SpanL: Creating Algorithms for Automatic API Misuse Detection with Program Analysis Compositions

Sazzadur Rahaman[1(✉)], Miles Frantz[2], Barton Miller[3], and Danfeng (Daphne) Yao[2]

[1] University of Arizona, Tucson, USA
sazz@cs.arizona.edu
[2] Virginia Tech, Blacksburg, USA
[3] University of Wisconsin-Madison, Madison, USA

Abstract. High-level language platforms provide APIs to aid developers in easily integrating security-relevant features in their code. Prior research shows that improper use of these APIs is a major source of insecurity in various application domains. Automatic code screening holds lots of potential to enable secure coding. However, building domain-specific security analysis tools requires both application domain and program analysis expertise. Interestingly, most of the prior works in developing domain-specific security analysis tools leverage some form of data flow analysis in the core. We leverage this insight to build a specification language named SpanL[1] for domain-specific security screening. The expressiveness analysis shows that a rule requiring any composition of dataflow analysis can be modeled in our language. Our evaluation on four cryptographic API misuse problems shows that our prototype implementation of SpanL does not introduce any imprecision due to the expressiveness of the language([1]SpanL stands for Security sPecificAtioN Language.).

Keywords: Program Analysis · Specification Language · API Misuse

1 Introduction

Platform API misuses [6,7,18,21,22] are a common source of software vulnerability in high-level language platforms such as Java and Android. For example, Java cryptography libraries (such as JCA, JCE, and JSSE[1]) [6,15,22], Spring security framework [21], Android system APIs [12,13] are widely misused by the developers. There are some other non-system APIs misuses that have serious security consequences in the Android ecosystem. For example, non-system APIs [23] to access sensitive information (location, IMEI, passwords, etc.), cloud service APIs

[1] JCA, JCE, and JSSE stand for Java Cryptography Architecture, Java Cryptography Extension, and Java Secure Socket Extension, respectively.

J. Zhou et al. (Eds.): ACNS 2023 Workshops, LNCS 13907, pp. 515–529, 2023.
https://doi.org/10.1007/978-3-031-41181-6_28

for information storage [28], etc. Prior research revealed a multitude of causes behind this API misuse problem. The lack of cybersecurity training [21], insecure and misleading suggestions in StackOverflow [7,21], lack of understanding of the underlying APIs [6,22], are some of them. It is speculated that large language model-based automatic code generation tools will add fuel to the fire. Thus, development-time security screening has become a vital need.

There has been an extensive body of research on building static analysis tools for development-time security screening for Java and Android platforms [8,12,15,19,23,26]. Most of these efforts focused on either improving the detection capability [26] or detecting a new class of misuses [12]. However, the detection rules and types of vulnerabilities are immutable in most of the frameworks. This implies that extending existing frameworks for i) detecting new classes of vulnerabilities or ii) domain-specific security screening is largely hindered by the necessity of in-depth program analysis expertise. To address this challenge, Kruger et al. [19] designed a specification language (named CrySL [19]) to model cryptographic API misuses. However, CrySL's specification language is tightly coupled with its underlying detection mechanism (i.e., typestate analyses). Given a data object, typestate analyses define a valid sequence of operations that are allowed on the object. In CrySL, it is unclear how to express compound rules involving constants that are not known beforehand (i.e., return true if, all constant values to reach program point x, are not in the set of all constant values to reach program point y). In this paper, we tackle the following research question. *Is it possible to develop a universal specification language to express a wide variety of program properties (which we call, modeling for analysis) that can be used for domain-centric security and correctness checking?*

Existing work on finding API misuse vulnerabilities (e.g., improper use of cryptographic APIs [26], Android's fingerprint APIs [12], third-party cloud APIs [28], payment application APIs [20], etc.) suggests that most of the application-level API misuse problems can be modeled with dataflow-based program properties. Inspired by these works, we propose a new specification language that can be used to model any security rule that can be expressed in any arbitrary composition of various dataflow analysis algorithms. Next, we build a system named SpanL that takes rule specification written in our language and Java code as input and outputs the analysis result. Specifically, our main technical innovation is to allow compositions of basic dataflow analyses to model program properties.

The contribution of this paper is summarized as follows.

- We design a universal specification language named SpanL, to model meta-level program properties that can be detected by using a composition of various forms of data-flow analyses. We theoretically prove the expressiveness of the language.

