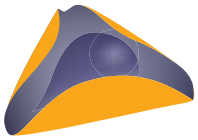

MASTER IN COMPUTER SCIENCE

BString: A String-based Framework to Improve Application Security

Master Thesis

Christian Zürcher

from

Bern BE, Switzerland

Faculty of Science
at the University of Bern

14 February 2022

Prof. Dr. Oscar Nierstrasz

Pascal Gadient

Software Composition Group
Institute for Computer Science

University of Bern, Switzerland

UNIVERSITÄT BERN

UNIVERSITÉ DE NEUCHÂTEL

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Abstract

Code to handle arbitrary user data is complex and likely suffers from one or more security code smells, which may leak sensitive data or enable remote code execution. The usability and security of software tools and frameworks has improved over the last decade, however their improvements usually focus on specific problems and do not offer a comprehensive solution. In fact, there exists no data-centric solution that can adequately prevent personal information from leaking and at the same time sanitize content to prevent remote code execution attacks, although the origins of both threats are very related.

In this work, we present a flexible framework called *BString* that can effectively prevent data leaks and other threats even if developers are inexperienced or unaware of potential security implications when they apply changes to their own code. To achieve that, we extended the String class of a recent *OpenJDK* release, which we could use with most of the top rated Maven libraries for Java, *e.g.*, *Apache HttpClient*, *Log4J*, and *OkHttp*. Using *BString*, we could successfully prevent data leaks, leverage off-memory encryption, or perform taint and data flow analyses even without changing the existing application logic. Therefore, we believe that such a framework could improve application security and should be incorporated into major programming languages.

Contents

1 Introduction

The Java String API does not provide any particular method related to security, *i.e.*, there is no method to validate the assigned data or to prevent data leakage. For example, there exists no embedded facility that can prevent email addresses and passwords stored in text strings from leaking to the console or to a log file. If additional protection is required, developers could, on the one hand, consider using additional checks at critical locations. Unfortunately, the thorough manual implementation of such checks is very error prone and introduces code duplication, which further complicates the problem. On the other hand, there exist static code analysis tools that cannot access problematic run time data, and there exist dynamic analysis tools which are rather limited and usually focus on a single security threat, *e.g.*, they can only perform variable tainting to determine potential data leaks. In other words, there is no comprehensive solution that can prevent sensitive data from leaking *and* remote code execution attacks, although these are among the top three major web application threats in 2021 according to the OWASP project.[1]

In this work, we present a String class that can react depending on the contained value. In consequence, the behavior of an application will adjust depending on the contained String values and their locations. For this purpose, we introduced two additional methods in the *OpenJDK* Java String class implementation, which accept compiled byte code as an input parameter. The provided code will be executed before every read, respectively before every write on the object. For example, if such a String object's value is requested from within a `FileWriter` instance, it can either block the request and raise an exception, grant and

[1] `https://owasp.org/www-project-top-ten/`, accessed on 09-FEB-2022

log the request, or return a safe replacement value that indicates the need for protection. The use of such strings has only a minor average performance impact on accesses of less than 16%, is completely optional and, if not used, it does not break existing code.

We investigate the following two research questions:

RQ$_1$: *What are the restrictions when using an instrumented Java String class with existing code?* We used an instrumented String instance as parameter in the API calls of the thirteen most popular Java libraries in the Maven repository, and we evaluated whether the behavior of the instrumented String was executed. If our code was not executed, we investigated the root cause. We found that only two libraries were not compatible with our framework, *i.e.*, the *Gson* library, which used reflection to access String values, and the *SLF4J* library, which used a not instrumented custom byte buffer implementation for text content.

RQ$_2$: *Can an instrumented String class offer protection against data leaks and remote code execution, and what are the security risks using such a technique?* Using our prototype, we implemented several remediation strategies against the two major threats, data leak and remote code execution, that have been presented in existing literature. Moreover, we briefly summarize common threats that may arise when misusing our prototype. We found that this technique can protect particularly well against data leaks, *i.e.*, it can encrypt data on demand or restrict access for certain classes without the need for changing existing code in the used libraries. Furthermore, the provided interface supports various other security-related tasks, *e.g.*, data verification and validation.

In summary, this work investigates the utility of injecting arbitrary code into String instances to leverage additional functionality within Java applications. Our evaluation with commonly used Java libraries showed promising results for security-related tasks and moreover, we expect that this concept can further be helpful in the domain of, for example, logging and debugging.

The remainder of this thesis is organized as follows. We motivate our approach with a review of existing security measures that we present in chapter 2. We discuss the implementation in chapter 3, and we present the restrictions of our tool in chapter 4. We show how our tool can contribute to security in chapter 5, and how our tool can be used for other purposes in chapter 6. We discuss the related work in chapter 7, and we recap the threats to validity in chapter 8. Finally, we conclude this thesis in chapter 9.

2

Existing Security Measures

In this chapter, we discuss well known security measures that can be used to prevent data leaks or remote code execution attacks. Security measures against data leaks ensure that sensitive data cannot leave a system by making it read-once so that it cannot accidentally leak to unintended places, by making the data unreadable for non-trusted readers, or they trace it through an application in a way they can intercept potential leaks. Security measures against remote code execution attacks ensure that no arbitrary commands are executed within a system by sanitizing values before they are used. In other words, for these two threats security is commonly maintained with a combination of restrictive data types, data encryption, and code analysis techniques.

2.1 Restrictive Data Types

We discuss five different data types and pluggable type systems that can either prevent data leaks or remote code execution.

2.1.1 Alias, Union, and Linear Types

Linear types ensure that their values are used exactly once and therefore, if a value is used at a certain location, it cannot be leaked at another place. Beyond the initial work from J.-Y. Girard that formalizes and discusses the concept of a linear logic in the context of a programming language [18], researchers

adopted the linearity property for arrays [41] and added support for parallelism with the help of a third type they call "observers for variables of a linear type" [30]. Since real world data use rarely behaves linearly Kobayashi relaxed these constraints and proposed to consider the evaluation order using simple data flow information, which they call *quasi-linear* types [26]. In practice, linear types must also support pointer aliasing offered by low-level typed languages like C where multiple pointers that can point to the same data instance; a problem that Smith *et al.* resolved with *Alias Types* for which they track the flow of pointers [37]. Finally, *Union Types* are finite joint linear types that have been proposed by Anderson *et al.* to simplify their use [2]. Nowadays, the resulting implementations can be found in several programming languages, *e.g.*, in Haskell [5] or Rust. In general, such implementations do not leverage the full potential of linear types, *e.g.*, in Haskell linear types can increase safety, but neither increase performance nor reduce memory consumption.

2.1.2 Liquid and Refinement Types

Refinement types support type invariants for existing types to refine them and allow errors that indicate broken system states to be detected at compile-time that can escape traditional type checkers [17, 23]. There exist more advanced approaches such as *Logically Qualified Data Types (Liquid Types)* which, for example, can automatically deduce certain value boundaries precise enough to be useful for error checking [34]. Moreover, there exist implementations such as *Liquid Haskell* that support preconditions and postconditions for value types.[1]

2.1.3 Pluggable Type Systems

Pluggable type systems leverage the idea of using on demand sequentially or simultaneously more than one type system to detect errors that might escape one but not another type system [9]. Two well-known implementations of such a system are *TypePlug* for Smalltalk [19] and the *Checker Framework* for Java [31],[2] which offers various error checkers at compile-time, *e.g.*, for nullness, value tainting, resource misuse such as missing close statements for network sockets, or even string formats [43]. If such a tool detects a type violation, it will raise a type error.

2.2 Data Encryption

Researchers have designed algorithms for the String class to prevent data leaks with a secure workflow for the creation, the clearing, the manipulation, and the comparison of their values [1]. Moreover, methods have been proposed to efficiently store encrypted string values in the cloud [13–15]. Although protected strings have been integrated into common programming languages they were not successful, because their protection must be temporarily lifted every time they are accessed, and their support on different platforms

[1] `https://ucsd-progsys.github.io/liquidhaskell-blog/`, accessed on 09-FEB-2022
[2] `https://checkerframework.org/`, accessed on 09-FEB-2022

is rather limited.[3]

2.3 Code Analysis Techniques

In this subsection, we discuss similar code analysis techniques that have been proposed by researchers and practitioners, *i.e.*, string tainting and reachability search, symbolic execution, string deduplication, and network data monitoring to prevent data leaks.

2.3.1 Tainting

Tainting is a dynamic analysis technique that uses an additional flag for each variable that can be evaluated before each value access. For example, the execution of potentially harmful data can be prevented easily by using a taint flag to indicate non-sanitized data. *Phosphor* is a typical framework for Java taint analyses with a commodity Virtual Machine (VM), *i.e.*, the tool adds additional fields and shadow variables to a Java application's byte code, but does not require any VM or base class library changes [4]. In particular, its class loader can perform these changes on the fly while loading a class. However, after instrumentation the applications have about twice as many lines of code. Alternatively, there exists an implementation that provides full control to the user and requires the manual insertion of tainting code [10]. A major problem is the analysis of the interaction between mixed code, *i.e.*, Java byte-code and C/C++ machine code. To circumvent this problem, Wang *et al.* extended a traditional tainting approach to let it tag every instruction in the native code so that the values can be tracked across language boundaries with which they could find several leaks in the network communication of Android mobile apps [42]. However, such proposals require changes in the application code, which are not always feasible. Therefore, researchers started to extend the language interpreters. Chin *et al.* modified the Java base classes String, StringBuffer, and StringBuilder to let them support tainting [11]. With this technique they could reduce the execution overhead of the taint analysis down to less than 15%. Such taint tracking techniques have also been implemented in other languages like PHP where researchers achieved a similar performance [29, 33]. Xu *et al.* further realized that tainting techniques can be used to mitigate many major attacks such as control-flow hijack, cross-site scripting, shell command and SQL injection, and consequently advocate the use of more complex tainting policies to detect them [44].

