Applications of Synchronized Pushdown Systems

Johannes Späth

Abstract A precise static data-flow analysis transforms the program into a context-sensitive and field-sensitive approximation of the program. It is challenging to design an analysis of this precision efficiently due to the fact that the analysis is undecidable per se. Synchronized pushdown systems (SPDS) present a highly precise approximation of context-sensitive and field-sensitive data-flow analysis. This chapter presents some data-flow analyses that SPDS can be used for. Further on, this chapter summarizes two other contributions of the thesis "Synchronized Pushdown System for Pointer and Data-Flow Analysis" called BOOMERANG and IDE^{al}. BOOMERANG is a demand-driven pointer analysis that builds on top of SPDS and minimizes the highly computational effort of a whole-program pointer analysis by restricting the computation to the minimal program slice necessary for an individual query. IDE^{al} is a generic and efficient framework for data-flow analyses, e.g., typestate analysis. IDE^{al} resolves pointer relations automatically and efficiently by the help of BOOMERANG. This reduces the burden of implementing pointer relations into an analysis. Further on, IDE^{al} performs strong updates, which makes the analysis sound and precise.

1 Introduction

Our economy as well as our society more and more depends on software solutions. In the era of digitalization almost every company hires software developers to build new or integrate existing software solutions to improve workflows and thus the company's productivity, and to monitor processes or experiment with new business models. The increase in demand also increases the amount of software code written.

Additionally, companies move their solutions to cloud resources and let them exchange sensitive data between internal or external services and company machines and desktop computers. The demand in cloud services simplifies cyber-

J. Späth (✉)
Paderborn University, Paderborn, Germany

M. Felderer et al. (eds.), *Ernst Denert Award for Software Engineering 2019*,
https://doi.org/10.1007/978-3-030-58617-1_3

attacks. The services are reachable from within a network. Therefore no service can trust *any* user input, and security must be also taken care of at implementation level.

Spotting security bugs on the software implementation level, however, is a cumbersome task and extremely challenging, even for experts. It is frequently a chain of events within the software that attackers abuse to access the system. Take for example a SQL injection attack; a SQL injection allows an attacker to read, manipulate, or even delete contents of a database. First, this attack requires a hacker to be able to manipulate external input data and second, the execution of a SQL command that relies on the manipulated data. Therefore, it is the combination of at least two lines of code that contribute to software being vulnerable to a SQL injection attack. Modern software projects commonly consist of hundreds of thousands to millions of lines of code [7, 19, 36], and finding the right sequence of events manually is near to impossible, particularly because most parts of modern software are third-party libraries that are developed externally.

This advocates for automated solutions to detect security bugs in code. Static data-flow analysis is one automated technique. Apart from many applications in compilers and bug detection [16, 31, 43], a static data-flow analysis has the capability to detect SQL injections directly within the code base [27, 29]. In general, static data-flow analysis reasons about the flow of program variables within the software without executing it, which means static analysis can be applied and used before the software is even tested. Static analysis can, at least in theory, trace all data-flows along all potential execution paths within software and hereby provides provable guarantees that the analysis does not miss a single pattern it is looking for. This is a helpful property from a security perspective, where missing a single security bug suffices for an attacker to take over the whole system.

However, the size of modern software applications not only challenge manual inspection but even limit automated static data-flow analyses. Static analyses are said to be imprecise and slow. They generate a large amount of false warnings and take hours or days to complete. In all cases, neither developers nor security experts are willing to use data-flow techniques on a daily basis [15].

There are various design dimensions of a static analysis fine-tuning its precision (i.e., reduce the false warnings). A data-flow analysis can be *intra-* or *interprocedural*. In the former, effects of a call site on a data-flow are over-approximated, while in the latter, effects are precisely modelled by analyzing the called method(s). Additionally, an interprocedural data-flow analysis is precise if it is *context-sensitive*, which means the data-flow analysis correctly models the call stack and the data-flow returns to the same call site it enters the method. A design dimension for the static analysis of object-oriented languages is *field-sensitivity*. A field-sensitive data-flow analysis reasons precisely with data-flows in the case the data escapes to the heap, i.e., when it is stored within a field of an object and loaded later during execution again.

Apart from being precise, a static analysis is also expected to guarantee *soundness*. For example, a compiler only applies a code optimization if the optimization does not change the program's behavior under *any* given user input. An analysis detecting unchecked null pointer dereferences better finds *all* critical dereferences

within the program, a single *false negative*, i.e., if the analysis misses reporting an unchecked flow, may lead to a program crash.

In practice, no static analysis can find all optimizations, all bugs, or all vulnerabilities within a program (no false negatives) and detect those with perfect precision (no false positives). False positives and false negatives are the fundamental consequence of Rice's theorem [35], which states that checking any semantic properties of a program is an undecidable problem. Consequently, any model for static analysis is forced to over- or under-approximate the actual runtime semantics of the program. Over-approximations add false positives to the result and reduce the *precision* of the analysis, while under-approximations introduce false negatives and lower the analysis' *recall*.

Apart from the effect on precision and recall, the approximation is also the influencing factor on the performance of a data-flow analysis. An interprocedural data-flow is less efficient to compute in comparison to an intraprocedural analysis. Adding context- or field-sensitivity to an interprocedural analysis introduces additional complexity within the model and negatively affects the computational effort. Therefore, balancing precision, recall, and performance of a static analysis is a tedious task.

The core contribution of the thesis is a new approach to data-flow analyses that balances precision and performance while retaining the analysis' recall. The solution, called *synchronized pushdown systems* (SPDS), models a context-, field-, and flow-sensitive data-flow analysis taking the form of two pushdown systems [9]. One system models context-sensitivity, and the other one models field-sensitivity. Synchronizing the data-flow results from both systems provides the final results of the data-flow analysis. A context- and field-sensitive analysis is undecidable [32] and forces SPDS to over-approximate. SPDS, though, are specifically designed to expose false positives *only* in corner cases for which the thesis hypothesizes (and confirms in the practical evaluation) that they are virtually non-existent in practice: situations in which an improperly matched caller accesses relevant fields in the same ways as the proper caller would.

2 Motivating Examples

In this section, we show several code flaws that a static data-flow analysis can detect. We highlight *null pointer dereference analysis*, *taint analysis*, *typestate analysis* and, an analysis that detects cryptographic misuses. Null pointer dereference analysis is a classical code flaw regularly faced by developers. Taint analysis is primarily used to detect security-related issues such as injection flaws or privacy leaks. Typestate analysis detects misuses of stateful APIs. The research that the thesis presents is fundamental, yet it applies to all these types of data-flow analyses.

2.1 Null Pointer Analysis

Null pointer dereferences cause `NullPointerExceptions`, one of the most common exception faced by Java developers [18, 26]. A static null pointer analysis detects statements in a program that dereference a potentially uninitialized variable. In Java this typically occurs for fields that are neither initialized nor initialized with `null`.

```
1 class Car{
2   Engine engine;
3
4   public class Car(){}
5
6   void drive(){
7     //throws a NullPointerException, if called on blueCar.
8     this.engine.start();
9   }
10
11  void setEngine(Engine e){
12    this.engine = e;
13  }
14
15  public static void main(String...args){
16    Car redCar = new Car();
17    redCar.setEngine(new Engine());
18    redCar.drive();
19
20    Car blueCar = new Car();
21    blueCar.drive();
22  }
23 }
```

Fig. 1 An application that crashes in a `NullPointerException` at runtime in line 8

Figure 1 showcases a program that throws a `NullPointerException` in line 8 when called from `Car::main()` in line 21. The program does allocate two `Car` objects in line 16 and in line 20. For the second object, stored in variable `blueCar`, no call to `setEngine()` is present and the field `engine` remains uninitialized.

Detecting this bug statically is challenging as the analysis needs to be *context-sensitive* to give precise information when the null pointer exception may occur. The analysis needs to distinguish the two objects and the two calling contexts of `drive()` in line 21 and in line 18. Under the former, the program crashes, whereas under the latter calling context, the program does not crash as `setEngine()` has priorly been called.