– To further demonstrate the expressiveness, we model 4 composite security rules from [26] in SpanL language. Our experimental evaluation on CryptoApi-Bench [8] shows that SpanL's performance is similar to CryptoGuard [26].

SpanL decouples application domain expertise from program analysis expertise and enables quick adoption of data flow analyses-based security screening to any application domain.

2 Need for Domain-Specific Security Screening

There is an emerging need for scientific methods to enable domain-specific security checking. Existing work in this space mostly targets individual domains e.g., improper use of cryptographic APIs [26], Android's fingerprint APIs [12], third-party cloud APIs [28], payment application-specific APIs [20], etc. However, the approach of building a new tool to i) either improve coverage in an existing domain, or ii) enable security screening in a new domain significantly hinders its potential.

Additionally, software targeted for the payment card industry [5], IoT industry [14,16,24,27], health care industry [10], etc., needs to follow certain security guidelines dictated by a governing body [5] or law [10]. These guidelines are created to protect the stakeholders by ensuring industry-wide baseline security. For example, the payment card industry (PCI) data security standard (DSS), prescribes that *"Render all passwords unreadable during storage and transmission for all system components"* [25]. This implies that passwords should be either encrypted or hashed before transmitting or storing. One can assume that a composition of data flow analysis can be leveraged to implement this rule. However, modeling this rule in a generalized fashion is challenging. Firstly, identifying a data element to be a password is non-trivial. Secondly, APIs for communication and storage is vastly dependent on the usage of underlying frameworks. For example, communication APIs in the Spring framework [18] are significantly different than Apache Struts [2]. Database APIs for MyBatis framework [4] are significantly different than Hybernate [3]. This implies that modeling *one program-property* to implement this rule for different software or domains is infeasible. However, a specification language to model security properties for automatic screening address this issue by *i)* decoupling domain expertise from program analysis algorithms, *ii)* enabling easy framework/domain/application-specific customization.

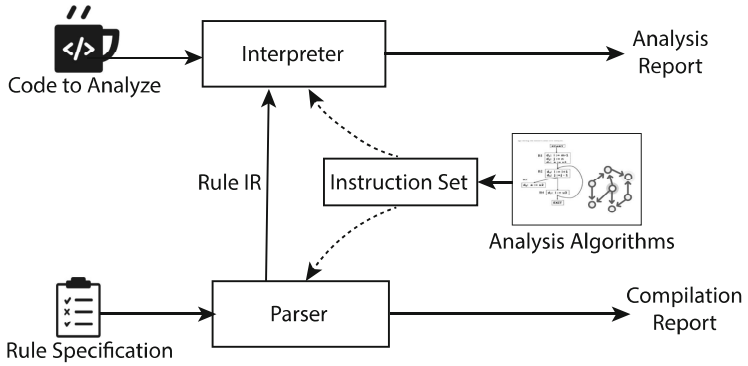

Fig. 1. System components of SpanL.

3 System Design

In this paper, we present SpanL (Fig. 1). SpanL defines a well-structured specification language to enable domain-specific security validation. Next, we provide a brief overview of our system.

3.1 Overview

The specification language in SpanL is guided by an extended Backus-Naur form (EBNF) grammar. EBNF is a collection of extensions to the Backus-Naur form (BNF) [11] to design modern compilers. We call each rule modeled in SpanL's specification language as *rule specification*. Given a rule specification, Parser parses the rule and creates a representation for execution. Then, the interpreter runs instructions from the rule specification and produces the analysis results (Interpreter in Fig. 1). The instruction set of the language is dictated by the various forms of data flow algorithms, which we discuss next.

$$\text{SpanL} ::= \textbf{APIs: } a \ \textbf{Operations: } o \ \textbf{Emits: } \epsilon \ \textbf{Constraints: } c \ \textbf{Exec: } s$$

$$v,\ a_{id},\ o_{id},\ \epsilon_{id},\ c_{id} ::= \gamma$$

$$Values \quad x ::= n \mid s$$

Type Enviornment $\Gamma ::= \ . \mid \Gamma$

γ : identifiers

n : Int

s : String

τ : Set of program-points

$\bar{\tau}$: A homogeneous array of τ or $\bar{\tau}$

Fig. 2. Overall structure of the SpanL specification language

3.2 Algorithms in SpanL

Data flow analyses enable a wide range of security applications in Java and Android platforms. Given a starting point, data flow analysis is a technique used to prove facts about a program. Given a set of definitions (starting point), forward dataflow analysis (aka reaching definition analysis) calculates if the definitions *reach* a given program point. Given a program point (starting point), backward dataflow analysis (aka liveness analysis) calculates a set of variables that are *live* at the program point. These algorithms are found to be used for cryptographic code screening [15,22,26], network [17,26], Android API misuses [13], payment application security validation, etc. Therefore, the current version of SpanL, offers security screening with inter- and intra-procedural forward and backward dataflow analysis with the support of various types of starting point definitions.