2.3.2 Data Flow

Data flow analyses trace data from an origin to a destination, respectively from a source to a sink. Comprehensive analyses with high recursion limits are heavy on resource usage, but rather limited analyses can be well-embedded as plug-ins into IDEs such as Eclipse or IntelliJ in which they can report their findings in distinct views. For example, Livshits *et al.* instrumented byte-code with a Program Query Language (PQL) interface that allows them to specify sources, sinks, and derived methods to

[3] `https://github.com/dotnet/platform-compat/blob/master/docs/DE0001.md`, accessed on 09-FEB-2022

identify relevant data flows [27]. Besides this particular implementation, there exist many more data flow frameworks, *e.g.*, Java String Analyzer (JSA) [16], Soot [39] and its adaptation for Android called FlowDroid [3].

2.4 Discussion

The reviewed literature proposes numerous techniques and measures, however all of them address a very specific problem and cannot be used for a more general purpose. For example, flow analysis tools can report a potential data leak threat, but they still require a manual fix resolve the identified problem. In consequence, a plethora of tools is required to improve the security of an application, which increases the technical debt and limits the ability to have unified security guidelines across different projects. In the increasingly complex environments that developers work nowadays, we believe that we cannot expect such an undertaking from them. Therefore, we advocate a simple, but flexible and reliable tool that offers multiple purposes depending on its configuration. We discuss such a tool in the remainder of this work.

3 Prototype

In this chapter, we start with a motivating example to describe the core idea, before we elaborate on our prototype, *i.e.*, its implementation, the features, build instructions, and the achievable performance.

3.1 Motivating Example

A developer has to implement an application that receives a user password from a web service and then securely stores it within an encrypted database. Moreover, the plain password must never be logged, stored to disk, or leaked through a network socket, and it must not contain any special characters that could enable RCE attacks.

To comply with the requirements, a continuous monitoring of all variables that get directly or indirectly "in touch" with the password value would be required since the password string can be concatenated with multiple other strings, for example, when a random salt value is added which increases resilience against brute-force and rainbow table attacks, before it reaches the destination. Therefore, a developer would currently require at least two different tools to solve this task: i) a dynamic analysis framework that can trace variables during run time configured with certain rules, and ii) a library or manual code that will check the variables for consistency before they are accessed. Besides the high complexity of using multiple tools for this particular task, it is difficult to reuse such code and configuration rules across different projects due to the additional dependencies introduced by the tools.

In contrast, our framework encourages developers to separate traditional application logic from String validation logic that can easily be reused and maintained across different projects. In Listing 1, we show an excerpt of an interface implementation that can protect a password value, which, using our tool *BString*, can be attached to one or more String class instances. In particular, the implementation ensures that no data can leak through `java.net` (network), `java.io` (disk), and `java.system.out` (console) classes (lines two to nine), and at the same time it checks whether the string value contains only safe characters (lines eleven to fourteen) to prevent potential RCE attacks. Moreover, the attached interface implementation of a String instance automatically can be derived to every other String instance that is involved in a shared operation, *e.g.*, when they get "in touch" because of concatenation. In short, the presented code in Listing 1 can offer several benefits: i) it separates security-related concerns from traditional code, ii) it prevents duplicated validation logic scattered across different classes, iii) it reduces the overall project complexity by centralizing code and by making several analysis libraries and frameworks obsolete, iv) it can be reused across different projects, and finally, v) security-related changes are immediately visible when using a versioning system.

```
1  public String applyOnRead(String s) {
2          leakyClasses = "java.net", "java.io", "java.system.out";
3          stackTraceElements = Thread.currentThread().getStackTrace();
4
5          For each fully-qualified class name n in stackTraceElements do {
6                  if(n.startsWithOneOf(leakyClasses)){
7                          throw new LeakException("Data leak identified in class " + n);
8                  }
9          }
10
11         allowedCharacters = "A-Z", "a-z", "0-9", "-", ".", ",";
12         if (!s.containsOnly(allowedCharacters) {
13                 throw new RCEException("The provided text contains unsafe characters.");
14         }
15
16         return s;
17 }
```

Listing 1: Pseudo-code that illustrates the use of *BString* against data leaks and RCE attacks

3.2 Implementation

We only modified the built-in Java classes of the OpenJDK VM to maintain compatibility across different platforms. In particular, we modified besides the Java `String` class three more Java base classes, which are used by the Java implementation when a developer concatenates multiple strings, *e.g.*, by using the + operator: `StringBuilder`, `StringBuffer`, and `AbstractStringBuilder`. In particular, we were interested in intercepting all methods of the String class that internally operate on the String value using a `byte[]` or `this` reference, *i.e.*, `equals(Object)`, `getBytes()`, `charAt(int)`,

`substring(int, int)`, `matches(String regex)`, and `toString()`. We check for each of these public methods whether there is a behavior attributed to the String object and act accordingly.

We define the term "behavior" as the required code that a user must provide to make use of our prototype, *i.e.*, an implementation of the `IStringBehavior` interface.

The intercepting code must follow the convention of the `IStringBehavior` interface shown in Listing 2 that we have added to the package `java.lang`. This interface describes five methods that can be implemented: i) `applyOnCreation(...)` that holds the code that is executed before the initialization of a String instance, ii) `applyOnRead(...)` that holds the code that is executed before a value is leaked, iii) `applyDerivationRule(...)` that specifies when the provided code should be attached to a derived String instance, iv) `recordHistory()` that allows, if enabled, to access each String transformation that occurred in the lifetime of a String, and finally, v) `getDescription()` that returns a textual description of the provided logic. Moreover, we provide the class `StringNotMatchingBehaviorException` to indicate an error state that can be raised by the developer, *e.g.*, if a password is about to get leaked, *etc.*

```
1 public interface IStringBehavior {
2     public String applyOnCreation(String s);
3     public String applyOnRead(String s);
4     public boolean applyDerivationRule(DerivationRule dr);
5     public boolean recordHistory();
6     public String getDescription();
7 }
```

Listing 2: Methods of the interface `IStringBehavior`

Whether a user has to use `applyOnCreation(...)`, `applyOnRead(...)`, or even both methods simultaneously depends on the task. If such a method is not implemented, the string with behavior will act like any regular non-modified string. If such a method is implemented, it must return a string that will be further used throughout the application instead of the original value. Four examples are listed in Listing 3: line two returns the original string and therefore does not alter the behavior, line three returns the string "NewString," line four returns a protected string encrypted by a custom method, and finally, line five throws the run time exception `StringNotMatchingBehaviorException` to block the current execution of the application.

```
1 public String applyOnRead(String s) {
2     return s;
3     return "NewString";
4     return EncryptionMechanism.encrypt(s);
5     throw new StringNotMatchingBehaviorException();
6 }
```

Listing 3: Behavior examples

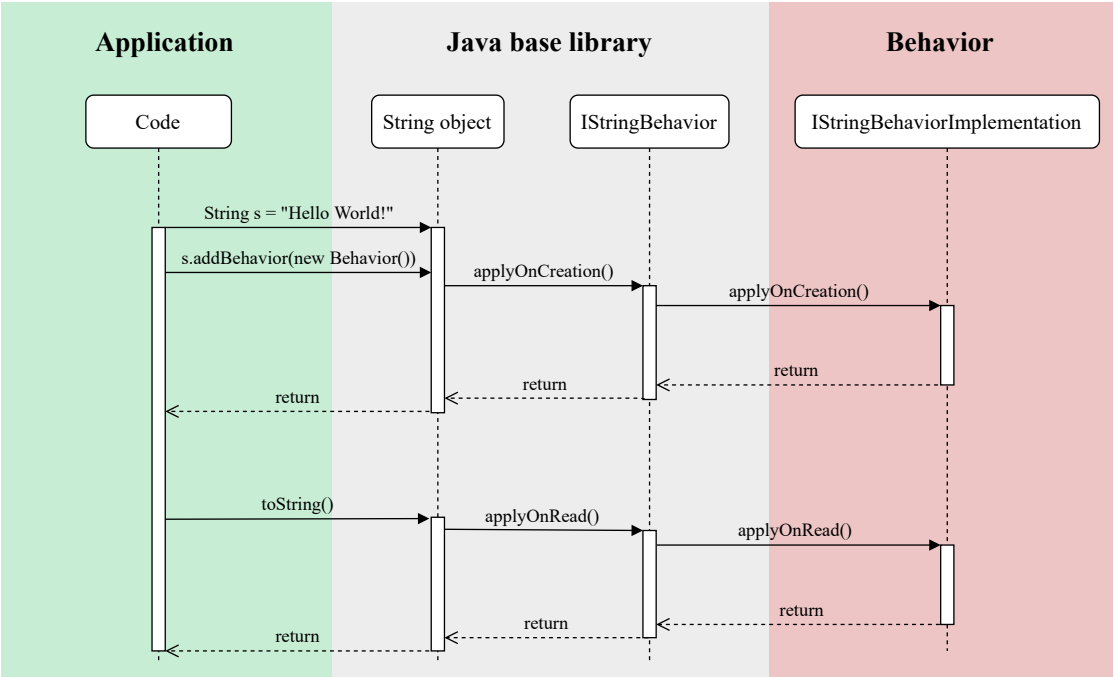

Figure 3.1: Message flow between software components

We illustrate the signalling between the different methods when the `applyOn*(...)` APIs are used in Figure 3.1. First, we create a String object and then we add some behavior to it, *i.e.*, an implementation of the `IStringBehavior` interface. This implementation contains the relevant logic and remains active when the String is later accessed with, for example, the `toString()` method. Besides the manual use of such String objects, we further provide the class `StringBehaviorController` that can programmatically assign a behavior to selected String instances, *e.g.*, inside a specified package, method, or class. This class is particularly important when working with closed-source code, however it only works when the support for modules is disabled in the JDK.

