Synthesis of Java Deserialisation Filters from Examples

Kostyantyn Vorobyov
Oracle Labs, Brisbane, Australia
kostyantyn.x.vorobyov@oracle.com

François Gauthier
Oracle Labs, Brisbane, Australia
francois.gauthier@oracle.com

Sora Bae
Oracle Labs, Brisbane, Australia
sora.bae@oracle.com

Padmanabhan Krishnan
Oracle Labs, Brisbane, Australia
paddy.krishnan@oracle.com

Rebecca O'Donoghue
Oracle Labs, Brisbane, Australia
rebecca.o.donoghue@oracle.com

***Abstract*—Java natively supports serialisation and deserialisation, features that are necessary to enable distributed systems to exchange Java objects. Deserialisation of data from malicious sources can lead to security exploits including remote code execution because by default Java does not validate deserialised data. In the absence of validation, a carefully crafted payload can trigger arbitrary functionality. The state-of-the-art general mitigation strategy for deserialisation exploits in Java is deserialisation filtering that validates the contents of an object input stream before the object is deserialised using user-provided filters.**

In this paper we describe a novel technique called *ds-prefix* for automatic synthesis of deserialisation filters (as regular expressions) from examples. We focus on synthesis of allowlists (permitted behaviours) as they provide a better level of security. *ds-prefix* is based on deserialisation heuristics and specifically targets synthesis of deserialisation allowlists. We evaluate our approach by executing *ds-prefix* on popular open-source systems and show that *ds-prefix* can produce filters preventing real CVEs using a small number of training examples. We also compare our approach with other synthesis tools which demonstrates that *ds-prefix* outperforms existing tools and achieves better F1-score.

***Index Terms*—Security, Synthesis, Regular expressions, Deserialisation filtering**

I. Introduction

Serialisation is a mechanism that converts an in-memory object into a stream that can be saved into a file or transmitted over a network. Deserialisation is a reverse process that reconstructs an object from a saved state. These features are necessary to enable cooperative distributed systems. Java[1] natively supports serialisation and deserialisation. One of the drawbacks of Java deserialisation is that by default it does not validate deserialised data. Deserialisation of data coming from malicious sources can thus lead to security exploits including remote code execution. This is because in the absence of validation, a carefully crafted payload can trigger arbitrary functionality. This type of vulnerability, known as *deserialisation of untrusted data*[2] is widespread with over 600 CVEs reported in the last 5 years.

Notably, deserialisation of untrusted data in Java goes beyond native Java deserialisation. One such example is Jackson-databind [1], a library for serialisation to and deserialisation from JSON. Jackson-databind is a core component of many popular Java frameworks (including Spring [2]) and is the 10th most popular package on Maven central at the time of writing[3]. Since 2017, 60 CVEs related to deserialisation of untrusted data in Jackson-databind were reported[4].

The state-of-the-art mitigation strategy for deserialisation exploits in Java is *deserialisation filtering* that validates the contents of an object input stream before the object is deserialised [3]–[5], classifies deserialisation targets at runtime as either safe or unsafe, and blocks deserialisation of unsafe classes. Such filters are usually based on the names of the deserialised classes and use either a blocklist or an allowlist approach. A blocklist specifies unsafe classes that should be prevented from being deserialised. An allowlist describes classes that are permitted to load with the rest of the classes treated as unsafe.

Manual construction and maintenance of deserialisation filters is tedious and error-prone. This is especially the case for large and complex applications, where deserialisation operations might be scattered across multiple components and require extensive knowledge of the system. Construction of these filters is thus best delegated to an automated approach that can synthesise a blocklist or an allowlist from positive and negative examples (i.e., class names that can be safely deserialised or potentially lead to security exploits respectively). The resulting filter should accept all positive examples, reject all negative examples and most importantly generalise to unknown examples. This is because most typically the input set of examples does not comprise the complete list of deserialisation targets. A synthesised filter, thus needs to allow deserialisation of previously unseen benign classes (as otherwise it may break the application), and block malicious targets that may represent new attack vectors.

[1]Java is a registered trademark of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

[2]https://cwe.mitre.org/data/definitions/502.html

[3]https://mvnrepository.com/popular

[4]https://www.cvedetails.com/vulnerability-list/vendorid-15866/productid-42991/Fasterxml-Jackson-databind.html

Deserialisation filters are typically implemented as regular expressions that identify benign and potentially malicious classes. Synthesis of regular expressions from examples is a well studied topic. Classic automata-theoretic approaches [6]–[11] use finite automata to represent input examples and produce a regular expression using a series of transformations that generalise the initial FA to unseen examples. Newer synthesis techniques use genetic programming [12]–[14] and sequence alignment [15].

Existing tools are not well-suited for synthesis of deserialisation filters for a number of reasons. Firstly, synthesisers are typically concerned with finding a regular expression that fits all examples and do not utilise the specific nature of the task at hand. The generalisations constructed by the state-of-the-art tools are often either too specific and prone to blocking safe classes, or too general allowing malicious classes to be deserialised. Secondly, many of the synthesis algorithms often take too long to execute. This is especially the case with automata-theoretic techniques that require costly conversion from automata to regular expressions. This makes it difficult to adopt such synthesis in the context of runtime security analysis, where a filter might be re-generated on the fly if new examples are encountered. Finally, automatic synthesis typically produces regular expressions that are difficult to understand, check or extend manually.

In this paper we describe a novel technique for automatic synthesis of deserialisation filters. We focus on synthesis of allowlists as they provide a better level of security. We use regular expressions and address the limitations of the state-of-the-art synthesisers. The key research questions answered by our synthesis approach called *ds-prefix* are as follows:

1) Does *ds-prefix* generate accurate allowlists?
2) Is the F1-score better than existing techniques?
3) Are the results of *ds-prefix* manually auditable?

Ds-prefix is based on the observation that deserialisation filters often reason at the level of packages, rather than individual classes, by allowing or blocking deserialisation of classes with given prefixes. This observation is supported by the design of Java Enhancement Proposal (JEP) 290 [3], the built-in Java deserialisation filtering mechanism, that allows to specify deserialisation filters using patterns that describe packages or sub-packages the targets belong to. The main idea behind *ds-prefix* is to find shortest positive prefixes that describe benign examples only and combine them in a regular expression. The resulting allowlist is generated by traversal of a tree-shaped finite automata with labelled states constructed from the input set of examples. Notably, *ds-prefix* generates a regular expression during traversal, surpassing costly conversion from a finite automaton to a regular expression. By using a heuristic approach, *ds-prefix* enables one to tailor synthesis to deserialisation and effectively prevent deserialisation exploits for complex systems that need a large number of input examples.

The rest of this paper is organised as follows. Section II introduces formal details required to describe our approach and Section III presents details of *ds-prefix*. Section IV describes a proof-of-concept implementation and presents the results of the empirical evaluation. Section V gives an overview of related work in the area of synthesis of regular expressions from examples and deserialisation filtering. Finally, Section VI offers concluding remarks.