3.3 Components of SpanL Language

In Fig. 2, we present the overall structure of the SpanL specification language with various identifiers, values, and the type environment of the language. Code written in SpanL language contains 5 code sections, which we discuss next.

API Section. API section (Fig. 3) contains various API definitions, that can be referred from other sections of the SpanL code. These API definitions are reusable across various rules. Each API section has unique identifiers (a_{id}). The method signature of an API requires name and type pairs for both method arguments and the return variable, which is used to define the starting points in our dataflow analysis.

$$
\begin{aligned}
\textbf{API Group } a &::= a_{id} : a'\ a \mid a_{id} : a' \\
\textbf{API } a' &::= methodSig\ a' \mid methodSig \\
methodSig &::= ret\ className\textbf{:}\ \text{methodName}(args\textbf{)} \\
ret &::= < name : Type > \mid < \textbf{void} > \\
args &::= args,\ \ < name : Type > \ \mid\ < name : Type > \\
methodName, name &::= v \\
Type &::= basicTypes \mid basicTypes[] \mid className \mid className[] \\
className &::= className.v \mid v \\
basicTypes &::= \textbf{int} \mid \textbf{char} \mid \textbf{byte} \mid \textbf{boolean}
\end{aligned}
$$

Fig. 3. Grammar for parsing the API section

Operation Section. Operation section defines various static analysis operations that are needed to be performed in order to validate the rule. These operations are referred to from various instructions in the execution section. SpanL supports 5 types of operations (Fig. 4), i.e., inter-, intra-procedural forward and backward

dataflow analysis and iteration. An operation produces a set of program points (τ) after its execution. These operations are referred from the execution section by using their identifiers (o_{id}).

$$
\begin{aligned}
\textbf{Operation } o &::= o \mid o_{id} \ : \ o_1 \mid o_{id} \ : \ o_2 \\
o_1 &::= \mathit{inter\ byApi} \mid \mathit{inter\ byRegex} \mid \mathit{intra\ on\ byApi} \mid \mathit{intra\ on\ byRegex} \\
\mathit{inter} &::= \textbf{inter-backward} \mid \textbf{inter-forward} \\
\mathit{intra} &::= \textbf{intra-backward} \mid \textbf{intra-forward} \\
\mathit{byApi} &::= \textbf{with } a_{id} \textbf{ and } v \\
\mathit{byRegex} &::= \textbf{with } x \\
o_2 &::= \textbf{iterate } \mathit{on} \\
\mathit{on} &::= a_{id} \mid \overline{\tau}
\end{aligned}
$$

Fig. 4. Grammar for parsing the Operation section

Emit Section. Emit sets contain the guidelines for collecting information from each of the operations defined in the operation sections. SpanL supports two types of *emit sets*, i) explicit and implicit. Emit sets that are explicitly defined in the emit set section are explicit emit sets. These emit sets can be used in various constraints in the constraint section or printed in the execution section. There are two types of implicit emit sets. i) simple, and i) compound. If an operation defined in the operation section doesn't have an explicitly defined emit set, then an implicit emit set is attached to it. These are called simple emit set, which collects all the program points after the execution of the corresponding operation. Compound emit sets are created as a result of compound operations. Explicit emit sets are of three types, i.e., constants, instruction, and API invocation based emit sets (Fig. 5). The inclusion criteria for constants can be specified by types or regular expressions. All the program points containing a value of the specified type or matching the regular expression will be added to the emit set. In an instruction-based emit set, all the program points matching the regular expression are collected. Invocation-based emit sets are used to collect program points containing the specified API invocations, which are identified by a_{id}.

$$
\begin{aligned}
\textbf{Emitsets } \epsilon &::= \epsilon \mid \epsilon_{id} \ : \ \epsilon' \\
\epsilon' &::= \mathit{constants} \mid \mathit{instructions} \mid \mathit{invocations} \\
\mathit{constants} &::= \textbf{constants of-type } \mathit{Types} \mid \textbf{constants matches } x \\
\mathit{Types} &::= \mathit{Types},\ \mathit{Type} \mid \mathit{Type} \\
\mathit{instructions} &::= \textbf{instructions matches } x \\
\mathit{invocations} &::= \textbf{invocations matches } a_{id}
\end{aligned}
$$