3.3 Features

The implementation adds three major features to the String class, *i.e.*, mutable strings, derivation of string behaviors, and a value history.

3.3.1 Mutable Strings

Strings are immutable in the Java VM and a change in a String value will always return a new, derived String instance. However, we can use the `applyOnCreation(...)` method to set a different value of the derived String instance during its initialization by accessing the internal fields `byte[] value` and

`byte coder`, or we could at any time change the returned value of the string with the help of a behavior, *e.g.*, to prevent data leaks.

We present two examples that illustrate the use and usefulness of a mutated string value:

- Listing 4 shows the code that is required to escape all apostrophes inside a string to prevent SQL injection attacks. Only the resulting escaped string value will be saved into memory.

```
1 public String applyOnCreation(String s) {
2     if(s.contains("'"))
3         return s.replaceAll("'","''");
4     return s;
5 }
```

Listing 4: Initialization behavior to prevent SQL injection attacks

- Listing 5 shows the code that is required to add a prefix to a String of which the prefix even can be altered during run time to, for example, inform users that the application is a debug release that should not be used for production.

```
1 public static String prefix = "DebugRelease:";
2 public String applyOnRead(String s) {
3     return prefix + s;
4 }
```

Listing 5: Read behavior that adds a prefix

3.3.2 Derivation of String Behaviors

A user can precisely control to which derived String instances a behavior is attached with the method `applyDerivationRule(...)`, which takes a `DerivationRule` enumeration as argument. Four different derivation rules are available: i) `COPY` only attaches the behavior to the derived String if it is a copy of the original String, ii) `ADD` only attaches the behavior to the derived String if it is the result of a concatenation from the original String with another String, iii) `DELETE` only attaches the behavior to the derived String if it is the original String with one or more removed characters, and iv) `REPLACE` only attaches the behavior to the derived String if at least one character is altered from the original String. The derivation rules are useful if certain string operations reduce the sensitivity of information, *e.g.*, a credit card number from which numbers were removed, might require less protection since the shortened numbers may not anymore unique.

Listing 6 shows typical examples of the different derivation rules. In line one we copy a string to another string. Therefore, the framework will consider that a COPY operation. In line two we add a character to an existing string, which will be classified as ADD operation. In line three we derive a new string without the first three characters, which will be classified as DELETE operation. Finally, in line four we replace the letter “a” with “b” in a string, which will be classified as REPLACE operation. We distinguish

the REPLACE operation although it could be seen as a mixture of add and delete operations, because it enables a developer to react more accurately on the performed changes.

```
String derivative = original;                    // resulting derivation rule: COPY
String derivative = original + "!";              // resulting derivation rule: ADD
String derivative = original.substring(3);       // resulting derivation rule: DELETE
String derivative = original.replace('a','b');   // resulting derivation rule: REPLACE
```

Listing 6: Typical examples of different derivation rules

We show the use of such derivation rules in Listing 7. During run time, the provided variable `dr` in line four holds the detected string operation by *BString*. For whatever string operation a developer returns true, the behavior will be derived to the resulting String instance. For example, in line five we specify that the behavior must be attached to derived String instances if they are copied or include additional characters, but not in any other case. In other words, if we want to always or never attach a behavior to derived String instances, we can simply return `true` or `false`, respectively.

```
public class BehaviorWithDerivationRule implements IStringBehavior {
    ...
    @Override
    public boolean applyDerivationRule(DerivationRule dr) {
        return (dr.equals(COPY) || dr.equals(ADD));
    }
    ...
}
```

Listing 7: Use of derivation rules

Listing 8 shows the interplay between a behavior and the derivation rules using more complex examples together with the behavior from Listing 7. We initially assign a behavior to the String instance `s1` (line one). The behavior will be derived from `s1` to `s2` (COPY, line two) and `s3` (ADD, line 3) since both derivation rules matched the specified condition in the used behavior. However, all the other strings will remain without any attached behavior, *i.e.*, `s4`, `s5`, `s6`, and both instances inside `arr`.

```
String s1 = new String("Hello World", new BehaviorWithDerivationRule());
String s2 = new String(s1);   // resulting derivation rule: COPY
String s3 = s1 + "!";         // resulting derivation rule: ADD
String s4 = s1.substring(6);  // resulting derivation rule: DELETE
String s5 = s1.toLowerCase(); // resulting derivation rule: REPLACE
String s6 = s1.toUpperCase(); // resulting derivation rule: REPLACE
String[] arr = s1.split(" "); // resulting derivation rule: DELETE
```

Listing 8: Interplay between a behavior and the derivation rules

3.3.3 Value History

BString can record the value of every instrumented String object and generate a tree representation of their lineages. In more detail, the String history node class `SHNode` represents a node of a tree that reflects the

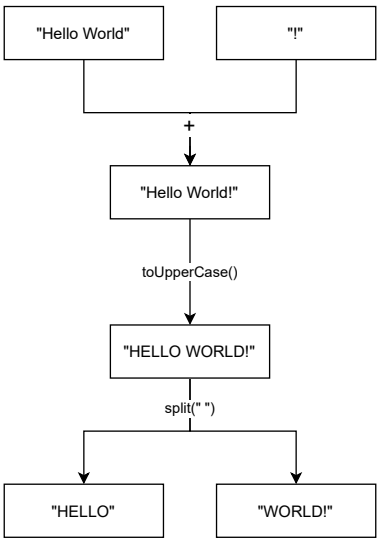

Figure 3.2: The resulting value history tree for Listing 9

tracked string transformations, *e.g.*, when two strings are concatenated the resulting string will contain a tree with at least two nodes, *i.e.*, a parent node that points to one or more child nodes, which refer to the originating strings. To use this feature, the implementation of the method `recordHistory()` must return `true` and then the developer can access the tree through the `SHNode` variable in the behavior interface. With this feature a user can track the changes that have been applied over time to a particular String, and act accordingly.

```
1 String s1 = new String("Hello World", new HistoryRecordBehavior());
2 String s2 = s1 + "!";
3 String s3 = s2.toUpperCase();
4 String[] arr = s3.split(" ");
5 SHNode historyNodeOfS1 = s1.getHistoryNode();
6 SHNode historyNodeOfArr = arr[1].getHistoryNode();
```

Listing 9: Use of the value history feature

Listing 9 illustrates the use of the value history feature. The content of the String `s1` is reused several times before the final result arrives in the String variables in `arr`. The corresponding value history tree is shown in Figure 3.2. First, we perform an ADD operation on “Hello World” and “!”, before we perform a REPLACE operation with `toUpperCase()`. Next, we perform a DELETE operation that splits the string into the two distinct strings “Hello” and “World.” For the original and every derived String instance a new history value node is attached to their existing tree after each operation, which we can then access with the method `String.getHistoryNode()`. We can further navigate the tree by calling the `getParents()`, `getChildren()`, and `getValue()` methods of `SHNode`. Alternatively, the tree can be inspected by any debugger. In Figure 3.3, we show a screenshot of the debugger available in Visual

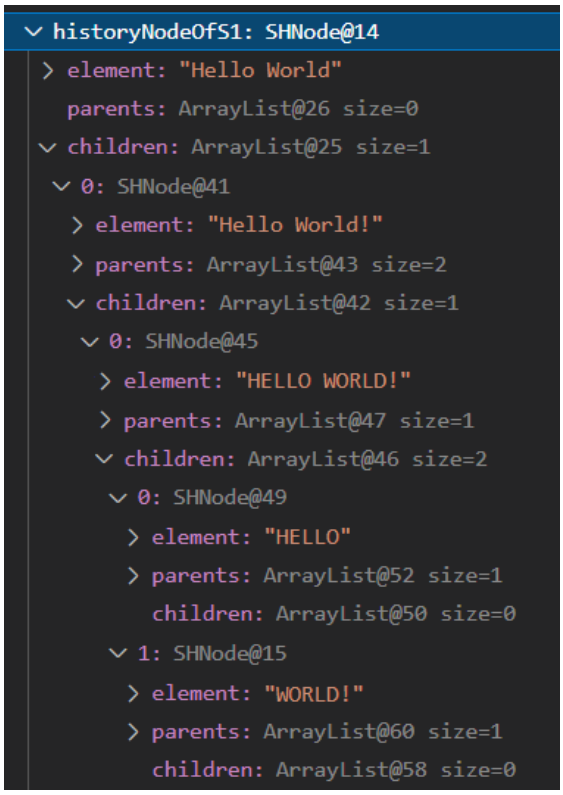

Figure 3.3: Value history tree visualized from a debugger

Studio Code[1] that currently inspects the variable `historyNodeOfS1` at the end of the execution (line six in Listing 9).