A common approach to model context-sensitivity is the k-limited call-strings approach, which limits the stack of calls by a fixed level of k. In practice, limits of length 1 to 3 are standard to achieve scalable solutions [24, 28]. For object-oriented program, these small values quickly lead to imprecise or unsound results, depending on if the analysis designer choses to over- or under-approximate. SPDS do not require to approximate the call stack.

```
24 class Application{
25   Map<String,String> requestData = new TreeMap<>();
26   Connection conn = ...;
27
28   /** Entry point to the web application.
29   * The HttpServletRequest object contains the payload.
30   */
31   void doGet(HttpServletRequest req, ...){
32     String val = req.getParameter("data"); //Untrusted data
33     Map<String,String> map = this.requestData;
34     map.put("data", val);
35   }
36
37   /** Executes two SQL commands to store this.requestData to the database.
38   */
39   void writeToDatabase(){
40     Map<String,String> map = this.requestData;
41     Statement stmt = this.conn.createStatement();
42     for(Entry<String,String> entry : map.getEntries()){
43       String key = entry.getKey();
44       String value = entry.getValue();
45       String keyQuery = "INSERT INTO keys VALUES (" + key+ ");";
46       stmt.executeQuery(keyQuery);//No SQL injection
47       String keyValueQuery = "INSERT INTO " + key +
48            " VALUES (" + value + ");";
49       stmt.executeQuery(keyValueQuery); //SQL injection
50     }
51   }
52 }
```

Fig. 2 A web application vulnerable to a SQL injection attack

2.2 Taint Analysis

Injection flaws are the most predominant security vulnerabilities in modern software. Injection flaws occur in a program when untrusted data reaches a statement that executes a command (for instance, `bash`) or when the untrusted data is used to construct a SQL query that is interpreted and executed. In 2017, OWASP[1] lists *Injections* as the top category of vulnerabilities with the highest risk of being exploited. A typical example of an injection attack for a database-backed software system is a *SQL injection*. If a software system contains a SQL-injection vulnerability, the database can be compromised and manipulated, and the system is no longer trustworthy. An attacker can read, add, and even remove data from the database.

A system is vulnerable to a SQL injection attack, if the system does not properly *sanitize* user input and uses the input to execute a dynamically constructed SQL command. Figure 2 demonstrates a minimal back-end of a web application vulnerable to a SQL injection. The back-end maps each incoming request to a call to `doGet()` within the application and hands over a `HttpServletRequest` object

[1] https://www.owasp.org/.

that represents the request with its parameter. Method `doGet()` loads the user-controllable parameter `"data"` from the request object in line 32 and stores the `String` as value into a `TreeMap`. The `TreeMap` is maintained as field `requestData` of the `Application` object.

Assume the application to persist the map to the database at a later time of execution by calling `writeToDatabase`. The method `writeToDatabase` dereferences the field `this.requestData` to variable `map` in line 40 and iterates over all entries of `map`. For each entry, it constructs and executes two SQL queries (calls in line 46 and in line 49). The first query string only includes a `key` of the map, whereas the second query contains both, the `key` and the `value` of each map's entry. As the `value` of the map contains untrusted data, the application is vulnerable to a SQL injection attack in line 49, which executes the query string contained in variable `keyValueQuery`. With a correct sequence of characters, the attacker can end the SQL insert command and execute any other arbitrary SQL command. For example, a command to delete the whole database.

Static data-flow analysis is an effective technique in preventing such injection flaws. However, detecting the SQL injection flaw in the example by means of a data-flow analysis is challenging to implement efficiently if the analysis is required to be precise and sound at the same time (i.e., no false positive and no false negatives). A precise and sound abstraction for the heap is required to model the data-flow through the map.

Injection flaws are detected by a static *taint analysis*, a special form of data-flow analysis. In the case of a taint analysis for SQL injections, a *taint* is any user-controllable (and hence also attacker-controllable and thus untrusted) input to the program. Starting from these inputs, a taint analysis models program execution and computes other aliased variables that are also *tainted*, i.e., transitively contain the untrusted input. When a tainted variable reaches a SQL query, the analysis reports a *tainted flow*. For the code example in Fig. 2, variable `val` in method `doGet()` is tainted initially. To correctly flag the code as vulnerable, the static taint analysis must model variable `value` in line 44 to be aliased to `val`.

A data-flow analysis trivially detects the alias relationship when the analysis uses an imprecise model. For instance, the *field-insensitive* model taints the whole `TreeMap` object when the tainted variable `val` is added to the `map` in line 34. While field-insensitivity is trivial to model, the analysis results are highly imprecise. Not only are the values of the map tainted, but also any key and the field-insensitive analysis imprecisely marks the constructed SQL query in line 45 as tainted. Therefore, a field-insensitive analysis reports a false positive, as it marks line 46 to execute an unsanitized SQL query.

Field-sensitive data-flow analyses track data-flows through fields of objects and are more precise than field-insensitive analyses. A field-sensitive analysis only reports a single SQL injection for the example. However, the detection of the alias relationship between the variables `value` and `val` is more than non-trivial for a field-sensitive static analysis. The analysis must model the complete data-flow through the map, which spans from the call to `put()` in line 34 to the call in line 44 and involves several accesses to the heap. For instance, at the call to `put()` in line 34,

```
1 public V put(K key, V value) {
2   TreeMap.Entry<K,V> parent = //complex computation done
        earlier
3   TreeMap.Entry<K,V> e = new TreeMap.Entry<>(key, value,
        parent);
4   fixAfterInsertion(e);
5 }
6 private void fixAfterInsertion(Entry<K,V> x) {
7   while (x != null && x != root && x.parent.color == RED) {
8     //removed many branches here...
9     x = parentOf(x);
10     rotateLeft(parentOf(parentOf(x)));
11   }
12 }
13 private void rotateLeft(TreeMap.Entry<K,V> p) {
14   if (p != null) {
15     TreeMap.Entry<K,V> r = p.right;
16     p.right = r.left;
17     if (l.right != null) l.right.parent = p;
18     //removed 8 lines with similar field accesses
19     r.left = p;
20     p.parent = r;
21   }
22 }
```

Listing 1 Excerpt code example of `TreeMap` which is difficult to analyze statically.

the value `val` escapes as second argument to the callee's implementation of the method `put()` of the class `TreeMap`.

Listing 1 shows an excerpt of the callee's code taken from the Java 8 implementation[2] of `TreeMap`. The class contains an inner class `TreeMap.Entry` that lists three fields (`parent`, `right`, and `left`), each of type `TreeMap.Entry`. Method `put()` creates a `TreeMap.Entry` that wraps the inserted element (`value`). The `TreeMap.Entry` is then used to balance the tree (call to `fixAfterInsertion()` in line 56). The method `fixAfterInsertion()` iterates over all `parent` entries and calls `rotateLeft()` to shift around elements within the tree (line 62). The latter method stores to and loads from the fields `parent`, `right`, and `left` of the class `TreeMap.Entry`.

The field-sensitive static taint analysis tracks variable `value`, which is the second parameter of method `put()`. To cope with heap-reachable data-flows, field-sensitive analyses commonly propagate data-flow facts in the form of access paths [1, 2, 4, 5, 6, 10, 14, 41, 42]. An access path comprises a local variable followed

[2] http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/eab3c09745b6/src/share/classes/java/util/TreeMap.java.

by a sequence of field accesses, and every field-store statement adds an element to the sequence. The `while`-loop of `fixAfterInsertion` (line 59) in combination with the three field stores (lines 68, 71, and 72) within the method `rotateLeft()` represents a common code pattern[3] that leads to the generation of access paths of all combinations contained in the set $T = \{\texttt{this}.f_1.f_2.\cdots.f_n.\texttt{value} \mid f_i \in \{\texttt{right}, \texttt{left}, \texttt{parent}\}, n \in \mathbb{N}\}$. The data-flow analysis reports the variable `value` of method `writeToDatabase()` to alias to variable `val` of method `doGet()` only if the correct access path exists in the respective set T of the statements retrieving the value from the map (`getEntries()` in line 42 and `getValue()` in line 44).