Fig. 5. Grammar for parsing the Emit section

Constraint Section. This section defines various constraints that are used in the conditional instructions in the execution section. These constraints are defined by using emit sets, constant values, and API references. A single constraint in SpanL involves one emit set. Constraints support two sets of conditioning on emit sets, i.e., *in* and *empty* (Fig. 6). *in* can be used to check whether a set of values exists in the emit set or not. *empty* constraints can be used to check whether an emit set is empty or not.

$$
\begin{aligned}
\textbf{Constraints}\ c &::= c \mid c_{id}\ :\ c' \\
c' &::= in \mid empty \\
in &::= \{vals\}\ \textbf{in}\ \{\epsilon_{id}\} \mid \{vals\}\ \textbf{not in}\ \{\epsilon_{id}\} \\
empty &::= \{\epsilon_{id}\}\ \textbf{empty} \mid \{\epsilon_{id}\}\ \textbf{not empty} \\
vals &::= vals,\ x \mid x
\end{aligned}
$$

Fig. 6. Grammar for parsing the Constraint section

Execution Section. Execution section contains specific instructions required to be executed to validate a rule. SpanL supports three types of instructions, i.e., operation, conditional and print instructions. Operation instructions are used to execute the operations defined in the operation section. Additionally, it supports some compound operations in the form of set union, subtraction, and join based on the results of other operations. The *join* of two operations indicates that the trailing operation is executed by using the output of the leading operation as its starting point. Conditional and print instructions are used to support conditional branching and print various types of messages to the standard output, respectively.

$$
\begin{aligned}
\textit{Stmt}\quad & s ::= v := o \mid o \mid \textbf{print}(x) \mid \textbf{if}\ \delta\ \textbf{then}\ s_1\ \textit{else}\ s_2 \mid \textbf{for}\ v\ \textbf{do}\ s \mid s \mid \\
& \qquad v := \text{array}(i) \mid v' := v.get(i) \mid v.set(i,\ o) \\
\textit{Operation}\quad & o ::= (o) \mid o_1\ +\ o_2 \mid o_1\ -\ o_2 \mid o_1\ \oplus\ o_2 \mid v \mid o_{id} \\
\textit{Conditional Expression}\quad & \delta ::= \delta_1\ \textbf{and}\ \delta_2 \mid \delta_1\ \textbf{or}\ \delta_2 \mid (\delta) \mid \textbf{not}\ \delta \mid c_{id}
\end{aligned}
$$

Fig. 7. Grammar for parsing the execution section

Figure 7 shows that SpanL's execution supports various types of instructions including, *if*, *for*, *add*, *sub*, *join*, *assign* and *print*. Here, the type of *array(i)* is $\bar{\tau}$. $\bar{\tau}$ has two built-in functions $get(i)$ and $set(i, o)$. As the name implies *get* is used to access an element of the array and the *set* is used to set an element in the array. Assign statements offer the functionality of assigning the output of an operation (a set of program points) to a variable. In addition to the basic operations (invoked by o_{id}), SpanL also supports some compound instructions i.e., set addition and subtraction and join operations. The join of $o_1 \oplus o_2$ indicates

the execution o_1 by using the output program points of o_2 as the starting point. We present the judgments of the type system for the execution section in Fig. 8.

$$\frac{\Gamma \vdash \delta : \text{boolean} \quad \Gamma \vdash s_1 : \tau \quad \Gamma \vdash s_2 : \tau}{\Gamma \vdash (\textbf{if } \delta \textbf{ then } s_1 \textbf{ else } s_2\ ) : \tau}\text{IF-OP} \quad \frac{\Gamma \vdash v : \tau \quad \Gamma \vdash s : \tau}{\Gamma \vdash (\textbf{for } v \textbf{ do } s\ ) : \tau}\text{FOR-OP}$$

$$\frac{\Gamma \vdash o_1 : \tau \quad \Gamma \vdash o_2 : \tau}{\Gamma \vdash (o_1\ +\ o_2\ ) : \tau}\text{ADD-OP} \quad \frac{\Gamma \vdash o_1 : \tau \quad \Gamma \vdash o_2 : \tau}{\Gamma \vdash (o_1\ -\ o_2\ ) : \tau}\text{SUB-OP}$$