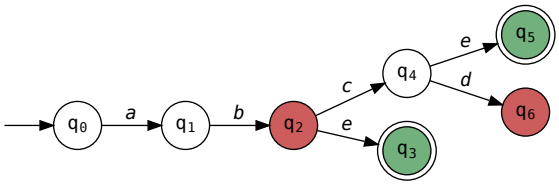

Fig. 1. An APTA with $S_+ = \{abe,\ abce\}$ and $S_- = \{ab,\ abcd\}$

II. Preliminaries

An augmented prefix tree acceptor (APTA) is a tree-shaped deterministic finite automaton that uses two kinds of state labels: *accept* and *reject* to distinguish between positive and negative examples respectively.

Definition 1 presents the formal description of APTA [6].

Definition 1 (APTA): An APTA is a tuple $\{Q, \Sigma, \delta, q_0, F_+, F_-\}$, where Q is a finite set of states, Σ is a finite input alphabet, $\delta : Q \times \Sigma \to Q$ is a transition function (may be partial), $q_0 \in Q$ is the start state, $F_+ \subseteq Q$ is a set of final accepting states, $F_- \subseteq Q$ is a set of final rejecting states such that the state transition system induced by δ is a tree.

An APTA is consistent with a set of positive examples (say $S_+ \subseteq \mathcal{P}(\Sigma^*)$) and negative examples (say $S_- \subseteq \mathcal{P}(\Sigma^*)$) iff runs of the automaton starting at q_0 lead to an element of F_+ for every string in S_+ and to an element of F_- for every string in S_-. For the sake of simplicity we assume that all the APTAs are consistent unless stated otherwise.

An example APTA is shown in Figure 1, where red and green states denote accept and reject labels respectively.

We now define a few auxiliary functions to simplify the presentation of the synthesis algorithm. The function *children* : $Q \to \mathcal{P}(Q)$ returns the set of outgoing states of the input state. The function *parent* : $Q \to Q$ returns the parent state of the input state. The function *value* : $Q \to \Sigma^*$ identifies the string that will lead the automaton from the initial state to the input state. The function *symbol* : $Q \to \Sigma^*$ identifies the string that will lead the automaton from the parent state of the input state to the input state. For instance, for the APTA in Figure 1, *children*(q_4) is $\{q_5, q_6\}$, *parent*(q_4) is q_2, *symbol*(q_4) is c and *value*(q_4) is abc.

Rather than use automata, we use regular expressions to represent the permitted behaviour. Recall that a regular expression $\mathcal{R}$ is a string over the same alphabet as an automaton extended with metacharacters [16]. The semantics of a regular expression is a set of strings over the underlying alphabet. Given regular expressions $\mathcal{R}_1$ and $\mathcal{R}_2$, $\mathcal{R}_1\mathcal{R}_2$ denotes concatenation, $\mathcal{R}_1 + \mathcal{R}_2$ represents union, $()$ is used to show grouping and * denotes Kleene closure.

We assume that function *join* accepts a set of regular expressions and returns a regular expression that is the union of all regular expressions in the input set. For instance, *join*($\{ab^*$, $cd\}$) is $ab^* + cd$.

Given an APTA A, we define APTA(q,A) to be the subtree of A rooted at state q. That is the resulting APTA has q as its initial state. For notational convenience, we let $F_+(q)$ and $F_-(q)$ indicate the sets of labelled accepting and rejecting states. When the context is clear we drop A and refer to only APTA(q).

Definition 2 (Positive, Negative and Conflicting Strings): Let $S = S_+ \cup S_-$ a labelled set of strings, A be an APTA = $\{Q, \Sigma, \delta, q_0, F_+, F_-\}$. Let $q \in Q$ such that *children*$(q) \neq \emptyset$ and *value*(q) be s. We say s is

- always positive if $F_-(q) = \emptyset$,
- always negative if $F_+(q) = \emptyset$,
- conflicting if $F_+(q) \neq \emptyset \wedge F_-(q) \neq \emptyset$

That is, if an internal state q is reachable via string s, then s is said to be positive if the subtree rooted at q contains no negative labels, negative if the subtree has no positive labels and conflicting if the subtree contains both negative and positive labels.

Tracking the longest positive or negative example prefix during the synthesis process is achieved by keeping a map between different states. More precisely, the set $M : Q \hookrightarrow \mathcal{P}(Q)$ denotes a partial mapping from a state to a set of states. M is a set of ordered pairs over elements $Q \times \mathcal{P}(Q)$, where each pair (q, P) associates state $q \in Q$ with a set of states $P \in \mathcal{P}(Q)$. $M[q]$ denotes the set of states mapped to q in M or an empty set if no association to q exists in M. The function *updateMap*(M, q, P) either adds pair (q, P) to M when there there is no mapping from q in M or augments the existing pair $(q, M[q])$ with $(q, M[q] \cup P)$.

This concludes the general summarisation of APTAs. Because we are using APTAs to synthesise allowlists of Java package and class names, we specialise the input alphabet to cater to this use-case. A well-formed deserialization example is a string representing a fully-qualified Java class name that can be used by the Java reflection API. Examples include *java.lang.String*, *int*, and *[[B*. The rightmost identifier of a deserialization example is referred to as a *class name*, the rest of the identifiers are called *sub-packages* and the character . is a package separator. For instance, in *java.lang.String*, *String* is a class name, while *java* and *lang* are sub-packages. By convention sub-packages start with lowercase letters and class names are capitalised. Thus Σ in our case consists of valid sub-package names ending with the literal '.' and valid class names. In addition, we use $\mathcal{R}_\alpha$ and $\mathcal{R}_\kappa$ to denote regular expressions that match any class or sub-package name and any class name respectively. For instance, *java.lang.*$\mathcal{R}_\alpha$ matches both *java.lang.String* and *java.lang.ref.Cleaner*, whereas *java.lang.*$\mathcal{R}_\kappa$ matches *java.lang.String* only. Regular expression $\mathcal{R}_\kappa \setminus \mathcal{R}$ matches any class name except names matched by $\mathcal{R}$.

Figure 2 shows an APTA example for deserialization. Here *java.lang.String* is a positive example as it is accepted while *java.io.Writer* is a negative example as it is rejected. In this example *value*(q_6) is always positive, *value*(q_5) is always negative and *value*(q_4) is conflicting.

We use APTA$^{\text{DS}}$ to represent APTAs that are specific to deserialization.

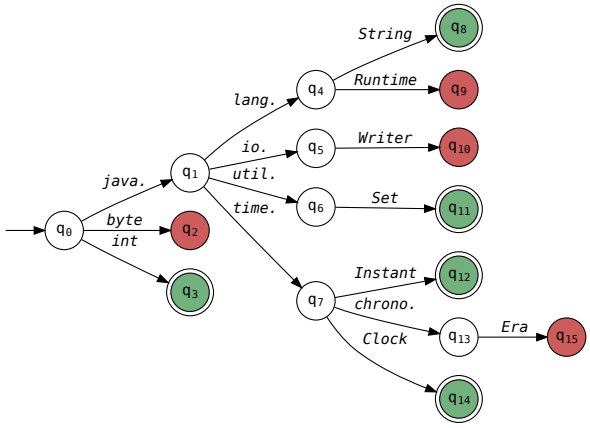

Fig. 2. An APTA representing deserialization examples.