The set of data-flow facts T is unbounded. Because most static data-flow algorithms require a finite data-flow domain, they typically use k-limiting to limit the field sequence of the access paths to length k [6]. When an access path of length larger than k is generated, the analysis conservatively over-approximates the $(k+1)^{th}$ field. Therefore, not only will the field `value` of a `TreeMap.Entry` of the map be tainted, but any other field will be tainted as well. For example, any `key` inserted into the map imprecisely is tainted as `TreeMap.Entry` has a field `key`. For this particular example, infinitely long field sequences are generated and for any value of k, k-limiting imprecisely reports `key` to alias to `value`.

Access graphs represent one approach that avoids k-limiting [13, 17]. They model the "language" of field accesses using an automaton. Access graphs represent the set T finitely and precisely. However, just as access paths, also access graphs suffer from the state explosion we show in Listing 1. In the illustrated situation, the flow-sensitive analysis must store a set similar to T (not necessarily the same) of data-flow facts, i.e., access graphs, at *every* statement, and potentially *every* context where a variable pointing to the map exists. Given the large size of T, computing the data-flow fixed-point for all these statements is highly inefficient, and the use of access graphs does not improve it.

The thesis presents the solution SPDS that does not suffer from the state explosion, because a pushdown system efficiently represents millions and even infinitely many access paths in *one* concise pushdown automaton holding data-flow results for *all* statements.

2.3 Typestate Analysis

A *typestate analysis* is a static data-flow analysis used, for instance, to detect misuses of Application Programming Interfaces (APIs) and is capable of detecting erroneous API uses at compile time, i.e., before execution. Typestate analyses use an API specification, mostly given in the form of a *finite state machine* (FSM)

[3] Recursive data structures, for instance `LinkedList` and `HashMap`, generate such patterns. Additionally, using inner classes provokes these patterns as the compiler automatically stores the outer class instance within a field of the inner class.

encoding the intended usage protocol of the API. Based on the specification, the analysis verifies the usage of the API within the code. For example, before an object is destructed, it must be in a state marked as accepting state within the FSM.

The API of the type `java.io.FileWriter` shipped with the standard Java Runtime is a textbook example[4] of an API for which a typestate analysis is helpful in preventing resource leaks. The API can be used to write data from the program to a file on the disk.

To use the API, the developer must first construct a `FileWriter` by supplying a `File` object that the `FileWriter` shall write to. Calling the method `write` on the `FileWriter` object with the respective data as argument tells the `FileWriter` which data shall be written into the `File`. Writing the content of a file to disk is an expensive operation delegated to the operation system, and the API delays the respective system calls to the `close()` method of the `FileWriter` object. The API assumes the `close()` method to be called exactly once prior to the destruction of the object. If the user of the API does not call `close()`, the file remains open. The file resource is blocked by the process, and other processes may not read and write the same file and the program has a *resource leak*. Additionally, data is never written to the file as the output is only flushed to the file upon calling `close()`.

Figure 3 shows the finite state machine that represents a correct usage pattern for the API. The state labeled by I is the initial state. The transition into this state is labeled by `<init>` and refers to the constructor of a `FileWriter` object. The accepting states are the states I and C, the latter is the state in which the `FileWriter` object is correctly closed. All transitions into the C state are labeled by `close`. The state machine lists a third state (W) that the object switches into after a `write` call. In this state, data has been written to the `FileWriter` object but not yet persisted to the actual file on disk. Therefore, it is not an accepting state.

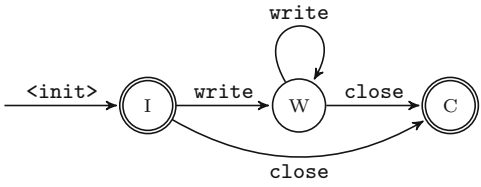

Fig. 3 The API usage pattern encoded as finite state machine for the class `java.io.FileWriter`

The program in Fig. 4 shows a code snippet that uses the API. The code constructs a `FileWriter` object and stores it into field `writer` of the `Example` object. After method `bar()` is called, the field `writer` is loaded and the contained `FileWriter` object is `closed` in line 81.

One challenge of a typestate analysis is to perform *strong updates* when the state of an object changes. At the `close()` call in line 81, it is not clear which actual

[4] In Java 7, `try-with-resources` blocks were introduced to automatically `close` and release file handles. We assume the developer does not use these syntax elements.

Fig. 4 Simple, but challenging program to analysis for a typestate analysis

```
75 class Example{
76   FileWriter writer;
77   public void foo() throws IOException {
78     File file = new File("Data.txt");
79     this.writer = new FileWriter(file);
80     bar();
81     this.writer.close();
82   }
83 }
```

object is closed. If method `bar()` allocates a new `FileWriter` and overwrites the field `writer`, the `FileWriter` allocated in line 79 remains open and the typestate analysis cannot strongly update the state of the latter object. If the analysis detects only a single object to ever be pointed to by field `writer` at statement 81, a strong update can be made. However, the typestate analysis suddenly requires precise points-to information, which is notoriously challenging to obtain efficiently.

Points-to analysis computes points-to information. Despite much prior effort, it is known that a precise points-to analysis does not scale for the whole program [25]. Instead, the typestate analysis only requires points-to information for a rather small subset of all pointer variables, namely the variables pointing to objects that the `FileWriter` is stored within.

The thesis presents BOOMERANG, a demand-driven, and hence efficient, points-to analysis that computes results for a query given in the form of a pointer variable at a statement. BOOMERANG is precise (context-, flow-, and field-sensitive). Based on BOOMERANG, the thesis presents the data-flow framework IDE^{al}, a framework that is powerful enough to encode a typestate analysis that performs strong updates.

2.4 Cryptographic Misuses

Almost any software system processes, stores, or interacts with sensitive data. Such data typically includes user credentials in the form of e-mail addresses and passwords, as well as company data such as the company's income, employee's health, and medical data. Cryptography is the field of computer science that develops solutions to protect the privacy of data and to avoid malicious tampering.

Software developers should have a basic understanding of key concepts in cryptography to build secure software systems. Prior studies [8, 30] have shown that software developers commonly struggle to do so and as a result fail to implement cryptographic[5] tasks securely. While cryptography is a complex and difficult-to-understand area, it also evolves quickly and software developers must continuously remain informed about broken and out-dated cryptographic algorithms and configurations.

[5]Hereafter, used interchangeably with crypto.

```
84 public class Encrypter{
85   private SecretKey key;
86   private int keyLength = 448;
87
88   public Encrypter(){
89     KeyGenerator keygen = KeyGenerator.getInstance("Blowfish");
90     keygen.init(this.keyLength);
91     this.key = keygen.generateKey();
92   }
93
94   public byte[] encrypt(String plainText){
95     Cipher cipher = Cipher.getInstance("AES");
96     //cipher.init(Cipher.ENCRYPT_MODE, this.key);
97     byte[] encText = cipher.doFinal(plainText.getBytes());
98     return encText;
99   }
100 }
```

Fig. 5 An example of a misuse of a cryptographic API

But it is not only the lack of education on the developer's side, common crypto APIs are also difficult to use correctly and securely. For instance, implementing a data encryption with the Java Cryptographic Architecture[6] (JCA), the standard crypto API in Java, requires the developer to combine multiple low-level crypto tasks such as secure key generation, choosing between symmetric or asymmetric crypto algorithms in combination with matching block schemes and padding modes. While the JCA design is flexible to accommodate any potential combination, it yields to developers implementing crypto tasks insecurely by misusing the API.

Figure 5 demonstrates an example code that incorrectly uses some of the JCA's classes for encryption. At instantiation time of an `Encrypter` object, the constructor generates a `SecretKey` for algorithm `"Blowfish"` (parameter to the call to `getInstance()` in line 89) of size 448 (parameter to call in line 90). In line 91, the key is stored to field `key` of the constructed `Encrypter` instance. The `Encrypter` object's public API offers a method `encrypt()`, which, when called, creates a `Cipher` object in line 95. The `Cipher` object is configured to encrypt data using the `"AES"` algorithm (parameter to the call to `getInstance()` in line 95). The developer commented out line 96 that (1) initializes the algorithm's mode and (2) passes the `SecretKey` stored in field `key` to the `Cipher` object. The call to `doFinal()` in line 97 performs the encryption operation and encrypts the content of the `plainText` and stores it in the `byte` array `encText`.