$$\frac{\Gamma \vdash o_2 : \tau \quad \Gamma \vdash o_1 : \tau}{\Gamma \vdash (o_1 \oplus o_2\ ) : \tau}\text{JOIN-OP} \quad \frac{\Gamma \vdash x : \text{Int}, \text{String}}{\Gamma \vdash (\text{print}(x)\ ) : \text{boolean}}\text{PRINT-OP}$$

$$\frac{\Gamma \vdash o : \tau}{\Gamma \vdash (v := o\ ) : \tau}\text{ASSIGN-OP} \quad \frac{\Gamma \vdash i : \text{Int}}{\Gamma \vdash (v := \text{array}(i)\ ) : \bar{\tau}}\text{ARRAY-CREATION-OP}$$

$$\frac{\Gamma \vdash v : \bar{\tau} \quad \Gamma \vdash i : \text{Int}}{\Gamma \vdash (v' := v.get(i)) : \tau}\text{GET-OP} \quad \frac{\Gamma \vdash v : \bar{\tau} \quad \Gamma \vdash i : \text{Int} \quad \Gamma \vdash o : \tau}{\Gamma \vdash (v.set(i, o)) : \text{boolean}}\text{SET-OP}$$

Fig. 8. Type judgements for execution section.

4 Expressiveness of the Language

In this section, we analyze the expressiveness of SpanL specification language. First, we discuss expressiveness theoretically and then we show several case studies by modeling various security rules.

4.1 Expressiveness Analysis

Corollary 1. *If the allowed analysis mechanisms are of forward (f) and backward (b) dataflow analysis, then from a program point* $p : x = \$r.func(a_1, a_2, \cdots, a_k)$*, one can start* $2k+3$ *numbers of analysis, including* $k+1$ *backward data flow analysis by using* $\$r$ *and* $a_i\ \forall i \in [1, k]$ *as starting criteria,* $k + 2$ *forward data flow analysis by using* $x, \$r$ *and* $a_i\ \forall i \in [1, k]$ *as starting criteria. So if the number of possible analyses can be run from a program point* p *is* $O = \{o_i\}, \forall i \in [1, |O|]$*, then* $|O| \leq 2k + 3$*.*

Definition 1. ***Super-analysis set.*** *If* P_{o_i} *represents the set of all the reachable program points after running an analysis* o_i *from a program point* p*, then* P_{o_i} *can be expressed as* $p \overset{o_i}{\rightsquigarrow} P_{o_i}$*. Let* O *is the set of a maximum number of data flow analysis that can be run from* p*. Running* O *analysis from* p *can be expressed as* $p \overset{O}{\rightsquigarrow} P_O$*, where* $P_O = \cup\{P_{o_i}\}, \forall i \in [1, |O|]$ *and* $|O| \leq 2k + 3$*. Here, we define* P_O *as the super-analysis set for* p*. This means,* P_O *can not be further extended by running new analyses from* p*.*

Definition 2. ***Super-analysis join sets.*** *If ρ_i is the set of all program points returned after starting the super-analysis from program point p_i and then recursively continue super-analysis by using the results and starting criteria until no new program points are reached. Then ρ_i can be expressed as follows.*

$$\rho_i = p_i \overset{O_1}{\rightsquigarrow} \cdots \{P_o\}_j \overset{O_j}{\rightsquigarrow} \cdots \{P_o\}_n$$

Definition 3. ***Mutually exclusive super analysis join sets.*** *Let the super-analysis sets, starting from program point $1, 2, \cdots, n$ are ρ_1, ρ_2, $\cdots$, ρ_n. We call two super-analysis sets ρ_i, ρ_j mutually exclusive if $\rho_i \subsetneq \rho_j, \forall i, j \in [1, n]$.*

If ρ_i and ρ_j are mutually exclusive super-analysis join sets, then combining both of them would increase more coverage of the analysis.

Definition 4. ***Simple rule.*** *A rule is simple if the expressiveness of its security property is bounded by a super-analysis join set.*

Definition 5. ***Compound rule.*** *A rule is compound if it requires at least two mutually exclusive super analysis join sets to model the security property of the rule.*

Theorem 1. *The expressiveness of a security property that can be detected by using a composition of various data flow analyses, is bounded by the compound rule with all mutually exclusive super analysis join sets.*

Proof. A compound rule that would need all the *mutually exclusive super-analysis join sets* combinely represents the set of all possible compositions of dataflow analysis that can be run on a program. That means if a security property can be expressed by a composition of data flow analysis, it must be bounded by the compound rule with all mutually exclusive super analysis join sets.