3.4 Application Support

BString supports three different kinds of software:

- *Open-source applications.* If the source-code of an application is available, a user can implement the `IStringBehavior` interface to specify the desired behavior and either assign it to individual strings with the method `String.setBehavior(IStringBehavior)` or assign it more generally with the provided class `StringBehaviorController` to, for example, all strings within a specified class. In more detail, the class `StringBehaviorController` supports the methods `addMethodBehavior(...)`, `addClassBehavior(...)`, and `addPackageBehavior(...)`, which all require a string parameter that either define the relevant method (*e.g.*, `some.package.class.Method`), the relevant class (*e.g.*, `some.package.class`), or the relevant package (*e.g.*, `some.package`), and an instance of the behavior that should be applied.

[1] `https://code.visualstudio.com/`, accessed on 09-FEB-2022

This will force all String constructors inside the defined scope to execute the `applyOnCreation()` method of the behavior. In cases where method, class and package behavior conflict, the method behavior is prioritized, then the class behavior and finally the package behavior.

- *Closed-source libraries.* A user can add custom String behavior to compiled libraries. The process is the same as for open-source applications, *i.e.*, a String with behavior must be created and then used as parameter in public library methods.
- *Closed-source applications.* The process to instrument compiled code that is packaged as a runnable `.jar` file or as class files is more complex: the interface implementation must be compiled manually, and then referenced in an `.xml` file that must be parameterized at the start of the VM, which then will load the configuration from disk. An example of such a configuration file is shown in Listing 10 where the root tag `<behaviors>` allows to specify one or more `IStringBehavior` implementations that will be used in the application. Each behavior is enclosed by a `<behavior>` tag that must contain the name of the behavior including the package name if necessary, and where it applies to. We can define each package, class, or method in which the String instances must use the provided behavior within the behavior's `<applyTo>` tag. The use of such a configuration file will force all String constructors within the defined scope to execute the `applyOnCreation(...)` method of the specified behavior. Whenever a conflict arises between different behaviors, package-level behaviors will have the least priority and method-level behaviors will have the highest priority.

```
<?xml version="1.0">
<behaviors>
    <behavior>
        <name>SomeBehaviorClass</name>
        <applyTo>
            <package>name.of.package</package>
            <class>name.of.other.package.Class</class>
            <method>name.of.package.Class.method1</method>
            <method>name.of.package.Class.method2</method>
        </applyTo>
    </behavior>
    <behavior>
        ...
    </behavior>
    ...
</behaviors>
```

Listing 10: Typical configuration file for closed-source application analyses

3.5 Build Instructions

The modified Java classes required to build *BString* are publicly available on GitHub.[2] These classes can be used with the latest OpenJDK release 17, which is an open-source Java platform implementation that is also available on GitHub.[3]

Three steps are required to build and use *BString* implementation: i) the OpenJDK source code must be downloaded from its official GitHub repository, ii) our modified Java classes must be integrated into the corresponding OpenJDK source code folders `./src/java.base/share/classes/java/lang` and `./src/java.base/share/classes/module-info.java`, and finally, iii) the entire VM must be compiled including the Java base library image files, which contain the compiled and condensed Java base classes. Since we are building a JDK 17, we used a clean JDK 16 as boot JDK to run the Java files during the build of the new VM. The last step requires about fifteen minutes on a six-core (12 threads) Intel Core i5-10400F CPU with 16 GB of memory. The exact commands depend on the used operating system:

Microsoft Windows 10 (x64):

1. First, the *cygwin* environment must be installed with at least the packages `autoconf`, `make`, `zip`, `unzip`, and `gcc`.
2. Then *Visual Studio 2017* or higher must be installed including the optional components *Visual C++ core desktop features* and *Visual C++ tools for CMake*. Since some components can interfere with the build process it is highly recommended to use a clean installation in case of any problems.
3. Next, in the cygwin console `bash configure -with-boot-jdk=/cygdrive/path/-to/boot-jdk-16` must be executed in the root folder of the custom VM. For other Visual Studio releases than 2017, the corresponding version must be specified with the parameter `-with-tool-chain-version=XXXX`.
4. Once the environment preparation has finished, the bash command `make images` can be executed to compile the entire JDK.

Ubuntu Linux 18.04.6 (x64):

1. First, the `gcc` package must be installed.
2. Then, the bash command `bash configure -with-boot-jdk=/path/to/boot-jdk-16` must be executed in the root folder of the custom VM.
3. Once that process has finished, the `make images` command must be executed. The compiled JDK will be stored in the folder `./build/[operating-system]-release/images/jdk`.

[2] `https://github.com/jacktraror/papers-behavioral-strings-code`, accessed on 09-FEB-2022
[3] `https://github.com/openjdk/jdk`, accessed on 09-FEB-2022

	initialization	read	derivation
baseline	4 ms	166 ms	11 ms
without behavior	13 ms	171 ms	20 ms
with empty behavior	16 ms	193 ms	22 ms
with derivation rules	16 ms	191 ms	32 ms
with history	59 ms	213 ms	87 ms
with encrypt on init	2 388 ms	9 331 ms	19 ms
encrypt on init and decrypt on read	5 112 ms	7 982 ms	timeout

Table 3.1: String performance evaluation

3.6 Performance

To assess the performance of the implementation, we instantiated one million String objects with random values using seven different configurations and measured the required time: i) a vanilla JDK without any changes in the String class, ii) a custom JDK where we used no behaviors, iii) a custom JDK where we used empty behavior stubs, iv) a custom JDK where we used empty behavior stubs and the behavior derivation feature, v) a custom JDK where we used empty behavior stubs, the behavior derivation feature, and the history feature, vi) a custom JDK where we used a behavior that encrypts the provided value according to the DES algorithm during the initialization, and finally, vii) a custom JDK where we used a behavior that encrypts the provided value during the initialization and decrypts the value when it is read. For each assessment we disabled any output to `System.out` to prevent potential biases.

We present the results in Table 3.1. We can clearly see that a read from a String object consumes much more time compared to its initialization, *i.e.*, a read is between 1.6 and 41.5 times slower. This seems to be caused by inefficient look-ups in the string pool that may involve value conversions between native and interpreted code. In general, encryption and decryption methods are computationally demanding and can prolong the initialization or read tasks by more than three orders of magnitude, *e.g.*, 5,1 s compared to 4 ms. Deriving behavior to other Strings is relatively cheap, *i.e.*, it usually prolongs the initialization task about 50%. Interestingly, it seems that the vanilla Java VM is performing additional background optimizations for String objects when they are reassigned, which is 2.75 times slower than the initialization of the original String instance. Overall, we observed that the use of strings with rather simple behavior has only a minor average performance impact on reads of less than 16%, which is very close to what other researchers have achieved when they added support for value tainting where they measured an overhead of less than 15% [11].

4

Restrictions

In this chapter we investigate the first research question, *i.e.*, *What are the restrictions when using an instrumented Java String class with existing code?* For that purpose we tested our implementation with some of the most popular libraries in the Maven repository.

4.1 Methodology

We searched in the Maven repository for the 13 most popular Java web communication libraries and downloaded for each the most recent version. Next, we set up a Java project for each library that uses our custom JDK and prepared a String instance with a behavior that logs a potential value leak to the console. Then we provided this String instance to the library and checked whether the behavior was executed within the library code. Whenever a behavior did not work as intended, *i.e.*, we received no message in the console, we kept notes and started to investigate the root cause.

4.2 Compatibility

Table 4.1 presents the results of this evaluation. The first column states the project name, the second column the Maven package identifier, the third column lists the tested library version, and finally, the last column indicates whether the library is compatible with our framework, and if not, it shows the reason

Project Name	Package	Version	Compatible?
Apache Commons (IO)	commons-io	2.8.0	✔
Apache Commons (Logging)	commons-logging	1.2	✔
Apache HttpClient	org.apache.httpcomponents	4.5.13	✔
Gson	com.google.code.gson	2.8.5	✖ (reflection)
JavaMail	com.sun.mail	1.6.0	✔
Log4J (core)	org.apache.logging.log4j	2.14.1	✔
Logback (classic)	ch.qos.logback	1.3.0-alpha5	✔
SLF4J	slf4j-simple	2.0.0-alpha1	✖ (custom byte buffer)
SLF4J (API)	org.slf4j	2.0.0-alpha1	✔
Spring	org.springframework	5.3.6	✔
Square Okhttp	com.squareup.okhttp3	5.0.0-alpha.2	✔
Square Okio	com.squareup.okio	2.10.0	✔
Square Retrofit	com.squareup.retrofit2	2.9.0	✔

Table 4.1: Evaluation of popular Java libraries

why not. We can see that only two of thirteen libraries are incompatible with our framework, *i.e.*, Gson and SLF4J. Gson uses reflection to access String data internally, which is not supported by our framework. SLF4J uses custom byte buffers that are not instrumented in our framework.

4.3 Limitations

We discuss the observed limitations of our prototype in more detail, and how these shortcomings could be mitigated in future work, *e.g.*, by extending the prototype or the Java environment.