There are four API misuses in this code example. First, the developer commented-out a required call in line 96. Second, if the developer includes the line in the comment, the generated key (`"Blowfish"`) and the encryption cipher (`"AES"`) do not match. Third, and related, the key length of 448 is not suitable for the algorithm AES that expects a size of 128, 192, or 256. Fourth, depending on the

[6] https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html.

crypto provider, AES is used with electronic codebook (ECB) mode. Using ECB results in low entropy within the bytes of `encText`. The first three API misuses throw exceptions at runtime that, using static analysis, could already be detected at compile time. Using ECB, however, does not throw an exception and silently leads to insecure code.

Such API misuses are found in real-world software artifacts. To cope with the detection of such misuses, in [20], we present a domain-specific language (DSL), called CrySL, for the specification of API usage *rules*. We designed a static analysis compiler that, based on a set of CrySL rules, automatically generates a static analysis. The analysis uses BOOMERANG and IDE^{al} and hereby is able to detect misuses even across data-flow constructs such as fields and callings contexts of distinct objects.

3 Synchronized Pushdown Systems

Pushdown systems solve context-free language reachability and have been studied intensively [3, 9, 21, 23, 34]. Synchronized Pushdown Systems (SPDS) [39] are one of the core contributions of the dissertation.

SPDS combines two pushdown systems, the pushdown system of calls and the fields-pushdown system. For each pushdown system, SPDS builds on existing efficient algorithms. When both pushdown systems are synchronized, the results yield a highly precise context- and field-sensitive data-flow analysis.

3.1 Calls-Pushdown System

The calls-pushdown system models the data-flow along the use-def chains of variables and also models the data-flow of variables along call and return methods.

Definition 1 A *pushdown system* is a triple $\mathcal{P} = (P, \Gamma, \Delta)$, where P and Γ are finite sets called the *control locations* and the *stack alphabet*, respectively. A *configuration* is a pair $\langle\!\langle p, w \rangle\!\rangle$, where $p \in P$ and $w \in \Gamma^*$, i.e., a control location with a sequence of stack elements. The finite set Δ is composed of *rules*. A rule has the form $\langle\!\langle p, \gamma \rangle\!\rangle \rightarrow \langle\!\langle p', w \rangle\!\rangle$, where $p, p' \in P$, $\gamma \in \Gamma$, and $w \in \Gamma^*$. The length of w determines the type of the rule. A rule with $|w| = 1$ is called a *normal rule*, one with length 2 a *push rule*, and a rule of length 0 a *pop rule*. If the length of w is larger than 2, the rule can be decomposed into multiple push rules of length 2.

The rules of a pushdown system $\mathcal{P}$ define a relation $\Rightarrow$: If there exists a rule $\langle\!\langle p, \gamma \rangle\!\rangle \rightarrow \langle\!\langle p', w \rangle\!\rangle$, then $\langle\!\langle p, \gamma w' \rangle\!\rangle \Rightarrow \langle\!\langle p', ww' \rangle\!\rangle$ for all $w' \in \Gamma^*$. Based on an initial start configuration c, the transitive closure of the relation ($\Rightarrow^*$) defines a set of reachable configuration $post^*(c) = \{c' \mid c \Rightarrow^* c'\}$. The set $post^*(c)$ is infinite

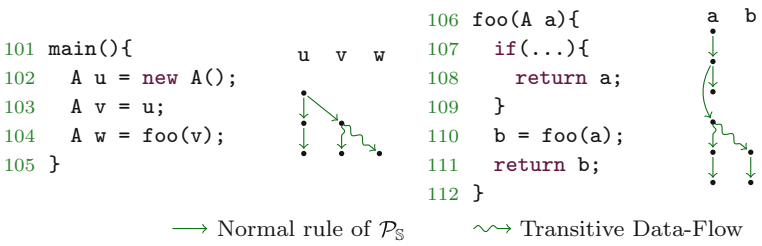

Fig. 6 Example of the data-flow within a recursive program

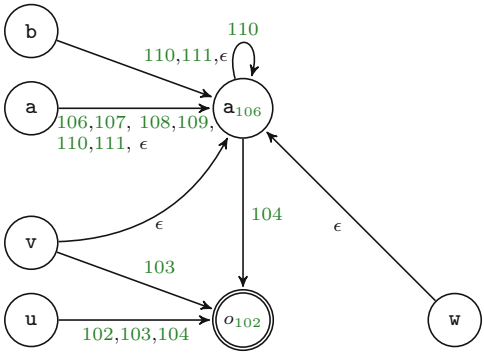

Fig. 7 The automat $\mathcal{A}_{\mathbb{S}}$ after saturation based on $\mathcal{P}_{\mathbb{S}}$

but has a finite representation in the form of an automaton. The algorithm post* constructs this automaton based on a given pushdown system.

Example 1 Figure 6 shows a program that instantiates an object (o_{102}). The object is stored and loaded to and from the local variables `u` and `v`. The variable `v` is used as argument to the call in 104. Next to the program, a directed graph represents the rules of the pushdown system of calls (hereafter $\mathcal{P}_{\mathbb{S}}$). Each edge represents a normal rule of $\mathcal{P}_{\mathbb{S}}$. Additionally, the pushdown system contains the push rule $\langle\!\langle \mathtt{v}, 103 \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{a}, 106 \cdot 104 \rangle\!\rangle$ and the pop rule $\langle\!\langle \mathtt{b}, 111 \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{w}, \epsilon \rangle\!\rangle$.

Based on the pushdown system $\mathcal{P}_{\mathbb{S}}$ and given an initial automaton accepting `u` at 102, the post* algorithms construct an automaton that encodes the reachability of the object o_{102}. Figure 7 depicts the saturated automaton that encodes the reachability of the object o_{102}.

The automaton also encodes the calling context under which each variable reaches (or points-to) the initial object. For instance, variable `a` in line 110 points to o_{102} under call stack 104. The program is recursive and there are potentially infinitely many calls on the call stack. Accordingly, the automaton contains a loop labeled by 110.

3.2 Field-Pushdown System

The call-pushdown system models the calling contexts and its context-sensitivity; however, it is designed field-insensitively and over-approximates access to fields.

Field store and load statements can also be modelled precisely as a pushdown system. Hereby, a field store statement matches a push rule, and a load statement resembles a pop rule. The fields-pushdown system overcomes the imprecision of k-limiting and renders the analysis more efficient.

k-limited analyses with low values of k, e.g., $k = 1, 2, 3$, are efficient to compute but quickly introduce imprecision into the results; higher values of k make the analysis precise but also affect the analysis time exponentially. In our practical evaluation, we compare our abstraction to k-limiting and show that pushdown systems are as efficient as $k = 1$ while being as precise as $k = \infty$ [37].

Definition 2 The *field-PDS* is the pushdown system $\mathcal{P}_{\mathbb{F}} = (\mathbb{V} \times \mathbb{S}, \mathbb{F} \cup \{\epsilon\}, \Delta_{\mathbb{F}})$. A control location of this system is a pair of a variable and a statement. We use $x@s$ for an element $(x, s) \in \mathbb{V} \times \mathbb{S}$. The notation emphasizes that fact x holds *at* statement s. The pushdown system pushes and pops elements of $\mathbb{F}$ to and from the stack. An empty stack is represented by the ϵ field.

We write a configuration of $\mathcal{P}_{\mathbb{F}}$ as $\langle\!\langle x@s,\ f_0 \cdot f_1 \cdot \ldots f_n \rangle\!\rangle$. The configuration reads as follows: The data-flow at statement s is accessible via the access path $x.f_0 \cdot f_1 \cdot \ldots f_n$.

$\mathcal{P}_{\mathbb{S}}$ and $\mathcal{P}_{\mathbb{F}}$ have similar set of rules. The major differences are at field store and load statements. A field store generates a push rule, a field load statement a pop rule.