SpanL supports joining (JOIN-OP), and combining multiple analysis sets (ADD-OP, SUB-OP). This means SpanL language contains all the properties of expressing a compound rule. Thus, SpanL can be used to define all possible combinations of data flow analysis possible on a given program.

Listing 1.1. Rule to detect insecure RSA keys

```
APIs:
kpg_apis:
 <kpg: KeyPair> KeyPairGenerator: getInstance(<algo: String>)
 <kpg: KeyPair> KeyPairGenerator: getInstance(<p: Provider>,
                                              <algo: String>)

kpg_init:
  void KeyPairGenerator: initialize(<size: int>)

Operations:
```

```
    o1: inter-backward with kpg_apis and algo
    o2: intra-forward with kpg_apis and kpg
    o3: inter-backward with kpg_init and size

Emits:
  {kpg}: *
  {algo}: constant of-type java.lang.String
  {size}: constant of-type int, java.lang.Integer

Constraints:
   c1: kpg_init in {kpg}
   c2: {"RSA"} in {algo}
   c3: {2048, 4096} not in {size}

Exec:
   o1, o1 ⊕ o2
   if c2 and (not c1):
       print ("Must invoke initialize for RSA")
   if c1:
        o3 ⊕ o2
        if c2 and c3:
           print ("Key size must be {2048, 4096}")
```

4.2 Case Studies

In this section, we present 4 case studies of cryptographic API misuse in SpanL language, i.e., i) insecure use of RSA, ii) insecure hostname verifier, iii) insecure symmetric crypto, and iv) insecure symmetric keys usage. These case studies are commonly found in existing literature [22,26].

In Listing 1.1, we show the code written in SpanL language to detect insecure RSA key uses. Note that, this is one of the most complex security rules that involves multiple rounds of analysis. The default use of the RSA key pair generator uses 1024 bit key size. Thus, one requires to invoke the `initialize` method with proper key size i.e., 2048, 4096 to generate secure RSA keys. Here, we first define two API groups for `getInstance` and `initialize` APIs. Next, we define the set of operations to be performed with these APIs. $o1$ is used to determine if the key pair is of RSA. Then $o2$ is used to find if `initialize` method was invoked on the generated key pair instance. Finally, $o3$ is used to find the value of the key size parameter of the `initialize` method invocation. The emit set defines what values should be collected in each operation. The left-hand side indicates the set and the right-hand side contains the criteria. For example, $\{kpg\} : *$ indicates to store all the program points reached after running $o2$. Next, the constraints and Execution section defines the corresponding constraints and the sequence of instructions to run, respectively. In the execution section, $o1, o2$ join with $o1$ indicates running operation $o2$ by using the outputs of $o1$ as starting points.

The code for other case studies is presented in the appendix. Note that, while modeling these rules we followed the mapping between the program property and the rule outlined in CryptoGuard [26].

5 Experimental Evaluation

In this section, we present the evaluation results of our SpanL system.

Implementation. We implemented SpanL in Java. We used ANTLR [1] for parsing and validating the rule specification code written in SpanL language. Our current implementation supports both backward and forward intra-procedural data flow analysis. To support inter-procedural backward data flow analysis, we leveraged the version implemented in CryptoGuard [26], which is path insensitive. Our current prototype does not support inter-procedural forward data flow analysis, which can be easily added in the future.

Experiment Design for Evaluating Expressiveness. There can be two sources of imprecision in SpanL, i) imprecision due to imprecise modeling of a security rule, ii) imprecision induced by the underlying analysis algorithm. Since the main contribution of SpanL is the specification language, thus we only focus on the expressiveness in our evaluation. Specifically, we ask the following research question: *Is there any imprecision due to modeling a security property in* SpanL*?*

To answer this question, we designed the following experiment. We modeled 4 cryptographic API misuse rules (Sect. 4.2) from CryptoGuard in SpanL. Since our implementation reuses CryptoGuard's algorithms, any deviation in our result from CryptoGuard would indicate imprecision in the model. To test this, we run CryptoGuard and SpanL in CryptoApi-Bench [9]. CryptoApi-Bench is a benchmark containing various cryptographic API misuse vulnerabilities.