III. Synthesis of Deserialisation Filters

The main goal of our approach is to use a set of labelled deserialisation examples to automatically synthesise a regular expression to act as a deserialisation allowlist. Such a regular expression should accept all positive examples (i.e., allow deserialisation of benign classes), reject all negative examples (i.e., prevent potentially malicious classes from loading) and generalise to unknown examples in a systematic fashion. In this section we present our synthesis algorithm called *ds-prefix*.

Ds-prefix has two main steps. The first is to find the shortest positive prefixes that describe benign examples only and combine them to generate a regular expression. For instance, if $S_+ = \{$*java.lang.Byte*, *java.lang.Short*$\}$ and $S_- = \{$*java.io.Writer*$\}$, then the shortest positive prefix is *java.lang.* as it covers all the positive examples and excludes S_-. This can be represented as the regular expression *java.lang.*$\mathcal{R}_\alpha$ that matches any class belonging to package *java.lang* or any of its sub-packages.

Finding a positive prefix is not always possible. Consider for instance, $S_+ = \{$*java.lang.String*,*java.lang.Short*$\}$ and $S_- = \{$*java.lang.Runtime*$\}$. Generalising to *java.lang.*$\mathcal{R}_\kappa$ is no longer safe because it will also match the negative example, and dropping this prefix altogether results in rejecting the positive example. In *ds-prefix*, such conflicts are resolved via one of the following strategies:

- **Additive Approach**: accept positive examples only via a union of all positive examples: *java.lang.(String + Short)*
- **Subtraction Approach**: reject negative examples and accept any other example belonging to the same package: *java.lang.*$(\mathcal{R}_\kappa \setminus \{$*Runtime*$\})$

The ds-prefix Synthesis Algorithm: At a high-level *ds-prefix* traverses an APTA$^{\text{DS}}$ obtained from a set of deserialisation examples and generates a mapping that comprises discovered

prefixes and conflicting class names (if any). The generated mappings are then combined into the resulting regular expression using either additive or subtraction resolution strategies.

A more formal description of *ds-prefix* is given via procedure DS-PREFIX (Algorithm 1).

Algorithm 1 Inference from Deserialisation Examples

Require: $S = S_- \cup S_+$
Require: $H \in \{\mathit{Additive}, \mathit{Subtraction}\}$
1: **procedure** DS-PREFIX(S)
2: $A \leftarrow$APTA$^{\text{DS}}(S)$, $M \leftarrow \emptyset$, $R_e \leftarrow \emptyset$
3: DS-STATE(q_0, M)
4: **for all** $(q, P) \in M$ **do**
5: **if** $P = \emptyset$ **then**
6: $R_e \leftarrow R_e \cup \{\mathit{value}(q) + \mathcal{R}_\kappa\}$
7: **else if** $F_+ \cap P \neq \emptyset$ **then**
8: $R_e \leftarrow R_e \cup \{\mathit{value}(q) +$ DS-RESOLVE$(P, H)\}$
9: **end if**
10: **end for**
11: **return** $\mathit{join}(R_e)$
12: **end procedure**

The input DS-PREFIX is the set of labelled deserialisation examples S and a conflict resolution strategy H. The first step constructs an APTA$^{\text{DS}}$ from the input examples and initialises an empty mapping M and a resulting per-prefix set of regular expressions R_e (Line 2).

Traversal of the APTA$^{\text{DS}}$ A is described in the procedure DS-STATE (Algorithm 2) that considers a single state at a time. The action performed depends on whether the input state q is an internal state (captures a prefix) or a leaf (describes a class name). An internal state q (Line 2) is mapped to an empty state in M if it is positive (has no negative descendants, Line 4). Otherwise, DS-STATE recursively traverses the immediate descendants of q (Lines 6-8) attempting to find positive prefixes that lie beyond q. If the traversal reaches a leaf (positive prefix cannot be found, Line 10), it is mapped to its parent in M that tracks a longest conflicting or negative prefix. In summary, a mapping (q, P) in M captures a positive prefix given by q if P is empty, a conflicting prefix if P contains positively labelled states and a negative prefix otherwise.

Algorithm 2 State Processing

Require: $q \in Q$
Require: $M : Q \hookrightarrow \mathcal{P}(Q)$
1: **procedure** DS-STATE(q, M)
2: **if** $\mathit{children}(q) \neq \emptyset$ **then**
3: **if** $F_-(q) = \emptyset$ **then**
4: $M \leftarrow M \cup (q, \emptyset)$
5: **else**
6: **for all** $q_d \in \mathit{children}(q)$ **do**
7: DS-STATE(q_d, M)
8: **end for**
9: **end if**
10: **else**
11: $\mathit{updateMap}(M, \mathit{parent}(q), \{q\})$
12: **end if**
13: **end procedure**

The final step of DS-PREFIX generates a regular expression for each positive and conflicting prefix tracked via M and consolidates the results in the set of per-prefix regular expressions R_e. Positive prefixes in M (mappings to empty states) are generalised to $\mathcal{R}_\kappa$. (Line 6). The regular expressions for conflicting prefixes are generated by the function DS-RESOLVE (Algorithm 3) that applies a specified conflict-resolution strategy (Line 8) to generate a regular expression from a mixture of positive and negative class names.

Specifics of conflict resolution is given by the DS-RESOLVE in Algorithm 3. The inputs to DS-RESOLVE is the conflict resolution heuristic (*additive* or *subtraction*) and a set of leaves belonging to the same parent state. The latter represents conflicts: positive and negative class names of the same prefix. DS-RESOLVE first populates sets S_+ and S_- with positive and negative class names respectively (Lines 2-3). A class name (given by some leaf state q) is positive if q belongs to the set of positively labelled states F_+ of APTA$^{\text{DS}}$ and negative if q belongs to F_-. With the *additive* heuristic the resulting regular expression is a union of all positive class names (Line 4). Note that S_+ cannot be empty, as otherwise it is a negative prefix. With the *subtraction* heuristic the resulting regular expression matches any class name except negative (Line 7).

It is worth noting that conflict resolution (via DS-RESOLVE) generalises class names within individual packages (i.e., using $\mathcal{R}_\kappa$). This prevents from matching negative examples that share similar prefixes. This is different to generalising positive prefixes, where any sub-package (i.e., via $\mathcal{R}_\alpha$) is permitted beyond the prefix itself. This is because a positive prefix describes only positive examples. We show an example of this distinction in the following section.

Algorithm 3 Conflict Resolution

Require: $P \in \mathcal{P}(Q)$
Require: $H \in \{\mathit{Additive}, \mathit{Subtraction}\}$
1: **procedure** DS-RESOLVE(P, H)
2: $S_+ \leftarrow \{\ \mathit{symbol}(q) \mid q \in F_+\ \}$
3: $S_- \leftarrow \{\ \mathit{symbol}(q) \mid q \in F_-\ \}$
4: **if** H = *Additive* **then**
5: **return** $\mathit{join}(S_+)$
6: **else**
7: **return** $\mathcal{R}_\kappa \setminus \mathit{join}(S_-)$
8: **end if**
9: **end procedure**

Synthesis Example: We now give an example of using *ds-prefix* to synthesise a regular expression. Consider the following set of input examples S.