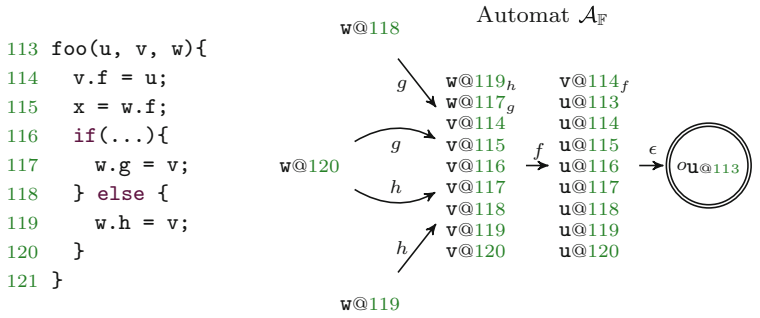

Fig. 8 Example code and $\mathcal{A}_{\mathbb{F}}$ automaton. The automaton $\mathcal{A}_{\mathbb{F}}$ is constructed based on the initial configuration $\langle\!\langle \mathtt{u}@114, \epsilon \rangle\!\rangle$ and saturated based on the rules provided in Table 1

Example 2 Figure 8 depicts a program code containing three field stores and one field load statement. Table 1 lists the push and pop rules of $\mathcal{P}_{\mathbb{F}}$. Given the initial configuration $\langle\!\langle \mathtt{u}@114, \epsilon \rangle\!\rangle$, algorithm post* computes the automaton given in Fig. 8. This automaton contains field- and flow-sensitive data-flow results. For instance, the automaton encodes that the object initially contained in variable u is also reachable

Table 1 Push and pop rules contained in $\Delta_{\mathbb{F}}$ of $\mathcal{P}_{\mathbb{F}}$ for the program code in Fig. 8. The wildcard ($*$) of the push rules represents a rule for any $g \in \mathbb{F}$

Push Rules
$\langle\!\langle \mathtt{u@113}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{v@114}, \mathtt{f} \cdot * \rangle\!\rangle$
$\langle\!\langle \mathtt{v@116}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{w@117}, \mathtt{g} \cdot * \rangle\!\rangle$
$\langle\!\langle \mathtt{v@118}, * \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{w@119}, \mathtt{h} \cdot * \rangle\!\rangle$

Pop Rules
$\langle\!\langle \mathtt{w@114}, \mathtt{f} \rangle\!\rangle \rightarrow \langle\!\langle \mathtt{w@115}, \epsilon \rangle\!\rangle$

via `w.g.f` and `w.h.f`. The automaton does not contain a node that references `x`, which means variable `x` of the program does not point to the same object as `u`.

3.3 Synchronization of Call-PDS and Field-PDS

While the call-pushdown system is field-*in*sensitive, the field-pushdown system is context-*in*sensitive. Synchronized pushdown systems overcome the weaknesses of each system and yield context- and field-sensitive data-flow results. The key idea is to synchronize the results of the saturated post* automaton of both pushdown systems. Note, the synchronization is *not* an automaton intersection. Both automata use different stack alphabets and encode different languages.

Definition 3 For the call-PDS $\mathcal{P}_{\mathbb{S}} = (\mathbb{V}, \mathbb{S}, \Delta_{\mathbb{S}})$ and the field-PDS $\mathcal{P}_{\mathbb{F}} = (\mathbb{V} \times \mathbb{S}, \mathbb{F} \cup \{\epsilon\}, \Delta_{\mathbb{F}})$, the *synchronized pushdown systems* are the quintuple SPDS $= (\mathbb{V}, \mathbb{S}, \mathbb{F} \cup \{\epsilon\}, \Delta_{\mathbb{F}}, \Delta_{\mathbb{S}})$. A configuration of SPDS extends from the configuration of each system: A *synchronized configuration* is a triple $(v, s, f) \in \mathbb{V} \times \mathbb{S}^+ \times \mathbb{F}^*$, which we denote as $\langle\!\langle v.f_1 \cdot \ldots \cdot f_m @ s_0^{s_1 \cdot \ldots s_n} \rangle\!\rangle$ where $s = s_0 \cdot s_1 \cdot \ldots \cdot s_n$ and $f = f_1 \cdot \ldots f_m$. For synchronized pushdown systems, we define the set of all reachable synchronized configurations from a start configuration $c = \langle\!\langle v.f_1 \cdot \ldots \cdot f_m @ s_0^{s_1 \cdot \ldots s_n} \rangle\!\rangle$ to be

$$post_{\mathbb{SF}}(c) = \{\langle\!\langle w.g @ t_0^{t_1 \cdot \ldots t_n} \rangle\!\rangle \mid \langle\!\langle w @ t_0, g \rangle\!\rangle \in post^*_{\mathbb{F}}(\langle\!\langle v @ s_0, f \rangle\!\rangle) \wedge \langle\!\langle w, t \rangle\!\rangle \in post^*_{\mathbb{S}}(\langle\!\langle v, s \rangle\!\rangle)\}. \quad (1)$$

Hence, a synchronized configuration c is accepted if $\langle\!\langle v, s_0 \cdot \ldots \cdot s_n \rangle\!\rangle \in \mathcal{A}_{\mathbb{S}}$ and $\langle\!\langle v @ s_0, f_1 \cdot \ldots \cdot f_m \rangle\!\rangle \in \mathcal{A}_{\mathbb{F}}$, and $post_{\mathbb{SF}}(c)$ can be represented by the automaton pair $(\mathcal{A}_{\mathbb{S}}, \mathcal{A}_{\mathbb{F}})$, which we refer to as $\mathcal{A}^{\mathbb{F}}_{\mathbb{S}}$.

Intuitively, a configuration SPDS is accepted, only if the field automaton *and* the call automaton accept the configuration.

Example 3 Figure 9 shows an example of a program with a data-flow path of interwined calling context and field accesses. A directed graph below the code visualizes the data-flow. Vertical edges correspond to calling context (push rules $\widehat{=}$ opening parentheses / pop rule $\widehat{=}$ closing parentheses), while horizontal edges

```
122 bar(u, v){
123   v.h = u;
124   w = foo(v);
125   x = w.g;
126   y = x.f;
127 }
```

```
128 foo(p){
129   q.g = p;
130   return q;
131 }
```

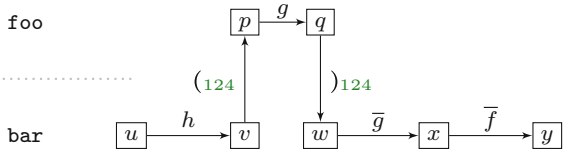

Fig. 9 A code example for and a representation of the code as directed graph

correspond to field store and loads. A field label with a line on top, e.g., $\overline{f}$, means the field f is loaded. SPDS computes reachability within this graph.[7] For the program, SPDS computes variable y *not* to point to u@122. The word along the path from y to u is $h \cdot (_{124} \cdot g \cdot)_{124} \cdot \overline{g} \cdot \overline{f}$.

The parentheses $(_{124}$ and $)_{124}$ are properly matched in the context-free language, and the path is context-sensitively feasible. However, the path is not feasible in a field-sensitive manner. The field store access g matches the field load access $\overline{g}$, however, the field store of h does not match the field load $\overline{f}$.

Any context- and field-sensitive data-flow analysis is undecidable [32]. Therefore, also SPDS must over-approximate. Indeed, it is possible to construct cases in which the analysis returns imprecise results [37]. In the evaluation however, we were not able to find such cases in practice.

4 Boomerang

A points-to set is a set of abstract objects (e.g., allocation statements) that a variable may point-to at runtime. SPDS does *not* compute full points-to set, but only a subset of the points-to set of a variable. Points-to analysis is a non-distributive problem [33, 40], SPDS, however, propagates distributive information and (intentionally) under-approximates the points-to set of a variable. Field accesses allow indirect data-flows that are non-distributive. BOOMERANG is a pointer analysis that builds on top of SPDS and computes points-to and all alias sets. All parts that can be computed in a distributive fashion using SPDS, non-distributive parts are handled as an additional fixed point computation. BOOMERANG is demand driven and answers queries. A query is a variable at a statement that BOOMERANG computes the points-to set for. For each query, BOOMERANG carefully combines forward- and backward-directed

[7] SPDS computes the reachability within the two automata $\mathcal{A}_{\mathbb{S}}$ and $\mathcal{A}_{\mathbb{F}}$. To keep the visualization simple, the automata are omitted.