4.3.1 Native Code

Our prototype relies on the Java class system and therefore cannot track strings that leave the object boundary and are forwarded to native code, which can only work with primitive data types. However, a string with behavior might still detect whether it is accessed in a native method and take action before it is transformed to a primitive string type. Conversely, a user can add behavior, *i.e.*, use the method `addBehavior(...)` to attach an implementation of `IStringBehavior` to a new String that is created within a native method. This limitation could be removed entirely, however this demands many changes. On the one hand, the original C and C++ languages only know a string in the form of a character array, which is a primitive type that is not extensible. If behavior features are required, they must be improved to support such mechanisms. On the other hand, the C++ boost library offers a string class that could be adapted, *i.e.*, to *BString*. Nevertheless, this requires that all native C++ code is recompiled and uses that particular class. In the end, the required native behavior interfaces and their implementations would not be interoperable with our Java implementation.

4.3.2 Value Conversion

Value conversions are not supported without additional support in the relevant classes, *e.g.*, `ByteArray`, *etc.* In other words, classes that do not use the String class to work with text and rather prefer primitive collections such as `char` arrays, `byte` streams, *etc.* cannot benefit from our framework and the attached behavior will be lost when the value conversion is performed. To be clear: we can detect a value conversion in a behavior, but the behavior will be lost after it. This limitation could be removed if additional classes and primitive types would support behaviors, *e.g.*, the `Object` class or the `char` type. However, a generic implementation for `Object` would require type-specific code, which may be difficult to maintain, and primitive types would become more complex and thus slower.

4.3.3 Reflection

In Java, the reflection mechanisms eventually rely on native code to interact with the system that can bypass our behavior. In other words, a behavior will be lost if a String instance is constructed using reflective methods. Therefore, our framework does not entirely support applications that use reflective methods although it is possible to detect an access from a reflection class in a behavior to act accordingly. This limitation could be removed if the native Java VM code that is responsible for the reflection feature would support behaviors.

4.3.4 Concurrency

Our framework is currently not thread-safe. This limitation could be removed by improving the existing code of *BString* to leverage concurrency features, *e.g.*, using concurrent locks to safeguard the offered methods. However, such a change would decrease the performance since string operations would then first need to acquire a lock before any further operation, which is very expensive.

4.3.5 Scope

BString only instruments commonly used methods in the String-related classes `String`, `StringBuffer`, and `StringBuilder`. There is currently no support for the `Object` class or wrapper classes like `Integer`, `Float`, or `Boolean`. Moreover, if enabled, the module system introduced in Java 9 prevents the use of our framework for closed-source projects since such code will not reside in the same module. In order to use our framework for such analyses, the module system must first be disabled or bypassed. This limitation could be removed by additional changes to the Java VM, and by adding support for behaviors in `Object` instances. However, the latter change would massively increase the complexity for such behaviors.

5 Security Gains

In this chapter, we explore the second research question, *i.e.*, *Can an instrumented String class offer protection against data leaks and remote code execution, and what are the security risks using such a technique?* For that purpose, we implemented well-known security measures to prevent data leaks and remote code execution attacks that we have presented in chapter 2. In particular, we show how a developer can use our framework to implement type systems, encrypt values, and perform code analyses. Finally, we reason about potential threats that could arise when using our prototype.

5.1 Data Type Emulation

Two major type systems that are used in practice are linear types and liquid or refinement types. Whereas a linear type can be accessed once at most, refinement types do not have such a limit. However, they allow value constraints, *e.g.*, the corresponding value must be shorter than five characters or only contain letters.

5.1.1 Security Gain

Such type systems can be used to prevent remote code execution by specifying value patterns that must not occur (refinement types), or to prevent data leakage and to improve performance (linear types), because a value can be accessed at most once and immediately after the access it can be safely deleted.

5.1.2 Implementation

The code snippet in Listing 11 shows a String behavior that imitates a linear type, *i.e.*, a boolean flag is checked (line two) to ensure the returned value has not been accessed before in which case the application will proceed as expected. However, if it already has been accessed before, the check will fail and thus raise a `LinearTypeException` (line six). The code snippet in Listing 12 shows a String behavior that imitates a refinement type, *i.e.*, the value is checked whether it meets the requirements (line two) when it is created. If the value does not meet the requirements, the check will fail and thus raise a `RefinementTypeException` (line three).

```
1 public String applyOnRead(String s) {
2     if (!this.hasBeenRead) {
3         this.hasBeenRead = true;
4         return s;
5     } else {
6         throw new LinearTypeException();
7     }
8 }
```

Listing 11: Code that simulates a typical linear type behavior

```
1 public String applyOnCreation(String s) {
2     if (s.length < 5) {
3         throw new RefinementTypeException();
4     }
5     return s;
6 }
```

Listing 12: Code that simulates a typical refinement type behavior

5.2 In-memory Encryption

String data can be securely encrypted using a robust encryption algorithm together with a complex password. Such data remains encrypted in the memory and is only decrypted when used.

5.2.1 Security Gain

The in-memory encryption of data values increases the difficulty for adversaries to understand and exfiltrate sensitive data from collected memory dumps.

5.2.2 Implementation

A String value is encrypted in the `applyOnCreation(...)` method and decrypted in the `applyOnRead(...)` method. The simplified code of such a behavior implementation is shown in Listing 13. The behavior will encrypt the original string and store the encrypted value in memory instead, when a

string is created or the behavior attached to an existing string (line four). If the string is requested, the `applyOnRead(...)` method is called and the string is decrypted for the requester (line 9), but it still remains encrypted in memory. The behavior will further attach itself to all derived String instances (lines thirteen to fifteen), *e.g.*, a substring will also have the identical behavior attached and hence be encrypted in-memory. The entire code of a working implementation can be found in the appendix in Listing 22.

```
1  public class InMemoryEncryptionBehavior implements IStringBehavior {
2     @Override
3     public String applyOnCreation(String original) {
4        return encrypt(original);
5     }
6
7     @Override
8     public String applyOnRead(String encryptedStringInMemory) {
9        return decrypt(encryptedStringInMemory);
10    }
11
12    @Override
13    public boolean applyDerivationRule(DerivationRule dr) {
14         return true;
15     }
16 }
```

Listing 13: A simplified behavior that illustrates the use of *BString* for in-memory encryption

5.3 Off-memory Encryption

Off-memory encrypted data remains as plain text in the memory and is only encrypted when it leaves the memory.

5.3.1 Security Gain

The off-memory encryption of data values ensures that adversaries cannot understand sensitive data that leaves a system, *e.g.*, through a network socket. Developers do not need to encrypt data manually, instead they can attach behavior to sensitive content that will automatically encrypt itself in a transparent process.

5.3.2 Implementation

Contrary to the in-memory encryption, the `applyOnCreation(...)` is not needed and the value encryption in the `applyOnRead(...)` method must only take place in certain execution contexts, *e.g.*, where a value is stored to disk or transmitted through a network socket. A strong but less efficient encryption technique like a symmetric AES-256 algorithm should be preferred since a secure algorithm prevents an adversary from recovering the decryption key. Therefore, the receivers of the encrypted data must know the decryption key. Listing 14 shows the simplified code for such an implementation. It is

not required to implement the `applyOnCreation(...)` method, because the value must only be encrypted when it is read in classes from network and file input/output packages. Two packages related to network and disk storage have been selected to enable the encryption: `java.net` and `java.io` (line two). The current execution context that contains the used packages is determined with the stack trace (line six). If a package name matches one of those that have been previously selected (lines seven to nine), the behavior will automatically encrypt the value before it is returned (line ten). Since it is reasonable to apply this behavior to every derived string for protecting the data, the `applyDerivationRule(...)` method always returns `true` (line nineteen) without checking for a specific `DerivationRule`. The entire working implementation can be found in appendix Listing 23.

```
1  public class OffMemoryEncryptionBehavior implements IStringBehavior {
2     private String[] ioPackages = { "java.net", "java.io" };
3
4     @Override
5     public String applyOnRead(String s) {
6        StackTraceElement[] contexts = Thread.currentThread().getStackTrace();
7        for(int i = 1; i < contexts.length; i++) {
8           for(String ioPackageName: ioPackages) {
9              if(contexts[i].getClassName().startsWith(ioPackageName)) {
10                return encrypt(s);
11             }
12          }
13       }
14       return s;
15    }
16
17    @Override
18    public boolean applyDerivationRule(DerivationRule dr) {
19         return true;
20     }
```

Listing 14: A simplified behavior that illustrates the use of *BString* for off-memory encryption

5.4 Taint Analysis

Taint checking or tainting is the concept of adding a marker to a specific variable that indicates its trustworthiness, *i.e.*, either the origin of the value is safe or unsafe. For example, a value received from an insecure network socket should be tainted as unsafe, because an adversary might have altered its value to exploit the application.

5.4.1 Security Gain

Properly tainted values can prevent arbitrary code execution or data leaks. However, it is important to accurately specify the contexts, *i.e.*, packages, classes, or methods for which unsafe tainted data represents a security threat.