$\mathbf{S}_+$	$\mathbf{S}_-$
int	*byte*
java.lang.String	*java.lang.Runtime*
java.util.Set	*java.io.Writer*
java.time.Instant	*java.time.chrono.Era*
java.time.Clock	

The APTA$^{\text{DS}}$ generated from S is shown via Figure 2.

The algorithm first considers the root state q_0. Since the tree of q_0 contains both positive and negative examples DS-

STATE is recursively called on all immediate descendants of q_0. States q_2 and q_3 are leaves, and thus they are mapped to the state q_0 in M. q_1, on the other hand, is a conflicting internal state therefore its descendants (q_4-q_7) are processed recursively. Since q_6 is a positive prefix it is tracked via M that associates q_6 with an empty set. The remaining states q_4, q_5 and q_7 are processed further. The final steps associate remaining states with their leaves in M. The post-traversal mapping M is shown in Table I.

q_0	$\mapsto$	$\{q_2, q_3\}$	q_4	$\mapsto$	$\{q_8, q_9\}$
q_5	$\mapsto$	$\{q_{10}\}$	q_6	$\mapsto$	$\emptyset$
q_7	$\mapsto$	$\{q_{12}, q_{14}\}$	q_{13}	$\mapsto$	$\{q_{15}\}$

TABLE I
MAPPING OF STATES TO LEAVES

Prefix of q_6 is positive and generalised to `java.util.`$\mathcal{R}_\alpha$ that matches any class with sub-package `java.util.`. This is because there are no negative examples sharing this prefix. Prefixes of q_5 and q_{13} are negative and therefore dropped. Prefix of q_7 is conflicting and contains both positive and negative examples. q_7 generates `java.time.(Instant+Clock)` using the *additive* heuristic and `java.time.`$\mathcal{R}_\kappa$ for the *subtraction* heuristic. Note that for the *subtraction* heuristic generalising with $\mathcal{R}_\kappa$ allows any class from `java.time` package (no negative examples in that package) but also rejects negative example `java.time.chrono.Era`. Note that using `java.time.`$\mathcal{R}_\alpha$ will not work as that will then allow the package `java.time.chrono` and all sub-packages and classes under it. Prefixes q_4 and q_0 are also conflicting, with *additive* heuristic generating `java.lang.String` and `int` and *subtraction* generating `java.lang.(`$\mathcal{R}_\kappa$`\Runtime)` and $\mathcal{R}_\kappa$`\byte` respectively. The overall resulting filters generated by using the *additive* and *subtraction* heuristics are shown via Listings 1 and 2 respectively.

`int+java.lang.String+java.time.(Instant+Clock)`
`+java.util.`$\mathcal{R}_\alpha$

Listing 1. Example Regular Expression using Additive Heuristic

$\mathcal{R}_\kappa$`\byte+java.lang.(`$\mathcal{R}_\kappa$`\Runtime)+java.time.`$\mathcal{R}_\kappa$`+`
`java.util.`$\mathcal{R}_\alpha$

Listing 2. Example Regular Expression using Subtraction Heuristic

IV. RESULTS

We now discuss the implementation of *ds-prefix* and present the results of its evaluation using several open-source systems.

A. Implementation

We have implemented *ds-prefix* using the `dk.brics.automaton` library [17] extended to support APTAs. At a high level our synthesiser receives a specification with positive and negative examples, constructs an APTA and synthesises a Java-compatible regular expression.

Translation of the abstract regular expressions described in Section III to Java is straightforward. `.` and `[` characters are escaped because they coincide with metacharacters. For instance, *java.lang.String* generates `java\.`, `lang\.` and `String` for alphabet symbols. Regular expressions $\mathcal{R}_\alpha$ and $\mathcal{R}_\kappa$ are expressed as `.*` and `[^.]+`, and the `\` operator is modelled using zero-width negative lookahead. Finally, the resulting regular expression is enclosed in `^` and `$` anchors to avoid partial matches.

To demonstrate the applicability of our technique to real issues we have integrated *ds-prefix* with a Java monitoring agent built using the ByteBuddy[5] code generation and manipulation library. Our agent has two modes of operation: monitoring and analysis. In the monitoring mode the agent logs names of the deserialised classes allowing to collect examples for synthesis. In the analysis mode the agent instruments Java code at runtime and flags deserialisation that violates the supplied allowlist. The current implementation allows filtering native Java serialisation and serialisation in the Jackson-databind library [1].

B. Vulnerability Detection

In this section we investigate whether *ds-prefix* generates accurate allowlists (RQ 1, Section I). To answer this question we show that *ds-prefix* combined with the monitoring agent detects real vulnerabilities related to deserialisation of untrusted data (CWE-502 [18]) in several popular open-source systems.

Our experimental approach is as follows. We first reproduce a known vulnerability. We then gather benign and malicious examples. The benign examples are typically collected from the runs of systems under test, while the negative examples are collected from previously discovered issues. In the final step we use *ds-prefix* to synthesise an allowlist and use the Java agent to confirm that the vulnerability is detected at runtime. For the purpose of this experiment we configure *ds-prefix* to synthesise filters using the *subtraction* heuristic as it produces more permissive allowlists comparing to the *additive* approach.

1) Apache Batik: Batik [19] is a Java library for manipulation of SVG graphics. Internally, Batik encodes SVG images as XML documents and allows for Java-based serialisation/deserialisation of the DOM object that represents an image.

In Batik versions before `1.10`, during the deserialisation of `AbstractDocument` objects, an arbitrary class name string is read from the input stream, and used to reflectively invoke the default constructor of the corresponding class. This effectively allows deserialisation of any Java object and can be used to construct a remote code execution exploit (CVE-2018-8013 [20]). The patch for this issue checks that the deserialised class implements the `org.w3c.dom.DOMImplementation` interface before invoking its constructor.

To reproduce CVE-2018-8013, we reverted the commit that fixed the issue and wrote a driver that sends a malicious serialised payload that triggers the execution of a user-specified command. Furthermore, because the fixed version will still invoke the constructor of any class that implements `org.w3c.dom.DOMImplementation` interface, we also created a malicious class that implements this interface.

[5]https://bytebuddy.net

To show that our approach can prevent CVE-2018-8013, we first synthesised the allowlist using a dataset consisting of all the publicly available implementations of `org.w3c.dom.DOMImplementation` we could identify. We then ran the custom driver under the supervision of our Java agent configured to use the allowlist (shown below) and confirmed that the exploit was successfully prevented.

```
^(\[Lorg|com\.sun\.org\.apache\.xerces|com\.
    sun\.org\.apache\.xml|org\.apache\.batik|
    org\.apache\.html|org\.apache\.wml|org\.
    apache\.xerces|org\.python|org\.w3c)\..*$
```

The allowlist permits specific classes such as `org.apache.wml.WMLDOMImplementation` or `org.w3c.dom.html.HTMLDOMImplementation`. This is because these are are publicly available classes that do not contain malicious code that lead to the CVE-2018-8013 exploit. Hence they can be used as part of the training set.

2) Jackson-databind: Jackson [21] is a suite of data-processing tools for Java that includes Jackson-databind [1], a component implementing general-purpose data-binding functionality. The Jackson-databind library is an extremely popular tool for serialisation and deserialisation to and from JSON.