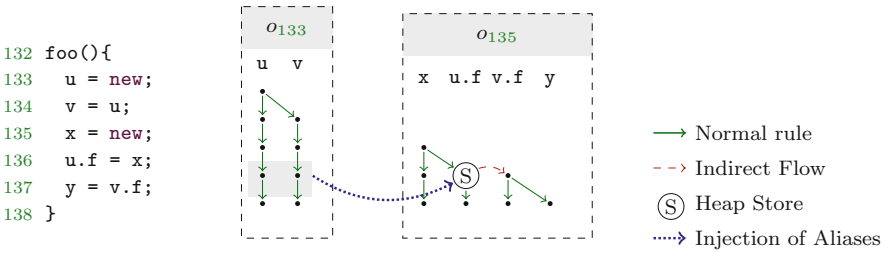

Fig. 10 Non-distributivity of pointer relations

data-flow analyses and focuses on the minimal program slice necessary to answer each query.

Example 4 Figure 10 is a minimal example showing that pointer relations are non-distributive. The program instantiates two objects o_{133} and o_{135} in lines 133 and 135. During runtime `x` and `y` both variable point to the object o_{135}. The object is stored to access path `u.f` in line 135, which at runtime, also changes the content of access path `v.f` as `u` and `v` alias. Using only a single SPDS, the static analysis only propagates the data-flow in line 135 from `x` to `u.f` but not to `v.f`. Therefore, a SPDS does not deliver sound points-to information. BOOMERANG, however, instantiates one SPDS per object allocation and uses an additional fixed-point iteration to connect indirect data-flows across multiple SPDS. Figure 10 indicates such indirect data-flow in form of the blue dashed arrow.

A single program frequently allocates several thousand of objects, and computing data-flow for each of those object does not scale if the analysis is field- and context-sensitive. Instead of starting at every allocation site, BOOMERANG uses a demand-driven design. A backward analysis decides which object allocation is relevant for a single forward query [40].

5 Typestate Analyses Based on IDE^{al}

With IDE^{al} the dissertation further extends SPDS and BOOMERANG. IDE^{al} is a general framework for data-flow analysis and can be used for typestate analysis or API usage mining. IDE^{al} lifts $\mathcal{P}_{\mathbb{S}}$ and associated a weight to each rule. The new idea of IDE^{al} is to use weights to carry additional information along each data-flow path, for instance, to encode typestate information as motivated in Sect. 2.3.

For a typestate analysis, the weights carry the actual state the object resides in at every program point. Previous research on typestate analysis [11, 12] encoded the state of the object as additional information within the data-flow domain, which unnecessarily increases the state space of the analysis and leads to state explosion. With weights, IDE^{al} separates alias information from typestate information and

reduces the state space of the analysis domain, enabling additional performance benefits.

As motivated in Sect. 2.3, in the case of typestate, also aliasing of variables plays an important role and IDE^{al} resorts to BOOMERANG to trigger on-demand queries to perform strong updates of the states [38].

In several experiments presented within publications [38, 39] as well as in the dissertation [37], we compare an IDE^{al}-based typestate analysis to a state-of-the-art typestate analysis, we show that the new concept of separating alias information and weights for typestate yields additional performance benefits.

6 CogniCrypt

BOOMERANG, IDE^{al}, and SPDS also form the base technology behind the static analysis component of the tool CogniCrypt.[8] CogniCrypt is an Eclipse plugin and is part of CROSSING, an interdisciplinary collaborative research center at the Technical University of Darmstadt. CogniCrypt supports software developers in correctly using cryptographic APIs within their software implementations. CogniCrypt accepts rules as input and automatically compiles them into an IDE^{al}- and BOOMERANG-based static analysis. The rules are written in a new domain-specific language, called CrySL.

6.1 The CrySL Language

To detect such API misuses, Krüger et al. [20] designed CrySL, a domain-specific language that allows the specification of crypto API uses. CrySL defines a whitelist approach that specifies correct uses of an API. $CryptoAnalysis$ is the component that compiles a set of rules to a static analysis. Executing the static analysis on program code reports parts of the code that deviate from the specification. We briefly introduce the main semantics of the language in this section and discuss the basic design of $CryptoAnalysis$. The language definition and CrySL rule specifications[9] are not part of the dissertation.

With the CrySL specifications for the JCA, $CryptoAnalysis$ is able to detect all four crypto-related issues showcased in Fig. 5. We discuss the important syntax elements of CrySL based on a minimal[10] CrySL specification covering the misuses in Fig. 5. We refer to the original work [20] for the definition of all syntax elements of CrySL.

[8] https://www.eclipse.org/cognicrypt/.

[9] https://github.com/CROSSINGTUD/Crypto-API-Rules.

[10] The complete specification is found at https://github.com/CROSSINGTUD/Crypto-API-Rules.

```
SPEC javax.crypto.KeyGenerator
OBJECTS
  int keySize;
  javax.crypto.SecretKey key;
  java.lang.String algorithm;
EVENTS
  Get: getInstance(algorithm);
  Inits: init(keySize);
  GenerateKey: key = generateKey();
ORDER
  Gets, Inits, GenerateKey
CONSTRAINTS
  algorithm in {"AES", "Blowfish", ...};
  keySize in {128, 192, 256};
ENSURES
  generatedKey[key, algorithm];
```

(a)

```
SPEC  javax.crypto.Cipher
OBJECTS
  java.lang.String trans;
  byte[] plainText;
  java.security.Key key;
  byte[] cipherText;
EVENTS
  Get: getInstance(trans);
  Init: init(_, key);
  doFinal: cipherText = doFinal(plainText);
ORDER
  Get, Init, (doFinal)+
CONSTRAINTS
  part(0, "/", trans) in {"AES", "Blowfish", "DESede", ..., "RSA"};
  part(0, "/", trans) in {"AES"} => part(1, "/", trans) in {"CBC"};
REQUIRES
  generatedKey[key, part(0, "/", trans)];
ENSURES
  encrypted[cipherText, plainText];
```

(b)

Fig. 11 Two simplified CrySL rules for the JCA. (**a**) CrySL rule for `javax.crypto.KeyGenerator`. (**b**) CrySL rule for `javax.crypto.Cipher`

A CrySL specification is comprised of multiple CrySL *rules*. Each CrySL rule starts with **SPEC** clause specifying the type of the class that the CrySL rule is defined for. Figure 11 depicts two CrySL rules for the classes `javax.crypto.Cipher` and `javax.crypto.KeyGenerator`. The **SPEC** clause is followed by an **OBJECTS** block that defines a set of *rule members*. The values of the rule members are then constrained on within the **CONSTRAINTS** block. For instance, the **CONSTRAINTS** for the rule to `KeyGenerator` restricts the rule member `keySize` in line 154 to the values `128`, `192`, or `256`. When using a `KeyGenerator`, the integer value for `keySize` must be one of the listed values.

The **EVENTS** block defines labels (e.g., `Get` in line 147 and `Inits` in line 148), each label is a set of events. An event is an invocation of a method and is defined via the method signature. For example, label `Inits` is defined as the event of calling the method with signature `init(int keySize)` (line 148). The parameter name (`keySize`) matches the name of a rule member, and when the program calls the event's method, the value of the parameter of the call is bound to the rule member `keySize`, which means that the parameter must satisfy the given constraint.

The labels defined within the **EVENTS** block are used in the **ORDER** block. The **ORDER** clause lists a regular expression (inducing a finite state machine) over the labels and defines the usage pattern (i.e., typestate property) of the specified type. Each object of the specification is required to follow the defined usage pattern. For instance, the specification for `KeyGenerator` expects each object of its type to call any method of the label `GetInstance` prior to any of the `Inits` call followed by a `GenerateKey` call. The **ORDER** specification for `Cipher` uses a `+` for the label `doFinal`, indicating that the method `doFinal()` must be called at least once and arbitrary additional calls of the method can follow.

