Conference*, 2001. 14, 19

[25] Barnett, Mike and Wolfram Schulte: *Spying on components: A runtime verification technique*. In *Workshop on Specification and Verification of Component-Based Systems*. Citeseer, 2001. 14, 19

[26] Barnett, Mike and Wolfram Schulte: *Runtime verification of. net contracts*. Journal of Systems and Software, 65(3):199–208, 2003. 14, 19

[27] Kiczales, Gregor, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean Marc Loingtier, and John Irwin: *Aspect-oriented programming*. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997. 14, 15, 16

[28] Laddad, Ramnivas: *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, July 2003. ISBN-10: 1930110936 ISBN-13: 978-1930110939. 15

[29] Meredith, Patrick O'Neil: *Efficient, expressive, and effective runtime verification*. University of Illinois at Urbana-Champaign, 2012. 16

[30] Chen, Feng and Grigore Roşu: *Mop: An efficient and generic runtime verification framework*. SIGPLAN Not., 42(10):569–588, oct 2007, ISSN 0362-1340. https://doi.org/10.1145/1297105.1297069. 16, 17, 18, 19

[31] Meredith, Patrick O'Neil, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu: *An overview of the MOP runtime verification framework*. International Journal on Software Techniques for Technology Transfer, 14(3):249–289, June 2011. 17

[32] Fox, A. and D.A. Patterson: *Engineering Software as a Service: An Agile Approach Using Cloud Computing*. Strawberry Canyon LLC, 2013, ISBN 9780984881246. https://books.google.com.au/books?id=3kqjmwEACAAJ. 21

[33] Brooks, Frederick P.: *The Mythical Man-Month: Essays on Softw*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1978, ISBN 0201006502. 28

[34] Ami, Amit Seal, Nathan Cooper, Kaushal Kafle, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni: *Why Crypto-detectors Fail: A Systematic Evaluation of Cryptographic Misuse Detection Techniques*. pages 397–414, 2022. 35

[35] Afrose, Sharmin, Ya Xiao, Sazzadur Rahaman, Barton P. Miller, and Danfeng Yao: *Evaluation of static vulnerability detection tools with java cryptographic API benchmarks.* CoRR, abs/2112.04037, 2021. https://arxiv.org/abs/2112.04037. 35, 40, 42, 45

[36] Kahneman, D., O. Sibony, and C.R. Sunstein: *Noise: A Flaw in Human Judgment.* Little, Brown, 2021, ISBN 9780316451383. https://books.google.com.au/books?id=g2MBEAAAQBAJ. 37

[37] Kahneman, D.: *Thinking, Fast and Slow.* Farrar, Straus and Giroux, 2011, ISBN 9781429969352. https://books.google.com.au/books?id=ZuKTvERuPG8C. 37

[38] Spearman, Charles: *The proof and measurement of association between two things.* 1961. 39

[39] Zhang, Ying, Ya Xiao, Md Mahir Asef Kabir, Yao Danfeng, and Na Meng: *Example-based vulnerability detection and repair in java code.* pages to–appear, 2022. 42, 45

[40] Jin, Dongyun, Patrick O'Neil Meredith, and Grigore Roşu: *Scalable parametric runtime monitoring.* Technical report, UIUC, 2012. 45

[41] Meredith, Patrick and Grigore Roşu: *Efficient parametric runtime verification with deterministic string rewriting.* pages 70–80, 2013. 45

[42] Luo, Qingzhou, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian Florin Serbanuta, and Grigore Roşu: *Rv-monitor: Efficient parametric runtime verification with simultaneous properties.* pages 285–300, 2014. 45

[43] Legunsen, O., Y. Zhang, M. Hadzi-Tanovic, G. Roşu, and D. Marinov: *Techniques for evolution-aware runtime verification.* pages 300–311, 2019. 46, 59

[44] Legunsen, O., D. Marinov, and G. Roşu: *Evolution-aware monitoring-oriented programming.* pages 615–618, 2015. 46, 59

[45] Arzt, Steven, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel: *Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps.* In O'Boyle, Michael F. P. and Keshav Pingali (editors): *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269. ACM, 2014. https://doi.org/10.1145/2594291.2594299. 50

[46] *Use of hard-coded credentials.* Available from MITRE, CWE-ID CWE-798. https://cwe.mitre.org/data/definitions/798.html, visited on 2022-04-23. 51

[47] Pacheco, Carlos, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball: *Feedback-directed random test generation.* In *ICSE 2007, Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, Minneapolis, MN, USA, May 2007. 54

[48] Legunsen, Owolabi, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov: *How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications.* pages 602–613, 2016. 54