5.4.2 Implementation

Listing 15 shows a typical implementation that provides taint support for a String variable. The use of a boolean marker is optional, because we can selectively assign this behavior to String instances that must be tainted. However, a taint variable can be introduced, if desired, to support use cases that require the `StringController` class. The String array (line two) denotes the packages in which the particular String instance must not be read, *i.e.*, network and input/output-related classes. The current execution context is retrieved from the stack trace (line five), and the package names are validated for each element in it (lines seven and eight). If an unauthorized package is found, a run time exception will be thrown (line nine). Otherwise, the application will proceed as expected (line thirteen). Since the tainting behavior should "taint" every derived string, we just return true and do not check for a specific `DerivationRule` (line seventeen). Please note that the caching of stack traces is not a viable option with the *existing* Java mechanisms, because the returned stack trace instance is different for every look up and therefore a manual comparison, which we implemented, is required anyway.

```
1  private String[] unauthorizedPackages = {"java.net", "java.io"};
2  private ArrayList<String> list = Arrays.asList(unauthorizedPackages);
3
4  public String applyOnRead(String s) {
5      StackTraceElement[] contexts = Thread.currentThread().getStackTrace();
6      for (int i = 1; i < contexts.length; i++) {
7          for (String packageName: list) {
8              if (contexts[i].getClassName().startsWith(packageName)) {
9                  throw new TaintException();
10             }
11         }
12     }
13     return s;
14 }
15
16 public boolean applyDerivationRule(DerivationRule dr) {
17     return true;
18 }
```

Listing 15: A behavior that simulates taint checking

Moreover, a behavior can behave responsibly for different kinds of threats as illustrated in Listing 16. For example, it could ignore them (lines four and five), log the use of a tainted value to the console for minor threats (lines six to eight), or interrupt the further code execution for more major threats by throwing a `TaintException` (lines nine and ten). Further possibilities could involve the manipulation of the String value before it is returned, or the termination of the entire application.

```
public String applyOnRead(String s) {
    ...
    switch(taintAction)  {
        case NO_ACTION:
            return s;
        case LOG:
            logLeak(stElements[i].getClassName(), stElements, i);
            return s;
        case BLOCK:
            throw new TaintException("This String must not be used in this package!");
    }
    ...
}
```

Listing 16: Behavior extension to support different severity levels

5.5 Data Flow Analysis

Whereas taint analyses are usually rather limited in the scope and the level of automation, data flow analyses are much more sophisticated and can trace data throughout the system from a source that provides sensitive data to a sink that can leak sensitive data. A data flow analysis typically requires a list of relevant data sources and sinks to report the desired traces.

5.5.1 Security Gains

Data flow analyses can prevent arbitrary code execution or data leaks. However, it is important to accurately specify the contexts, *i.e.*, data sources and sinks for which an exchange of data represents a security threat, *e.g.*, a password field as data source and the console output as disallowed data sink. Using the history feature of our implementation, the originating method and class where a String value was instantiated is always known, every access can be tracked and, if necessary, prevented for selected components of the application.

5.5.2 Implementation

The analysis of data flows is very similar to taint checking, and thus it is straightforward to reuse that code as shown in Listing 17. In essence, the behavior must be automatically attached to relevant String sources by using the `StringController` class, and every relevant sink must be added to the list of unauthorized contexts (lines one and two). In addition, access to the origin and every subsequent state is required, which can be enabled using the String history feature (lines twenty to twenty-two). The use of history nodes has already been discussed in subsection 3.3.3.

```
private String[] unauthorizedPackages = {"java.net", "java.io"};
private ArrayList<String> list = Arrays.asList(unauthorizedPackages);

public String applyOnRead(String s) {
    StackTraceElement[] contexts = Thread.currentThread().getStackTrace();
    for (int i = 1; i < contexts.length; i++) {
        for (String packageName: list) {
            if (contexts[i].getClassName().startsWith(packageName)) {
                throw new TaintException();
            }
        }
    }
    return s;
}

public boolean applyDerivationRule(DerivationRule dr) {
    return true;
}

public boolean recordHistory() {
    return true;
}
```

Listing 17: A behavior that enables data flow analyses

5.6 Discussion

In this section we discuss potential threats and summarize the security gains.

5.6.1 Potential Threats

In this subsection, we briefly mention threats that can arise when using our prototype, how they present a security risk, and how they could be mitigated.

5.6.1.1 String or Application Hijacking

We call string hijacking the attack that allows an adversary to collect the content of string variables. If adversaries gain access to the behavior files of an application, they could induce malicious behavior that could, for example, send the value of an instrumented string over the internet to any recipient. Consequently, sensitive information may leak. Even worse, malicious behavior classes would also allow to hijack an entire application, *i.e.*, adversaries could execute their own code in the context of the Java VM, *e.g.*, to download and execute arbitrary malicious code from the internet to take over the computer that runs the application and eventually the corporate network. This threat can only be mitigated by protecting the behavior files from unauthorized accesses and instituting regular code reviews.

5.6.1.2 Developer Confusion

A developer who encounters unexpected behavior might become confused and start to debug the application, *e.g.*, when the code does not function as expected. Therefore, the productivity of such a developer will decrease. This threat cannot entirely be mitigated, however a developer who has to work on projects that rely on behavior classes should be made familiar with the mechanism before writing any code.

5.6.2 Summary

We elaborated five different measures to prevent data leaks and remote code execution that would require several distinct tools, and showed that our proposed framework can provide initial support for all of them. Moreover, different functionality could be combined to provide new solutions to common problems. For example, a developer could create a behavior that validates whether a variable access originates from a network socket class and block the subsequent storage into a database, or transparently encrypt sensitive data to prevent a potential data leak.

6

Further Use

Our prototype is built around security, however it can be used for other purposes.

6.1 Programming Paradigms

Two popular programming paradigms are *Design by Contract (DbC)* and *Aspect-oriented Programming (AoP)*. DbC is somewhat similar to the concept of refinement types, but contrary to that any value type is supported. DbC further allows the declaration of preconditions, postconditions, and invariants for which it ensures that they hold during execution time. AoP serves a different purpose and enables a developer to inject additional code at arbitrary locations, *e.g.*, before the business logic with `@Before` and for specific methods within a class with `@Pointcut`. In the context of a String value both of these paradigms can be simulated with our framework.

6.1.1 Motivation

DbC improves software reliability by constraining value ranges, which can effectively prevent certain error states and it fosters formal methods to validate such logic. AoP allows developers to reuse code in a flexible manner.

6.1.2 Implementation

Listing 18 shows the implementation of a DbC contract that has been specified in the lines one to three. Preconditions can be implemented in the `applyOnCreation(...)` method (line 6), and postcondition checks can be implemented in the `applyOnRead(...)` method (line 15). Invariants must be specified in both methods (see lines eight and seventeen). Listing 19 mimics the ability of AoP to inject arbitrary code before receiving the value of a String variable. In this example, line five prints a message to the system console whenever the variable value is accessed.

```
1  // @pre s != null
2  // @post !s.isEmpty()
3  // @inv s.startsWith("http://")
4
5  public String applyOnCreation(String s) {
6      if (s == null) {
7         throw new DbCException();
8      } else if (!s.startsWith.equals("http://")) {
9          throw new DbCException();
10     }
11     return s;
12 }
13
14 public String applyOnRead(String s) {
15     if (s.isEmpty()) {
16        throw new DbCException();
17     } else if (!s.startsWith.equals("http://")) {
18         throw new DbCException();
19     }
20     return s;
21 }
```

Listing 18: Code that simulates a typical DbC behavior

```
1  public String applyOnRead(String s) {
2      String style = "dd/MM/yyyy HH:mm:ss";
3      SimpleDateFormat formatter = new SimpleDateFormat(style);
4      String currentDate = formatter.format(new Date());
5      System.out.println("Accessed at: " + currentDate);
6      return s;
7  }
```

Listing 19: Code that simulates a typical AoP behavior

6.2 Profiling

Profiling is used to detect inefficiencies in code. Such analyses require usage data that can be retrieved with our framework.

6.2.1 Motivation

Long strings that are involved in complex String operations may degrade system performance, because a String is immutable and must be recreated if it is changed. Therefore, it may be reasonable to gather accurate knowledge about the length of the used string values to apply further system optimizations, *e.g.*, the use of `StringBuffer` instances whenever long strings are modified.

6.2.2 Implementation

The behavior presented in Listing 20 can log the lengths of the used strings. That is, the method `applyOnCreation(...)` collects the current context (line four) and stores the length and the context to the log file (line five) before it proceeds with the normal execution (line six). The resulting data in the log file will reveal the string lengths that are used in a specific class, package, or application.

```
1 public class CollectStringLengthBehavior implements IStringBehavior {
2    @Override
3    public String applyOnCreation(String s) {
4       StackTraceElement[] context = Thread.currentThread().getStackTrace();
5       StringLengthLog.addCreated(s.length(), context);
6       return s;
7    }
8 }
```

Listing 20: A behavior that can report string lengths

6.3 E-Mail Notifications

Emails are convenient for notifications, because they represent commonly used communication channels and can be targeted to a particular person or group, *e.g.*, lead developers or security engineers.

6.3.1 Motivation

If a behavior detects a particular problem, an email can be sent to the relevant people that timely informs them about the issue in their system. An example of such an email is shown in Figure 6.1.