The remaining two blocks are the **REQUIRES** and **ENSURES** block of a rule. Each line of these blocks lists a *predicate*. A predicate is defined by a name followed by a list of parameters. CrySL predicates cover the specification of the interaction of multiple objects of different types. The `KeyGenerator` rule lists a predicate `generatedKey` with two parameters `key` and `algorithm` in the **ENSURES** block in line 156. When an object of type `KeyGenerator` is used according to the specification in the **CONSTRAINTS**, **ORDER**, and **REQUIRES** block, the predicate listed in the **ENSURES** block is generated for the object. Other CrySL rules that interact with `KeyGenerator` objects can list the predicate in their **REQUIRES** block. For instance, the CrySL rule `Cipher` lists the predicate `generatedKey` as a required predicate in line 173.

6.2 *Compiling CrySL to a Static Analysis*

CryptoAnalysis is a static analysis compiler that transforms CrySL rules into a static analysis. Internally, *CryptoAnalysis* is composed of three static sub-analyses: (1) an IDE^{al}-based typestate analysis, (2) a BOOMERANG instance with extensions to extract `String` and `int` parameters on-the-fly and (3) an IDE^{al}-based

taint analysis (i.e., all weights are identity). The three static analyses deliver input to a constraint solver that warns if any part of the CrySL specification is violated.

Example 5 We discuss a walk-through of *CryptoAnalysis* based on the CrySL specification defined in Fig. 11, and the code snippet provided in Fig. 5. *CryptoAnalysis* first constructs a call graph and computes call-graph reachable allocation sites for in CrySL-specified types. Factory methods can also serve as allocation sites. For example, the factory methods `getInstance()` of `Cipher` and `KeyGenerator` internally create objects of the respective type, and *CryptoAnalysis* considers these calls as allocations sites. In the code example in Fig. 5 the allocation sites are the objects o_{89} and o_{95}.

Starting at the allocation sites, *CryptoAnalysis* uses IDE^{al} to check if the object satisfies the **ORDER** clause of the rule. The call sequence on the `KeyGenerator` object o_{89} satisfies the required typestate automaton defined as regular expression in the **ORDER** block. Opposed to that, the `Cipher` object o_{95} does not satisfy the **ORDER** clause, because the developer commented out line 96. *CryptoAnalysis* warns the developer about the violation of this clause (line 168).

CryptoAnalysis also extracts `String` and `int` parameters of events (statements that change the typestate) to bind the actual values to the rule members of a CrySL rule. For instance, the `getInstance("Blowfish")` call in line 89 binds the value `"Blowfish"` to the rule member `algorithm` of the CrySL rule for `KeyGenerator`. In this example, the `String` value is easy to extract statically, but it might also be defined elsewhere in the program. For example, the value binding for the rule member `keySize` is the actual `int` value flowing to the `init` call in line 90 as a parameter. The actual value is loaded from the heap, because it is the value of the instance field `keyLength` of the `Encrypter` object. Therefore, *CryptoAnalysis* triggers a BOOMERANG query for 90 to find the actual `int` value of the field.

To conclude, *CryptoAnalysis* infers that object o_{89} generates a `SecretKey` for the algorithm `"Blowfish"` with a key length of 448 in line 91. The `KeyGenerator` rule disallows the chosen key length (**CONSTRAINTS** in line 154), and *CryptoAnalysis* warns the developer to choose an appropriate `keySize`.

Assume the developer to change the code to use an appropriate value for `keySize`, and the `KeyGenerator` is used in compliance to its CrySL specification, then *CryptoAnalysis* generates the predicate `generatedKey` for the `SecretKey` object stored to field `key` of the `Encrypter` instance as expected.

If, additionally, the developer includes the `init` call on the cipher object in line 96, (1) the **ORDER** clause of the CrySL rule for `Cipher` is satisfied and (2) the `generatedKey` predicate flows via the field `this.key` to the `Cipher` object o_{95}. As the `Cipher` rule **REQUIRES** the predicate (line 173), the **ORDER** and **REQUIRES** blocks for the object o_{95} are satisfied.

However, the **CONSTRAINTS** for object o_{95} are still not satisfied. Therefore, *CryptoAnalysis* reports that (1) the key is generated for algorithm `"Blowfish"`, and this selection does not fit the algorithm chosen for `Cipher` (`"AES"`) and (2) when using algorithm `"AES"`, one should use it in `"CBC"` mode (**CONSTRAINTS** in

line 171). When the developer fixes these two mistakes, *CryptoAnalysis* reports the code to correctly use the JCA with respect to the CrySL rule.

6.3 Evaluation on Maven Central

Maven Central is the most popular software repository to which developers can publish their software artifacts. Publishing allows other developers to easily access and include the software into their own projects. At the time the experiment was conducted, over 2.7 million software artifacts were published at Maven Central.

The repository contains artifacts in different versions. For one experiment of the thesis, we run *CryptoAnalysis* on all artifacts in their latest versions of Maven Central, a total of 152.996 artifacts. For over 85.7% of all crypto-using Maven artifacts, the analysis terminates in under 10 min, and on average each analysis takes 88 s. Given that *CryptoAnalysis* performs a highly precise and sophisticated static analysis, these results are promising. Unfortunately, we also discovered that many artifacts use the JCA insecurely, and 68.7% of all crypto-using Maven artifacts contain at least one misuse.

Example 6 We want to elaborate on one finding more closely, because it shows the capability of the analysis. Listing 2 shows a code excerpt of an artifact that uses a `KeyStore` object. A `KeyStore` stores certificates and is protected with a password. A `KeyStore` object has a method `load()` whose second parameter is a password. The API expects the password to be handed over as a `char[]` array. The `KeyStore` API explicitly uses the primitive type instead of a `String`, because `Strings` are immutable and cannot be cleared.[11] However, many implementations convert the password from a `String` and hereby introduce a security vulnerability; when not yet garbage collected, the actual password can be extracted from memory, e.g., via a memory dump.

CryptoAnalysis detects the two security vulnerabilities code presented in Listing 2. First, the password is converted from a `String` object via a call to `toCharArray()` to the actual array (line 199), i.e., during the execution of the code the password is maintained in memory as `String`. Second, under some conditions (lines 178, 182, and 189 must evaluate to *true*), the password is hard-coded.

CryptoAnalysis reports a **`CONSTRAINTS`** error on this example, because the `String pass` (highlighted by the green box) in line 199 may contain the `String "changeit"` as it is defined in line 179 (also highlighted). The data-flow corresponding to the finding is non-trivial to detect manually; however, *CryptoAnalysis* is able to do so by the support of BOOMERANG. *CryptoAnalysis* triggers a BOOMERANG query for the second parameter of the `load()` call in line 199 and finds the `toCharArray()` call. From that call, the analysis traces the variable `pass`

[11] https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#PBEEx.

```
23 protected String getKeystorePassword(){
24   String keyPass = (String)this.attributes.get("keypass");
25   if (keyPass == null) {
26     keyPass = "changeit";
27   }
28   String keystorePass = (String)this.attributes.get("
         keystorePass");
29   if (keystorePass == null) {
30     keystorePass = keyPass;
31   }
32   return keystorePass;
33 }
34 protected KeyStore getTrustStore(){
35   String truststorePassword = getKeystorePassword();
36   if (truststorePassword != null) {
37     ts = getStore(..., ..., truststorePassword);
38   }
39   return ts;
40 }
41 private KeyStore getStore(String type, String path, String
       pass){
42   KeyStore ks = KeyStore.getInstance(type);
43   if ((!"PKCS11".equalsIgnoreCase(type)) && ...){
44     ...
45   }
46   ks.load(istream, pass.toCharArray());
47   return ks;
48 }
```

Listing 2 Simplified real-world code example with a hard-coded password

in method `getStore()` and finds it to be a parameter of `getStore()`, and the data-flow propagation continues at invocations of the method. The method `getStore()` is called in line 190, where BOOMERANG data-flow propagation follows the variable `truststorePassword`. This variable is assigned the return value of the call site in line 188. The backward data-flow analysis continues in line 185 and eventually finds the allocation site `"changeit"` in the highlighted line with the line number 179. Eventually, *CryptoAnalysis* reports that variable `pass` is of type `String` and that it may contain the hard-coded password `"changeit"`.