Table 1. Expressiveness evaluation results on CryptoApi-Bench. It indicates that SpanL does not introduce new imprecision than its underlying algorithms (which were reused from CryptoGuard).

Detection Goals	CryptoGuard			SpanL		
	TP	FP	FN	TP	FP	FN
Insecure RSA keys	4	1	1	4	1	1
Insecure hostname verifier	1	0	0	1	0	0
Insecure symmetric crypto	30	5	0	30	5	0
Insecure symmetric keys	5	1	2	5	1	2

Experimental Result. In CryptoApi-Bench, there is a total of 6 cases (5 true positives (TP) and 1 true negative cases (TN)) of insecure RSA key usage.

It has 1 instance of insecure hostname verification, 36 instances (30 TP, 6 TN) of insecure symmetric ciphers, and 9 (7 TP, 2 TN). In Table 1, we show the evaluation results of both SpanL and CryptoGuard. It shows that modeling the security rules in SpanL did not incur new imprecision, indicating the expressiveness

6 Conclusion

Enabling domain-specific security screening is important to ensure baseline security in various application domains. Prior research [13,17,18,25] showed that various domain-specific security rules can be modeled as API-centric data flow problems. In this paper, we designed a specification language to model such problems that can be automatically checked. Then we built a system named SpanL, to run codes written in our specification language for automatic code screening. The expressiveness analysis shows that a rule requiring any composition of dataflow analysis can be modeled in our language. Our evaluation on four cryptographic API misuse problems shows that our prototype implementation of SpanL does not introduce any imprecision due to the expressiveness of the language.

Acknowledgements. This work has been supported in part by the National Science Foundation under Grant No. CNS-1929701.

Appendix

Listing 1.2. Rule to detect insecure hostname verifier

```
APIs:
host_name_apis:
 boolean HostnameVerifier: verify(<name: String>,
                                  <session: SSLSession>)

Operations:
    o1: intra-backward host_name_apis with "return"

Emits:
  {o1_out}: *

CONSTRAINTS:
   c1: "@parameter1: javax.net.ssl.SSLSession" not in {o1_out}

Exec:
   o1
   if c1:
       print("verify method is not properly implemented!")
```

Listing 1.3. Rule to detect insecure symmetric ciphers

```
APIs:
crypto_apis:
  Cipher Cipher: getInstance(<name: String>)
  Cipher Cipher: getInstance(<name: String>, <p: Provider>)

Operations:
    o1:  inter-backward with crypto_apis  and name

Emits:
{name}: constants of-type java.lang.String

Constraints:
   c1: {"AES/ECB", "DES", "RC4", "IDEA"} in {name}

Exec:
o1
if c1:
    print("Found broken crypto instances")
```

Listing 1.4. Rule to detect insecure symmetric keys

```
APIs:
sk_apis:
  void SecretKeySpec: SecretKeySpec(<keyBytes: byte[]>)
  void SecretKeySpec: SecretKeySpec(<keyBytes: byte[]>,
                                          <p: Provider>)


Operations:
    o1: inter-backward with sk_apis and keyBytes

Emits:
  {keyBytes}: constant of-type java.lang.String, byte[]

Constraints:
   c1: {keyBytes} not empty

Exec:
   o1
   if c1:
       print("Keys must not be derived from constants")
```