6.3.2 Implementation

Whenever a problem is detected in a behavior, a custom email send routine can be called within the String instance. For less urgent problems it may be reasonable to collect the incidents and report them periodically. However, its user must be aware of the implications: if implemented incorrectly, for frequent problems thousands of emails could be sent in a very short time. Listing 21 shows the simplified implementation of such a behavior. First, sensitive packages are defined (lines two and three) and the current context is evaluated (lines seven to ten, and lines seventeen and eighteen). An error is reported immediately by email

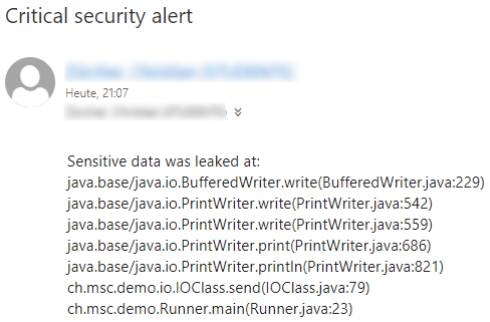

Figure 6.1: Example email notification for a critical problem that has been sent from a behavior

(lines eleven to thirteen), whereas a warning is queued and sent periodically (lines nineteen to twenty-one). The entire working implementation can be found in the appendix in Listing 24 and in Listing 25.

```
1  public class SendMailBehavior implements IStringBehavior {
2     private String[] criticalPackages = { "java.net" };
3     private String[] warningPackages = { "java.io" };
4
5     @Override
6     public String applyOnRead(String s) {
7        StackTraceElement[] contexts = Thread.currentThread().getStackTrace();
8        for(int i = 1; i < contexts.length; i++) {
9           for(String errorPackageName: criticalPackages) {
10             if(contexts[i].getClassName().startsWith(errorPackageName)) {
11                MailAgent.sendError(
12                    "Sensitive data was leaked at:\n"
13                    + stackTraceToString(contexts)
14                );
15             }
16          }
17          for(String warningPackageName: warningPackages) {
18             if(contexts[i].getClassName().startsWith(warningPackageName)) {
19                MailAgent.queueWarning(
20                    "Data was stored to disk at:\n"
21                    + stackTraceToString(contexts)
22                );
23             }
24          }
25       }
26       return s;
27    }
28 }
```

Listing 21: A behavior that can report issues by email

7 Related Work

We discuss work which is related, but cannot easily be replicated with our tool for various reasons, or is out of scope for this work.

7.1 Language Enhancements

In general, a language enhancement refines the existing syntax or behavior to improve a particular scenario, and its use is optional to maintain compatibility with existing code. Examples are language extensions, traits, specialized classes, and island grammars.

7.1.1 Language Extensions

A language extension introduces new language constructs by providing keywords and leveraging the existing type system to support, for instance, polytypic functions such as Java's generics [8, 22]. Another typical use case is providing tool support for embedded languages such as SQL within Java code by using attribute grammars [24, 40]. Our framework cannot replicate language extensions, because such extensions change the language specification, but we have to obey them.

7.1.2 Sealed Classes

A sealed class is a class that restricts the classes that may extend it [6] and many popular programming languages support this feature, *e.g.*, Java[1] and C#.[2] Besides sealed classes, there exist sealed interfaces, which introduce similar constraints for their implementors. A sealed type or interface supports any type or interface, whereas we focus solely on the String class.

7.1.3 Specialized Classes

A specialized class reifies existing classes, *e.g.*, by using meta objects, mutable or protected strings, *etc.* Meta objects were used to expand logic, *e.g.*, meta classes have been added to the Java base library [38] and to the String class [20] to offer tainting functionality. Beyond the support for tainting, researchers identified a problem with existing String implementations, which either favor static read-only data that never changes, or dynamic write-only data that always changes. According to them, many workloads in existing applications show a balanced behavior, which requires occasional reads and writes. Therefore, they proposed a *MutableString*, which combines the efficiency of String with the changability of the StringBuffer class [7]. Since specialized classes usually introduce entirely new data types, we cannot fully replicate their features. Moreover, our implementation was not designed to increase efficiency, but to add configurable security when using a String instance.

7.1.4 Traits

A trait describes a set of methods, which can be used to extend the utility of a class [12]. In other words, classes with a specific trait can leverage the corresponding methods. This concept has been widely adapted by industry, *e.g.*, Java 6 supports default methods that share some properties of traits, and by academia, *e.g.*, Pharo Smalltalk natively supports them [35]. Traits work, contrary to our approach, beyond class boundaries. Therefore, we cannot replicate this particular functionality.

7.2 Island Grammars

An island grammar is a simplified grammar, which can only describe a subset of the underlaying language syntax. Therefore, it is mainly used to parse and extract certain data from rather complex languages. Researchers used such grammars with success on strings to reason about embedded languages, or fragments that need further processing [28]. An island grammar could be used within a behavior implementation, however we did not further explore this domain.

[1] `https://docs.oracle.com/en/java/javase/15/language/sealed-classes-and-interfaces.html`, accessed on 09-FEB-2022

[2] `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/sealed`, accessed on 09-FEB-2022

7.3 Symbolic Execution

Symbolic code execution is the reasoning about input values and their impact on different code paths, which has applications in model checking. Such tools generate a symbolic execution tree before they apply constraint solving algorithms to identify input sequences, which can lead to error reports. A typical example is "Symbolic PathFinder" from Pasareanu *et al.* that supports the automated test case generation and error detection for compiled Java applications [32]. In particular, it supports constraint solving algorithms for mixed numeric and string values. Since symbolic reasoning involves knowledge about application logic, we cannot offer this feature with our framework.

7.4 Deduplication

In Java and other languages strings are immutable, *e.g.*, to simplify caching and reuse, or to ensure thread safety, integrity, and security. However, researchers have found room for improvement in existing implementations. Kawachiya *et al.* delayed literal instantiations and allowed the garbage collector to erradicate duplicated strings [25]. On top of that, Horie *et al.* developed a tool, which can detect string duplicates across different JVMs by using a read-only memory mapped file that contains the string value mappings [21]. They achieved a 27% reduction of allocated string data and a 9% reduction for char arrays on the Apache DayTrader benchmark suite with only little impact on real-world applications. Deduplication could be implemented using our framework, however the potential overhead would most probably be much higher than the actual gains, because our framework operates on higher levels of abstraction compared to the existing or proposed implementations that use native code.

7.5 Network Data Monitoring

Instead of selectively track data within apps, a more comprehensive approach is to monitor all data sent from or to a specific application through network sockets. Sekar *et al.* proposed a technique that is conceptually similar to *Intrusion Detection and Prevention (IDP)* appliances as it searches for patterns in the transmitted data, which indicate malicious intentions [36]. Contrary to IDP devices, the upstream or downstream processors must not break the used encryption since an application usually terminates the secure connection and therefore has access to the clear text. The analysis of String-related network traffic is out of scope for this work.

8

Threats to Validity

In this section we discuss the threats to validity that might affected our results.

The generalization of our implementation might be limited. Our framework currently only offers support for String-related classes and we do not genuinely know whether such functionality can be provided for other programming languages or classes since their implementation may differ. This is an inherent threat, because we cannot guarantee that the proposed approach can be generalized beyond what we have presented. To reduce the impact of this threat, we applied only few changes to the Java classes to avoid unnecessary dependencies and side-effects.

We introduced arbitrary decisions during the implementation and evaluation of our work. We selected the String-related classes in our work, because they are used very frequently and can hold credentials unlike numeric types that can only hold numbers. However, other value types may still be valuable for some developers, and we expect that an instrumentation of `Object` would allow more flexibility. For the evaluation, we randomly chose top listed libraries from the Maven repository, which may not represent well the libraries used in practice. We tried limit the impact of this threat by selecting popular classes and libraries that are very likely representative of the current development practices.

The applicability of our results might be limited. We did not test our framework in large-scale and distributed applications and thus we cannot predict the performance and reliability for such applications. Thread-safety is not yet provided. However, we expect that a well-defined scope can effectively reduce the

performance overhead.

The literature survey could lack important publications. We searched for literature related to the String class, types, and code analysis. However, we might have missed important literature that would contribute to our work. We tried to overcome this threat by iteratively searching in the citations and the cited papers for relevant works.

The authors have carried the work themselves. Finally, the fact that the literature survey and the evaluation is performed by the authors is a threat to construct validity through potential bias in experimenter expectancy.

9 Conclusion

In this work, we modified the String class of a recent OpenJDK release to enable the execution of arbitrary Java code whenever a value is assigned to, or read from a String object. This new interface enables the adoption of several existing security concepts that have been proposed by numerous researchers such as linear and refinement types, or taint and data flow analyses. Moreover, it supports more complex use cases that are hardly possible without such a framework, *e.g.*, transparent in-memory or off-memory data encryption, email notifications, *etc.* Since the implemented behaviors consist of regular Java code, they can be easily integrated into existing development processes and even used beyond a single project. Therefore, such an API would greatly enhance the capabilities of existing Java VMs, and the corresponding behaviors could increase application security at a reasonable cost. We conclude that the proposed framework provides obvious benefits and should be adapted to other languages that use a particular entity to represent text strings.

Bibliography

[1] K. Amrichová and T. Mézešová. A secure String class compliant with PCI DSS. In *Proceedings of the Third Central European Cybersecurity Conference*, pages 1–5, 2019.

[2] C. Anderson, F. Barbanera, and M. Dezani-Ciancaglini. Alias and union types for delegation. *Ann. Math., Comput. & Teleinformatics*, 1(1), 2003.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[4] J. Bell and G. Kaiser. Phosphor: Illuminating dynamic data flow in commodity JVMs. *ACM Sigplan Notices*, 49(10):83–101, 2014.

[5] J.-P. Bernardy, M. Boespflug, R. R. Newton, S. Peyton Jones, and A. Spiwack. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–29, 2017.

[6] M. Biberstein, V. C. Sreedhar, and A. Zaks. A case for sealing classes in Java. In *Israeli Workshop on Programming Languages & Development Environments*. Citeseer, 2002.

[7] P. Boldi and S. Vigna. Mutable strings in Java: design, implementation and lightweight text-search algorithms. *Science of Computer Programming*, 54(1):3–23, 2005. Principles and Practice of Prgramming in Java (PPPJ 2003).

[8] G. Bracha. Generics in the Java programming language, 2004.

[9] G. Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.