7 Conclusion

Finding an acceptable balance between precision, recall, and performance of a static analysis is a tedious task when designing and implementing a static analysis. With SPDS, BOOMERANG, and IDE^{al}, the dissertation presents new approaches to static data-flow analysis that is demand-driven context-, field-, and flow-sensitive, or in short precise and efficient.

In this chapter, we first motivate (Sect. 2) various applications ranging from null pointer analysis to typestate analysis and next detail on the contributions SPDS, BOOMERANG, and IDE^{al}.

With SPDS (Sect. 3), we present a precise and efficient solution to a known to be undecidable problem [32]. SPDS synchronizes the results of two pushdown systems, one that models field-sensitivity, and a second that models context-sensitivity. SPDS presents a theoretical as well a practical new model to data-flow analysis. The new formal approach to data-flow analysis also enables a direction for future work, for instance, to additionally summarize the pushdown systems [22].

The demand-driven pointer analysis BOOMERANG, presented in Sect. 4, addresses pointer analysis, which is known to be hard to scale. BOOMERANG gains efficiency as it separates the distributive parts of a non-distributive propagation into efficiently SPDS-solvable sub-problems.

IDE^{al} (Sect. 5) extends the ideas of the distributive propagations of BOOMERANG and additionally propagates weights along the data-flow path. The weights allow data-flow analyses to model typestate analyses. In an experiment presented within the dissertation, we compare an IDE^{al}-based typestate analysis to a state-of-the-art typestate analysis and show the efficiency benefit of distributive propagation.

Lastly, we showcase the application of SPDS, BOOMERANG, and IDE^{al} within the tool CogniCrypt. By the use of the domain-specific language CrySL, CogniCrypt is able to statically detect cryptographic misuses.

Acknowledgments My high appreciation to all my co-authors of the work, who largely shaped and influenced this work: Karim Ali, Eric Bodden, Stefan Krüger, Johannes Lerch, Mira Mezini, and Lisa Nguyen Quang Do. Also, I also want to thank the Fraunhofer-Gesellschaft for supporting this research through a Fraunhofer Attract grant.

References

1. Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *PLDI*, 2014.
2. George Balatsouras, Kostas Ferles, George Kastrinis, and Yannis Smaragdakis. A Datalog Model of Must-Alias Analysis. In *International Workshop on State Of the Art in Java Program analysis, (SOAP)*, pages 7–12, 2017.

3. Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *International Conference on Concurrency Theory (CONCUR)*, pages 135–150, 1997.
4. Ben-Chung Cheng and Wen-mei W. Hwu. Modular Interprocedural Pointer Analysis Using Access Paths: Design, Implementation, and Evaluation. In *PLDI*, pages 57–69, 2000.
5. Arnab De and Deepak D'Souza. Scalable Flow-Sensitive Pointer Analysis for Java with Strong Updates. In *ECOOP*, pages 665–687, 2012.
6. Alain Deutsch. Interprocedural May-Alias Analysis for Pointers: Beyond *k*-limiting. In *PLDI*, 1994.
7. Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at facebook. *Commun. ACM*, 62(8):62–70, 2019.
8. Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *International Conference on Computer and Communications Security (CCS)*, pages 73–84, 2013.
9. Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. Efficient Algorithms for Model Checking Pushdown Systems. In *International Conference on Computer Aided Verification (CAV)*, pages 232–247, 2000.
10. Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. Bottom-Up Context-Sensitive Pointer Analysis for Java. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 465–484, 2015.
11. Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective Typestate Verification in the Presence of Aliasing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 133–144, 2006.
12. Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective Typestate Verification in the Presence of Aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2008.
13. Manuel Geffken, Hannes Saffrich, and Peter Thiemann. Precise Interprocedural Side-Effect Analysis. In *International Colloquium on Theoretical Aspects of Computing (ICTAC)*, pages 188–205, 2014.
14. David Hauzar, Jan Kofron, and Pavel Bastecký. Data-Flow Analysis of Programs with Associative Arrays. In *International Workshop on Engineering Safety and Security Systems (ESSS)*, pages 56–70, 2014.
15. Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. Why don't Software Developers use Static Analysis Tools to Find Bugs? In *ICSE*, 2013.
16. Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. Static Data Race Detection for Concurrent Programs with Asynchronous Calls. In *International Symposium on Foundations of Software Engineering (FSE)*, pages 13–22, 2009.
17. Uday P. Khedker, Amitabha Sanyal, and Amey Karkare. Heap Reference Analysis Using Access Graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2007.
18. Shuhei Kimura, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Does return null matter? In *Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 244–253, 2014.
19. P. S. Kochhar, D. Wijedasa, and D. Lo. A large scale study of multiple programming languages and code quality. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016.
20. Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In *ECOOP*, 2018.
21. Akash Lal and Thomas W. Reps. Improving Pushdown System Model Checking. In *International Conference on Computer Aided Verification (CAV)*, pages 343–357, 2006.
22. Akash Lal and Thomas W. Reps. Solving Multiple Dataflow Queries Using WPDSs. In *International Symposium on Static Analysis (SAS)*, pages 93–109, 2008.

23. Akash Lal, Thomas W. Reps, and Gogul Balakrishnan. Extended Weighted Pushdown Systems. In *International Conference on Computer Aided Verification (CAV)*, pages 434–448, 2005.
24. Ondrej Lhoták and Laurie J. Hendren. Context-Sensitive Points-to Analysis: Is it Worth it? In *International Conference on Compiler Construction (CC)*, pages 47–64, 2006.
25. Yue Li, Tian Tan, Anders Møler, and Yannis Smaragdakis. Scalability-First Pointer Analysis with Self-Tuning Context-Sensitivity. In *International Symposium on Foundations of Software Engineering (FSE)*, November 2018.
26. Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, pages 25–33, 2006.
27. V. Benjamin Livshits and Monica S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX Security Symposium*, 2005.
28. Ravi Mangal, Mayur Naik, and Hongseok Yang. A Correspondence Between Two Approaches to Interprocedural Analysis in the Presence of Join. In *European Symposium on Programming (ESOP)*, pages 513–533, 2014.
29. Michael C. Martin, V. Benjamin Livshits, and Monica S. Lam. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. In *OOPSLA*, pages 365–383, 2005.
30. Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping Through Hoops: Why do Java Developers Struggle with Cryptography APIs? In *ICSE*, pages 935–946, 2016.
31. Mangala Gowri Nanda and Saurabh Sinha. Accurate Interprocedural Null-Dereference Analysis for Java. In *ICSE*, pages 133–143, 2009.
32. Thomas W. Reps. Undecidability of Context-Sensitive Data-Independence Analysis. *ACM Transactions on Programming Languages and Systems*, pages 162–186, 2000.
33. Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*, pages 49–61, 1995.
34. Thomas W. Reps, Stefan Schwoon, Somesh Jha, and David Melski. Weighted Pushdown Systems and their Application to Interprocedural Dataflow Analysis. *Science of Computer Programming*, pages 206–263, 2005.
35. H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 1953.
36. Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM (CACM)*, pages 58–66, 2018.
37. Johannes Späth. *Synchronized Pushdown Systems for Pointer and Data-Flow Analysis*. PhD thesis, University of Paderborn, Germany, 2019.
38. Johannes Späth, Karim Ali, and Eric Bodden. IDE^{al}: Efficient and Precise Alias-Aware Dataflow Analysis. In *OOPSLA*, 2017.
39. Johannes Späth, Karim Ali, and Eric Bodden. Context-, Flow- and Field-Sensitive Data-Flow Analysis using Synchronized Pushdown Systems. In *POPL*, 2019.
40. Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *ECOOP*, 2016.
41. Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and Scalable Security Analysis of Web Applications. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 210–225, 2013.
42. Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective Taint Analysis of Web Applications. In *PLDI*, pages 87–97, 2009.
43. Dacong Yan, Guoqing (Harry) Xu, and Atanas Rountev. Demand-Driven Context-Sensitive Alias Analysis for Java. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 155–165, 2011.