References

1. Antlr: Quick start. https://www.antlr.org/. Accessed 20 Feb 2023
2. Apache struts. https://struts.apache.org/. Accessed 20 Feb 2023
3. Hybernate: Everything data. https://hibernate.org/. Accessed 20 Feb 2023
4. Mybatis 3: Introduction. https://mybatis.org/mybatis-3/. Accessed 20 Feb 2023
5. Payment Card Industry (PCI) Data Security Standard: Requirements and security assessment procedures. https://www.pcisecuritystandards.org/documents/PCI_DSS_v3-2-1.pdf (2018) (2018)
6. Acar, Y., et al.: Comparing the usability of cryptographic APIs. In: IEEE S&P 2017, pp. 154–171 (2017)
7. Acar, Y., Backes, M., Fahl, S., Kim, D., Mazurek, M.L., Stransky, C.: You get where you're looking for: the impact of information sources on code security. In: IEEE S&P 2016, pp. 289–305 (2016)
8. Afrose, S., Rahaman, S., Yao, D.: CryptoAPI-Bench: a comprehensive benchmark on java cryptographic API misuses. In: 2019 IEEE Cybersecurity Development, SecDev 2019, Tysons Corner, VA, USA, 23–25 September 2019, pp. 49–61 (2019)
9. Afrose, S., Xiao, Y., Rahaman, S., Miller, B.P., Yao, D.: Evaluation of static vulnerability detection tools with java cryptographic API benchmarks. IEEE Trans. Softw. Eng. **49**(2), 485–497 (2023)
10. Annas, G.J.: HIPAA regulations: a new era of medical-record privacy? N. Engl. J. Med. **348**, 1486 (2003)
11. Backus, J.W.: The syntax and semantics of the proposed international algebraic language of the zurich ACM-GAMM conference. In: Information Processing, Proceedings of the 1st International Conference on Information Processing, UNESCO, Paris 15–20 June 1959, pp. 125–131 (1959)
12. Bianchi, A., et al.: Broken fingers: on the usage of the fingerprint API in android. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, 18–21 February 2018 (2018)
13. Bosu, A., Liu, F., Yao, D.D., Wang, G.: Collusive data leak and more: large-scale threat analysis of inter-app communications. In: AsiaCCS 2017, pp. 71–85 (2017)
14. Department of Justice: Securing your "internet of things" devices (2017). https://www.justice.gov/criminal-ccips/page/file/984001/download
15. Egele, M., Brumley, D., Fratantonio, Y., Kruegel, C.: An empirical study of cryptographic misuse in Android applications. In: ACM CCS 2013, pp. 73–84 (2013)
16. European Union Agency for Network and Information Security: Baseline security recommendations for IoT in the context of critical information infrastructures (2017). https://www.enisa.europa.eu/publications/baseline-security-recommendations-for-iot/@@download/fullReport
17. Fahl, S., Harbach, M., Muders, T., Smith, M., Baumgärtner, L., Freisleben, B.: Why eve and mallory love android: an analysis of android SSL (in) security. In: ACM CCS 2012, pp. 50–61 (2012)
18. Islam, M., Rahaman, S., Meng, N., Hassanshahi, B., Krishnan, P., Yao, D.D.: Coding practices and recommendations of spring security for enterprise applications. In: 2020 IEEE Cybersecurity Development, SecDev 2020 (2020). (to appear)
19. Krüger, S., Späth, J., Ali, K., Bodden, E., Mezini, M.: CrySL: an extensible approach to validating the correct usage of cryptographic APIs. In: 32nd European Conference on Object-Oriented Programming, ECOOP 2018, 16–21 July 2018, Amsterdam, The Netherlands, pp. 10:1–10:27 (2018)

20. Mahmud, S.Y., Acharya, A., Andow, B., Enck, W., Reaves, B.: Cardpliance: PCI DSS compliance of android applications. In: Capkun, S., Roesner, F. (eds.) 29th USENIX Security Symposium, USENIX Security 2020, 12–14 August 2020, pp. 1517–1533. USENIX Association (2020)
21. Meng, N., Nagy, S., Yao, D., Zhuang, W., Argoty, G.A.: Secure coding practices in java: challenges and vulnerabilities. In: ACM ICSE 2018. Gothenburg, Sweden (2018)
22. Nadi, S., Krüger, S., Mezini, M., Bodden, E.: Jumping through hoops: why do java developers struggle with cryptography APIs? In: ICSE 2016, pp. 935–946 (2016)
23. Nan, Y., Yang, Z., Wang, X., Zhang, Y., Zhu, D., Yang, M.: Finding clues for your secrets: semantics-driven, learning-based privacy discovery in mobile apps. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, 18–21 February 2018 (2018)
24. National Institute of Standards and Technology: IoT device cybersecurity guidance for the federal government: Establishing IoT device cybersecurity requirements (2021). https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-213.pdf
25. Rahaman, S., Wang, G., Yao, D.D.: Security certification in payment card industry: testbeds, measurements, and recommendations. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, 11–15 November 2019, pp. 481–498. ACM (2019)
26. Rahaman, S., et al.: Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, 11–15 November 2019, pp. 2455–2472 (2019)
27. US Chamber of Commerce: The IoT revolution and our digital security: principles for IoT security (2017). https://scglegal.com/wp-content/uploads/2018/02/2017-Denver-TR-1550-PP-The.IoT_.Revolution..Our_.Digital.Security.Final-002-WILEY-REIN.pdf
28. Zuo, C., Lin, Z., Zhang, Y.: Why does your data leak? Uncovering the data leakage in cloud from mobile apps. In: IEEE S&P 2016 (2019)