Jackson-databind supports *polymorphic type handling* that allows adding additional type information to the serialised JSON object and use that type information at deserialisation time to create an object of a given class. In cases where the deserialised object has a field of a generic type (such as `Object`), an attacker can manipulate the type information and have Jackson-databind create an instance of any class since any class in Java is a subclass of `Object`.

Polymorphic type handling has resulted in over 60 CVEs since 2017 [22]. One the main reasons for such high number of CVEs is that up until Jackson-databind `2.9.x`, the deserialisation targets were checked against a blocklist that was evaded as new deserialisation gadgets were discovered. Starting with Jackson-databind `2.10.x`, deserialisation targets are validated against an allowlist by default. So far this approach has been successful as there have been no reported CVEs related to deserialisation of untrusted data.

To evaluate our approach we reproduced one of the earliest Jackson-databind exploits: CVE-2017-17485 [23]. To confirm that our approach prevents this issue, we generated the following deserialisation filter from a dataset consisting of benign examples from the Jackson databind test suite and malicious examples from its default blocklist.

```
^((\[Lcom|\[Ljava|com\.fasterxml|java\.io|java
    \.lang|java\.text|java\.util\.concurrent)
    \..+|[^.]+|java\.util\.[^.]+)$
```

This above allowlist represents a generalisation of what should be blocked. This filter blocks instances of `FileSystemXmlApplicationContext` from the `org.springframework.context.support` package that trigger CVE-2017-17485 but does not appear in Jackson blocklist. This filter also rejects malicious classes from Jackson databind blocklist such as `org.apache.xalan.xsltc.trax.TemplatesImpl` or `org.codehaus.groovy.runtime.ConvertedClosure`.

To further investigate detection capabilities of *ds-prefix* allowlists we constructed two additional input datasets. Both datasets contain the same positive examples obtained from test runs. The list of negative examples in the first dataset (say D) contains 9 negative examples and has been drawn from Jackson-databind blocklist after discovery and resolution of its first deserialisation exploit. The second dataset (say D') contains 134 negative examples and has been drawn from the latest Jackson-databind blocklist (at the time of experimentation). The list of negative examples in D' captures the blocklist preventing 46 known CVEs (all reported CVEs available at the time of experimentation).

A simple experiment show that the blocklist synthesised from D (`^((\[Lcom|\[Ljava|com\.fasterxml|java)\..+|[^.]+)$`) rejects all but two negative examples from D'. This result indicates a *ds-prefix* allowlist generated from just 9 negative examples would be sufficient to effectively prevent 44 CVEs. Our further experiment with the historic data indicates that the negative examples from the 4th earliest CVE of Jackson-databind (comprising 48 negative examples), is sufficient to prevent all exploits.

3) Olingo: Apache Olingo [24] is a Java library that implements the Open Data Protocol (OData). The *odata-client-proxy* Olingo component (a Java client for OData services) versions `4.0.0` to `4.7.0` is vulnerable to deserialisation of untrusted data (CVE-2019-17556 [25]) because prior to version `4.7.0` the `AbstractService`, which is a public API, deserialised arbitrary objects without checking the supplied data. Olingo developers have eventually fixed this vulnerability by allowlisting the deserialisation targets and restricting them to packages and sub-packages of `org.apache.olingo.*`.

We reproduced this vulnerability by using an affected version of Olingo and supplied a malicious payload that triggered execution of an arbitrary command. To confirm that our approach detects CVE-2019-17556 we used `^org\.apache\.olingo\..+$` allowlist synthesised from positive examples collected from tests and negative examples consisting of class names from known deserialisation gadget chains from ysoserial [26] and a default blocklist of the Jackson-databind library. Notably the filter generated is essentially identical to the custom filter supplied by Olingo developers.

C. Accuracy of Synthesised Allowlists

In our next experiment we investigate whether F1-score of *ds-prefix* is superior to the existing tools (RQ 2, Section I). For this we conduct a K-fold cross-validation study and compare results with several state-of-the-art synthesis techniques. The main reason for selecting cross-validation is to evaluate synthesis in a setting that uses a small number of examples.

For this experiment we use the same datasets that were used to synthesise allowlists in Sections IV-B1 and IV-B2. We exclude the Olingo dataset used in Section IV-B3 because it contains a large number of examples (over 2000), whereas our goal is to investigate F1-score using only a small number of

examples. We further investigate the tools' performance using large datasets in Section IV-D.

Synthesisers: Firstly, we compare *ds-prefix* to classic automata-theoretic techniques for regular inference, namely BlueFringe [8], RPNI [9] and Traxbar [10], [11]. All these algorithms generate prefix tree acceptors from the set of input examples and then apply series of transformations that generate DFAs that represent resulting regular expressions. For the purpose of this experiment we implemented these techniques in Java alongside *ds-prefix* with DFA to regular expression conversion using the algorithm based on Andern's lemma [27]. Another tool used in this experimentation is Search-and-Replace generator (S&R) based on genetic programming [28], [29]. To explore application of S&R to synthesis of allowlists we also experiment with a variation of S&R called S&R-DS that uses an alphabet tailored to deserialisation comprising sub-packages, separators and class names. Finally, we compare *ds-prefix* with our Python implementation of MSA, a synthesis approach based on multiple sequence alignment [30].

Experiment Set-up: For this experiment we randomly shuffle datasets and split them into 5 folds of equal size. Each fold is then treated as a testing dataset with the rest taken for training. For each training dataset we generate a regular expression and check deserialisation examples in the test dataset. In this set-up, a true positive result constitutes matching a positive example or rejecting a negative example, a false negative result is not matching a positive example, and a false positive result is matching a negative example.

Results: The results of this experiment are given in Table II, that shows the average F1-score and standard deviation (in parenthesis) per synthesiser. The table further shows the number of positive and negative examples per-dataset. The results show that *ds-prefix* has the best F1-score of 0.93 for the Batik dataset and 0.99 for Jackson-databind. The results of automata-theoretic techniques (Blue-fringe, RPNI, Traxbar) were slightly worse ranging from 0.83 to 0.90. The results of the MSA and S&R synthesisers were also worse than *ds-prefix*. For these tools, the drop in F1-score was due to low recall, where positive examples failed to be matched. Also, for the Batik datasets, these tools had higher standard deviation, which indicates more diverse results compared to *ds-prefix*.

D. Performance Evaluation

We now investigate performance of *ds-prefix* relative to the state-of-the-art tools on large datasets. For this experiment we use Batik, Olingo and Jackson-databind (Section IV-B) datasets and additionally a large dataset collected from the test runs and the allowlist of XStream[6], yet another popular library for serialisation and deserialisation to and from XML. We note we have not used XStream it in our prior experiments with CVE detection because the current implementation the Java agent lacks transformations that can intercept XStream-style deserialisation. All reported performance results were generated on an 8-core 1.7GHz I5 Intel processor with 16GB RAM, running 64-bit Ubuntu Linux.

[6]https://x-stream.github.io

The results of this experiment is given via Table III that shows the runtime in seconds taken for synthesis and the size of the resulting regular expression in bytes (with one byte representing a single character) per-tool. The results show that *ds-prefix* has best performance of all synthesisers used, taking under a second to synthesise regular expressions even for large datasets. RPNI and Traxbar had the worst performance and could not complete analysis (due to out-of-memory errors) for larger XStream and Olingo datasets. Blue-fringe performed better than RPNI and Traxbar and was able to complete synthesis for all four datasets. With this approach, however, much effort is spent on conversion from a DFA obtained by merging states of an APTA into a regular expression. Regular inference of MSA has a better performance when compared to the automata-theoretic techniques. Even for the largest XStream dataset MSA was able to compile a regular expression in 68 seconds and just a few seconds for the remaining 3 datasets. S&R and S&R-DS were also able to synthesise regular expressions in a reasonable time. The performance of S&R using the alphabet of characters on for the XStream dataset (over 10 minutes) was, however, considerably worse comparing to the other techniques.

E. Auditability of Results

Finally, we inspect features of regular expressions generated by different synthesisers to understand whether the results of *ds-prefix* are manually auditable (RQ 3, Section I).

The structure of regular expression generated by *ds-prefix* are straightforward, as one can identify (or update) prefixes used in their construction. Consider, for instance the regular expression generated for the Batik dataset (Listing 3), where prefixes such as `org\.apache\.batik` or `com\.sun\.org\.apache\.xml` stand out and can be manipulated in a straightforward manner without complex changes to the structure of a regular expression.

The regular expressions produced by automata-theoretic techniques (RPNI, Blue-fringe, Traxbar) on the other hand typically consist of individual characters from the examples and as such can be difficult to understand or amend. Consider, for instance regular expressions generated by Blue-fringe and RPNI for the Batik dataset (Listing 3). Furthermore, as shown by the results of experiments, due to the complexity of the obtained DFAs automatic DFA-to-regular expression conversion can yield extremely long regular expressions. For instance, a 4.7 MB regular expression was generated by Blue-fringe for the XStream dataset.

The regular expressions generated by MSA are similar to the results of *ds-prefix* in that one can clearly identify parts of Java classes. MSA regular expressions, however, can be too specific and restrict matches to specific lengths or concrete classes. For example, the MSA generated expression for Batik (Listing 3) includes `AbstractElement\$Entry` class. Another example is regular expression `^org.apache.olingo.{0,126}$` generated for the Olingo dataset, where class names are restricted by 126 characters. As such, MSA can reject examples from *safe* packages.

Dataset	ds-prefix	Blue-fringe	RPNI	Traxbar	MSA	S&R	S&R-DS
Batik (34/97)	0.93 (0.07)	0.85 (0.12)	0.89 (0.17)	0.87 (0.07)	0.82 (0.16)	0.90 (0.12)	0.76 (0.20)
Jackson (157/135)	0.99 (0.01)	0.83 (0.12)	0.90 (0.03)	-	0.92 (0.03)	0.87 (0.06)	0.99 (0.01)

TABLE II
5-FOLD CROSS-VALIDATION RESULTS SHOWING F1-SCORE AND STANDARD DEVIATION (IN PARENTHESIS) PER SYNTHESISER.

Dataset (pos/neg)	ds-prefix	Blue-fringe	RPNI	Traxbar	MSA	S&R	S&R-DS
Batik (34/97)	0.3s (184B)	1.7s (230B)	6.6s (147B)	7.9s (384B)	1.8s (252B)	35.4s (33B)	4.1s (B)
Jackson (157/135)	0.3s (118B)	5.3s (44K)	12.6s (3Kb)	88.2s (7.9Mb)	2.3s (47B)	49.9s (20B)	8.0s (B)
Olingo (1915/97)	0.5s (26B)	27.3s (893K)	-	-	1.1s (10.6Kb)	29.1s (49B)	3.7s (B)
XStream (3099/97)	0.6s (3.2Kb)	125.7s (4.7Mb)	-	-	68.1s (482B)	600.2s (52B)	242.0s (B)

TABLE III
PERFORMANCE RESULTS OF DIFFERENT SYNTHESISERS SHOWING RUNTIMES (SECONDS) AND SIZES OF RESULTING REGULAR EXPRESSIONS (BYTES).

```
ds-prefix:^(\[Lorg|com\.sun\.org\.apache\.xerces|com\.sun\.org\.apache\.xml|org\.apache\.batik|org\.apache
    \.html|org\.apache\.wml|org\.apache\.xerces|org\.apache\.xml|org\.python|org\.w3c)\..+$
Blue-fringe:^([a-zA-CE-HJLMOPR-X02-46\$\.\[]|[DIN;]([cmopS]|([dtu]|l[enp])([iI]|[mo][enp])*[aelnC])*([
    aenrB-DMOPRT]|([dtu]|l[enp])([iI]|[mo][enp])*E))*[DIN;]([cmopS]|([dtu]|l[enp])([iI]|[mo][enp])*[aelnC])
    *(([dtu]|l[enp])([iI]|[mo][enp])*)?$
RPNI:^([a-ce-ik-mopr-uw-yAC-EG-IL-PTV-X3\$\.\[]|[dnS][del-nptIS]*[a-cf-ikorsuw-yAC-EGHL-PTV-X3\$\.\[])*([
    dnS][del-nptIS]*|([dnS][deptIS]*)?;[elmptI]*)$
MSA:^\[Lorg.apache.batik.dom.AbstractElement\$Entry;$|^com.sun.org.apache.x.{2,5}.internal.{0,8}..{6,25}
    Implementation.{0,4}$|^org.apache.{10,44}ent.{0,9}$|^org.python.apache.{0,27}DOMImplementation.{0,4}$|^
    org.w3c.dom.{0,5}..{4,4}DOMImplementation.{0,3}$
S&R:^[^I]++[^p]++(?:[^m]++[^r]++)++$
S&R-DS:^(\[L?\.?|org\.?|xerces\.?)([^.]+\.?)(xml\.?|html\.?|wml\.?|org\.?|apache\.?|batik\.?|dom\.?|xerces
    \.?)([^.]+\.?)++;?$
```

Listing 3. Regular expressions generated by all synthesisers for Batik dataset

S&R and S&R-DS generated the shortest regular expressions. A closer examination, however, shows that such regular expressions appear to be overly general. Consider for instance a regular expression generated for the Batik dataset (Listing 3). Furthermore, as SNR-generated regular expressions do not include specific patterns for packages (such as *ds-prefix* or MSA) they can be easier to evade.

In summary, our experiments with *ds-prefix* suggest that it is well suited for synthesis of deserialisation filters capable of detecting real security issues. This has been demonstrated in Section IV-B, where *ds-prefix*-generated allowlists were used to detect recent CVEs in popular Java applications using deserialisation. In addition, experiments with Jackson-databind suggest that *ds-prefix* generated allowlists have the potential to prevent new vulnerabilities using a limited set of input examples. Furthermore, as suggested by the comparison with other synthesis techniques in Section IV-D, *ds-prefix* is capable of generating concise and straightforward regular expressions taking under a second even for large datasets. Fast synthesis time makes it easy to re-synthesise filters on-the-fly (e.g., using incoming data), whereas the structure allows for manual customisation. Finally, a cross-validation study discussed in Section IV-C suggests that for the problem of deserialisation filtering *ds-prefix* is more accurate than state-of-the-art tools.

V. RELATED WORK

This section reviews prior work focusing on tools for protection against insecure deserialisation attacks in Java applications and synthesis of regular expressions from examples.

A. Protection Against Insecure Deserialisation Attacks in Java

Deserialisation filtering is a state-of-the-art mitigation strategy for Java deserialisation exploits that validates the contents of an object input stream before the object is deserialised [31], [32]. One of the most popular deserialisation filtering solutions is Java Enhancement Proposal (JEP) 290 that enables filtering of incoming streams of object-serialization data either through a pattern-based specifications or programmatically, via a custom function. JEP290 also provides built-in filters for Java Remote Method Invocation (RMI) Registry, the RMI Distributed Garbage Collector, and Java Management Extensions (JMX). Apache Commons `ValidatingInputStream` [4] and Contrast Security `contrast-rO0` [5] also use approaches that are based on deserialisation filtering. Our approach is related to the deserialisation filtering tools in that they could be used to enforce the synthesised deserialisation filters at runtime.

Another way of protecting an application against deserialisation attacks at runtime has been proposed by Cristalli et al. [33]. Their approach first monitors trusted executions and collects paths comprising invoked methods. The system in question is then run in a sandbox allowing only deserialisation

that matches collected benign paths to proceed. Our approach bears similarity with the technique by Cristalli et al., in that they both construct monitors for deserialisation. One difference however, is that our approach enables generalisation that allows deserialisation of previously unseen classes. Furthermore, adopting our allowlist in practice does not require sandboxing and can be done using deserialisation filtering techniques.

An orthogonal approach for protection against deserialisation attacks is by using machine learning techniques [34]. This approach uses traces generated by test suites to learn a model of correct executions with a symmetric deep neural network. During a validation phase, traces extracted from a running application are classified according to the learned model.

Overall, finding *unsafe* classes that could lead to insecure deserialisation exploits is one of the main issues when dealing with this problem. In a typical scenario an insecure deserialisation vulnerability is exploited through a chain of objects deserialised one after another. Such a sequence is typically referred to as a gadget chain. Some well-known gadget chains are tracked by the *ysoserial* [26] project that also provides tooling to specify custom gadget chains and trigger user-specified commands by exploiting them. *ysoserial* has been used by other tools focusing on preventing deserialisation such as the Serianalyzer [35] static bytecode analyser.

Gadget Inspector [36] focuses on discovery of existing and new gadget chains in a Java application by examining gadgets and deserialisation libraries that exist on the classpath. *Gadget Inspector* uses static analysis, whereas filters generated by *ds-prefix* are aimed to be deployed at runtime.

B. Synthesis of Regular Expressions from Examples

Synthesising regular expressions from examples is a long-standing problem. Classic automata-theoretic synthesis techniques represent the input set of examples using an augmented prefix tree acceptor (APTA) – a tree-shaped deterministic finite automata (DFA) whose states additionally capture positive or negative labels of the input strings. The input APTA is then converted into a DFA by applying a series of transformations (or merges) that generalise the APTA to unknown examples. The final (often skipped) step converts the DFA into a regular expression by applying one of the well-known algorithms [27]. A notable feature of these techniques is that the resulting regular expression has a soundness guarantee in that it matches all positive inputs and does not match negative.

One of the earliest APTA techniques is the state merging an algorithm by Trakhtenbrot and Barzdin [11]. This algorithm traverses the states of the input APTA (assuming that each state is labelled) in a breadth-first order and merges compatible subtrees with matching positive and negative labels. This algorithm has been further extended by Lang [10] to work on the sparsely labelled string sets. Regular Positive Negative Inference (RPNI) [9] is another well-known automata-theoretic approach that builds a tree-shaped finite automata from the set of positive examples and iteratively merges the states of the tree as long as the resulting automaton does not accept negative examples. EDSM (Evidence-Driven State Merging) is an APTA-based algorithm developed during the Abbadingo-1 competition [8]. EDSM generalises the input APTA by merging *most similar* subtrees, where the notion of similarity is given by the scoring function that counts the number of matching labels. EDSM supports arbitrary merges and backtracks if a merge results in accepting negative or rejecting positive examples. Related techniques using backtracking [37] and extensions to EDSM that explore different properties have also been proposed [38]. BlueFringe is another state-merging approach developed during Abbadingo-1. This algorithm performs state merging using the red-blue framework [7] that preserves tree properties in the merged states.

An alternative way of computing a DFA as a generalisation of input examples captured by an APTA is by constraint solving. Such techniques [6], [39]–[41] build a system of constraints (via as relationships between APTA states) and use algorithms that differ in complexity to resolve these constraints. Typically, however, the overhead of constraint solving is too high for these techniques to be feasible for synthesis of deserialisation filters. Application of a constraint solver to address the issue as also been explored [42].

Ds-prefix is related to automata-theoretic techniques in that it also represents the input set of examples using an APTA. The key difference, however, is that the *ds-prefix* synthesises regular expressions directly, by traversing the APTA and examining the states rather than via a series of merges followed by DFA to regular expression conversion.

An orthogonal way of synthesising regular expressions from examples is through an application of genetic programming [12], [13]. This approach models regular expression synthesis using an evolutionary search where individuals are regular expressions, and the fitness function combines Levenshtein distances between detected and the desired strings, and the length of the regular expression. Another orthogonal approach is using multiple string alignment (MSA) [30]. Various synthesis approaches drew inspiration from MSA in bioinformatics, where, given n sequences of characters, the goal is to find an alignment that produces as few insertions and deletions as possible. Given an MSA alignment, a regular expression can be easily derived by converting aligned characters into anchors, and insertions and deletions into wildcards.

VI. Conclusion

In this paper we presented *ds-prefix* – a novel approach to synthesise precise and manually auditable deserialisation allowlists from examples.

The key idea behind *ds-prefix* is to combine state-of-the-art synthesis techniques with deserialisation specificities to produce filters that are well suited to the task at hand. *Ds-prefix* is based on the observation that hand-written deserialisation filters often reason at the level of packages. With that, rather than considering specific classes, *ds-prefix* attempts to find a set of shortest prefixes that describe all positive examples but none of the negative. This is accomplished by leveraging an APTA that represents input examples. However, rather then

merging states and converting the resulting DFA into a regular expression (as is typical with automata theoretic synthesis), *ds-prefix* generates the regular expression via APTA traversal.

Ds-prefix leads to fast synthesis of manually auditable regular expression using a small number of examples. This is because generated regular expressions combine prefixes, rather than characters. Experimentation with several popular open-source components shows that *ds-prefix* filters can prevent known exploits and have the potential to prevent future vulnerabilities. Consider our experiment with Jackson-databind (Section IV-B2), where an allowlist synthesised using only 9 negative examples was sufficient to prevent 44 known CVEs. Finally, the results of experiments with different state-of-the-art synthesisers show that the presented approach is more precise and considerably faster then other synthesisers.

REFERENCES

[1] "Jackson-databind," October 2021. [Online]. Available: https://github.com/FasterXML/jackson-databind

[2] "Spring," October 2021. [Online]. Available: https://spring.io

[3] "JEP 290: Filter Incoming Serialization Data," 2020. [Online]. Available: https://openjdk.java.net/jeps/290

[4] Apache Commons, "ValidatingObjectInputStream," 2021. [Online]. Available: https://commons.apache.org/proper/commons-io/javadocs/api-2.5/org/apache/commons/io/serialization/ValidatingObjectInputStream.html

[5] Contrast Security, "contrast-ro0," 2015. [Online]. Available: https://github.com/Contrast-Security-OSS/contrast-rO0

[6] F. Coste and J. Nicolas, "Regular inference as a graph colouring problem," in *ICML*, July 1997.

[7] P. García, M. Vázquez de Parga, D. López, and J. Ruiz, "Learning automata teams," in *ICGI*, ser. LNCS, vol. 6339. Springer, 2010, pp. 52–65.

[8] K. J. Lang, B. A. Pearlmutter, and R. A. Price, "Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm," in *ICGI*, ser. LNCS, vol. 1433. Springer, 1998, pp. 1–12.

[9] J. Oncina and P. Garcia, "Inferring regular languages in polynomial updated time," *Advances in Structural and Syntactic Pattern Recognition*, vol. 5, pp. 99–108, 1992.

[10] K. J. Lang, "Random DFA's can be approximately learned from sparse uniform examples," in *COLT*, D. Haussler, Ed. ACM, July 1992, pp. 45–52.

[11] B. Trakhtenbrot and Y. Barzdin, *Finite Automata: Behavior and Synthesis*. North-Holland Publishing Company, Amsterdam, 1973.

[12] A. Bartoli, G. Davanzo, A. D. Lorenzo, M. Mauri, E. Medvet, and E. Sorio, "Automatic generation of regular expressions from examples with genetic programming," in *GECCO*. ACM, 2012, pp. 1477–1478.

[13] A. Bartoli, G. Davanzo, A. D. Lorenzo, E. Medvet, and E. Sorio, "Automatic synthesis of regular expressions from examples," *Computer*, vol. 47, no. 12, pp. 72–80, 2014. [Online]. Available: https://doi.org/10.1109/MC.2014.344

[14] A. Bartoli, A. D. Lorenzo, E. Medvet, and F. Tarlao, "Inference of regular expressions for text extraction from examples," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 5, pp. 1217–1230, 2016.

[15] Y. Tang, X. Lu, and B. Xiao, "Generating simplified regular expression signatures for polymorphic worms," in *ATC*, ser. LNCS, vol. 4610. Springer, 2007, pp. 478–488.

[16] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to automata theory, languages, and computation*, ser. Pearson International Edition. Addison-Wesley, 2007.

[17] A. Møller, "dk.brics.automaton – Finite-state automata and regular expressions for Java," 2017. [Online]. Available: http://www.brics.dk/automaton

[18] "CWE-502: Deserialization of Untrusted Data," 2006. [Online]. Available: https://cwe.mitre.org/data/definitions/502.html

[19] "Apache Batik SVG Toolkit," October 2021. [Online]. Available: https://xmlgraphics.apache.org/batik

[20] "CVE-2018-8013," May 2018. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2018-8013

[21] "Jackson," October 2021. [Online]. Available: https://github.com/FasterXML/jackson

[22] "Jackson-databind CVEs," 2021. [Online]. Available: https://www.cvedetails.com/vulnerability-list/vendor_id-15866/product_id-42991/Fasterxml-Jackson-databind.html

[23] "CVE-2017-17485," October 2018. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-17485

[24] "Apache Olingo," October 2021. [Online]. Available: https://olingo.apache.org

[25] "CVE-2019-17556," December 2019. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-17556

[26] C. Frohoff, "ysoserial," 2015. [Online]. Available: https://github.com/frohoff/ysoserial

[27] H. Gruber and M. Holzer, "From finite automata to regular expressions and back - A summary on descriptional complexity," *Int. J. Found. Comput. Sci.*, vol. 26, no. 8, pp. 1009–1040, 2015.

[28] "Search and Replace Generator," 2021. [Online]. Available: https://github.com/MaLeLabTs/SearchAndReplaceGenerator

[29] A. Bartoli, A. D. Lorenzo, E. Medvet, and F. Tarlao, "Automatic search-and-replace from examples with coevolutionary genetic programming," *IEEE Trans. Cybern.*, vol. 51, no. 5, pp. 2612–2624, 2021.

[30] S. A. Aljawarneh, R. A. Moftah, and A. M. Maatuk, "Investigations of automatic methods for detecting the polymorphic worms signatures," *Gener. Comput. Syst.*, vol. 60, pp. 67–77, 2016.

[31] P. Ernst, "Look-ahead Java deserialization," 2013. [Online]. Available: https://www.ibm.com/developerworks/library/se-lookahead

[32] R. Seacord, "Combating java deserialization vulnerabilities with look-ahead object input streams (LAOIS)," June 2017, nCC Group Whitepaper. [Online]. Available: https://www.researchgate.net/publication/318099751_Combating_Java_Deserialization_Vulnerabilities_with_Look-Ahead_Object_Input_Streams_LAOIS

[33] S. Cristalli, E. Vignati, D. Bruschi, and A. Lanzi, "Trusted execution path for protecting java applications against deserialization of untrusted data," in *RAID*, ser. LNCS, vol. 11050. Springer, 2018, pp. 445–464.

[34] Y. Pan, F. Sun, Z. Teng, J. White, D. C. Schmidt, J. Staples, and L. Krause, "Detecting web attacks with end-to-end deep learning," *J. Internet Serv. Appl.*, vol. 10, no. 1, pp. 16:1–16:22, 2019.

[35] M. Bechler, "Serianalyzer," 2015. [Online]. Available: https://github.com/mbechler/serianalyzer

[36] I. Haken, "Automated discovery of deserialization gadget chains," 2018. [Online]. Available: https://i.blackhat.com/us-18/Thu-August-9/us-18-Haken-Automated-Discovery-of-Deserialization-Gadget-Chains-wp.pdf

[37] H. Juillé and J. B. Pollack, "A sampling-based heuristic for tree search applied to grammar induction," in *AAAI*. AAAI Press / The MIT Press, 1998, pp. 776–783.

[38] O. Cicchello and S. C. Kremer, "Beyond EDSM," in *ICGI*, ser. LNCS, vol. 2484. Springer, September 2002, pp. 37–48.

[39] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE Trans. Computers*, vol. 21, no. 6, pp. 592–597, 1972.

[40] A. L. Oliveira and J. P. M. Silva, "Efficient algorithms for the inference of minimum size DFAs," *Mach. Learn.*, vol. 44, no. 1/2, pp. 93–119, 2001.

[41] A. L. Oliveira and S. Edwards, "Limits of exact algorithms for inference of minimum size finite state machines," in *ALT*, ser. LNCS, vol. 1160. Springer, 1996, pp. 59–66.

[42] M. Heule and S. Verwer, "Exact DFA identification using SAT solvers," in *ICGI*, ser. LNCS, vol. 6339. Springer, September 2010, pp. 66–79.