[10] B. Chess and J. West. Dynamic taint propagation: Finding vulnerabilities without attacking. *Information Security Technical Report*, 13(1):33–39, 2008.

[11] E. Chin and D. Wagner. Efficient character-level taint tracking for Java. In *Proceedings of the 2009 ACM workshop on Secure web services*, pages 3–12, 2009.

[12] G. Curry, L. Baer, D. Lipkie, and B. Lee. Traits: An approach to multiple-inheritance subclassing. In *Proceedings of the SIGOA conference on Office information systems*, pages 1–9, 1982.

[13] G. Fahrnberger. Computing on encrypted character strings in clouds. In *International Conference on Distributed Computing and Internet Technology*, pages 244–254. Springer, 2013.

[14] G. Fahrnberger. A second view on SecureString 2.0. In *International Conference on Distributed Computing and Internet Technology*, pages 239–250. Springer, 2014.

[15] G. Fahrnberger and K. Heneis. SecureString 3.0. In *International Conference on Distributed Computing and Internet Technology*, pages 331–334. Springer, 2015.

[16] A. Feldthaus and A. Møller. The big manual for the Java String Analyzer. *Department of Computer Science, Aarhus University*, 2009.

[17] T. Freeman and F. Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 268–277, 1991.

[18] J.-Y. Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

[19] N. Haldiman, M. Denker, and O. Nierstrasz. Practical, pluggable types. In *Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007*, pages 183–204, 2007.

[20] W. G. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 175–185, 2006.

[21] M. Horie, K. Ogata, K. Kawachiya, and T. Onodera. String deduplication for Java-based middleware in virtualized environments. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, page 177–188, New York, NY, USA, 2014. Association for Computing Machinery.

[22] P. Jansson and J. Jeuring. PolyP—a polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 470–482, 1997.

[23] R. Jhala and N. Vazou. Refinement types: A tutorial. *ArXiv*, abs/2010.07763, 2020.

[24] T. Kaminski, L. Kramer, T. Carlson, and E. Van Wyk. Reliable and automatic composition of language extensions to C: the ableC extensible language framework. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.

[25] K. Kawachiya, K. Ogata, and T. Onodera. Analysis and reduction of memory inefficiencies in Java strings. *SIGPLAN Not.*, 43(10):385–402, Oct. 2008.

[26] N. Kobayashi. Quasi-linear types. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 29–42, 1999.

[27] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX security symposium*, volume 14, pages 18–18, 2005.

[28] L. Moonen. Generating robust parsers using island grammars. In *Proceedings eighth working conference on reverse engineering*, pages 13–22. IEEE, 2001.

[29] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *IFIP International Information Security Conference*, pages 295–307. Springer, 2005.

[30] M. Odersky. Observers for linear types. In *European Symposium on Programming*, pages 390–407. Springer, 1992.

[31] M. M. Papi, M. Ali, T. L. Correa Jr, J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212, 2008.

[32] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, 2013.

[33] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *International Workshop on Recent Advances in Intrusion Detection*, pages 124–145. Springer, 2005.

[34] P. M. Rondon. Liquid types. In *PLDI '08*, 2008.

[35] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In *European Conference on Object-Oriented Programming*, pages 248–274. Springer, 2003.

[36] R. Sekar. An efficient black-box technique for defeating web application attacks. In *NDSS*. Citeseer, 2009.

[37] F. Smith, D. Walker, and G. Morrisett. Alias types. In *European Symposium on Programming*, pages 366–381. Springer, 2000.

[38] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A class-based macro system for Java. In *Workshop on Reflection and Software Engineering*, pages 117–133. Springer, 1999.

[39] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. ACM New York, NY, USA, 2010.

[40] E. Van Wyk, L. Krishnan, D. Bodin, and A. Schwerdfeger. Attribute grammar-based language extensions for Java. In *European Conference on Object-Oriented Programming*, pages 575–599. Springer, 2007.

[41] P. Wadler. Linear types can change the world! In *Programming concepts and methods*, page 5. Citeseer, 1990.

[42] C.-W. Wang, S. W. Shieh, et al. DROIT: Dynamic alternation of dual-level tainting for malware analysis. *J. Inf. Sci. Eng.*, 31(1):111–129, 2015.

[43] K. Weitz, S. Srisakaokul, G. Kim, and M. D. Ernst. A format string checker for Java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 441–444, 2014.

[44] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, pages 121–136, 2006.

A

Code Examples

In this appendix we list the working code examples for the more complex behaviors, *i.e.*, the in-memory encryption, the off-memory encryption, and the email notifications.

A.1 In-memory Encryption

The behavior `InMemoryEncryptionBehavior` encrypts the provided string during its initialization to ensure it is encrypted in memory. For every access it checks if the context matches a package that has been specified in the `ioPackages` array. This check can also be adapted to check for specific classes or methods. If the current context matches, the string is returned in its decrypted form, and otherwise it remains encrypted in memory.

```
1 import java.util.Arrays;
2 import java.util.List;
3 import java.util.Random;
4
5 import javax.crypto.Cipher;
6 import javax.crypto.KeyGenerator;
7 import javax.crypto.SecretKey;
8 import javax.crypto.spec.IvParameterSpec;
9
```

```
10 public class InMemoryEncryptionBehavior implements IStringBehavior {
11     // variables required for encryption
12    private SecretKey key;
13    private IvParameterSpec iv;
14    private Random rnd = new Random();
15 
16     // packages that can trigger the behavior
17    private String[] ioPackages = { "java.net", "java.io" };
18 
19    /**
20     * Initialization of cryptographic tools.
21     */
22    public InMemoryEncryptionBehavior() {
23       try{
24          key = KeyGenerator.getInstance("DES").generateKey();
25          byte[] ivBytes = new byte[8];
26          rnd.nextBytes(ivBytes);
27          iv = new IvParameterSpec(ivBytes);
28       } catch(Exception e) {
29          e.printStackTrace();
30       }
31    }
32 
33    @Override
34    public String applyOnCreation(String original) {
35       // encrypt created String and store result in memory
36       return encrypt(original);
37    }
38 
39    @Override
40    public String applyOnRead(String encryptedStringInMemory) {
41       // check if data is leaked
42       StackTraceElement[] contexts = Thread.currentThread().getStackTrace();
43       for(int i = 1; i < contexts.length; i++) {
44          for(String ioPackageName: ioPackages) {
45             if(contexts[i].getClassName().startsWith(ioPackageName)) {
46                 // if data is leaked return the encrypted string
47                return encryptedStringInMemory;
48             }
49          }
50       }
51       // if data is not leaked, decrypt and return result
52       return decrypt(encryptedStringInMemory);
53    }
54 
55    @Override
56    public boolean applyDerivationRule(DerivationRule dr) {
57        // all derived strings must have this behavior, so we
58        // have to return true regardless of the DerivationRule
```

```
59        return true;
60    }
61
62    @Override
63    public boolean recordHistory() {
64        // we do not require a history tree for these strings; we return false
65        return false;
66    }
67
68    @Override
69    public String description() {
70        return "A behavior that performs in-memory encryption.";
71    }
72
73     /**
74      * Helper method that takes a plain text string and encrypts it.
75      */
76    private String encrypt(String s) {
77        try{
78            Cipher enCipher = Cipher.getInstance("DES/CBC/PKCS5Padding");
79            enCipher.init(Cipher.ENCRYPT_MODE, key, iv);
80            return new String(enCipher.doFinal(s.getBytes()));
81        } catch(Exception e) {
82            e.printStackTrace();
83        }
84        return new String(value);
85    }
86
87     /**
88      * Helper method that takes an encrypted string and decrypts it.
89      */
90    private String decrypt(String s) {
91        try{
92            Cipher deCipher = Cipher.getInstance("DES/CBC/PKCS5Padding");
93            deCipher.init(Cipher.DECRYPT_MODE, key, iv);
94            return new String(deCipher.doFinal(s.getBytes()));
95        } catch(Exception e) {
96            e.printStackTrace();
97        }
98        return s;
99    }
100 }
```

Listing 22: Working implementation of an in-memory encryption behavior

A.2 Off-memory Encryption

The behavior `OffMemoryEncryptionBehavior` encrypts the provided string if it is accessed from classes in untrusted packages. Therefore, it checks if the string is leaked to a class that is in a package that has been specified in the `ioPackages` array. This check can also be adapted for specific classes or methods. If the current context matches, the string is encrypted, and otherwise it remains decrypted in memory.

```
1  import java.util.Arrays;
2  import java.util.List;
3  import java.util.Random;
4
5  import javax.crypto.Cipher;
6  import javax.crypto.KeyGenerator;
7  import javax.crypto.SecretKey;
8  import javax.crypto.spec.IvParameterSpec;
9
10 public class OffMemoryEncryptionBehavior implements IStringBehavior {
11     // variables required for encryption
12    private SecretKey key;
13    private IvParameterSpec iv;
14    private Random rnd = new Random();
15
16     // packages that can trigger the behavior
17    private String[] ioPackages = { "java.net", "java.io" };
18
19    /**
20     * Initialization of cryptographic tools.
21     */
22    public OffMemoryEncryptionBehavior() {
23       try{
24          key = KeyGenerator.getInstance("DES").generateKey();
25          byte[] ivBytes = new byte[8];
26          rnd.nextBytes(ivBytes);
27          iv = new IvParameterSpec(ivBytes);
28       } catch(Exception e) {
29          e.printStackTrace();
30       }
31    }
32
33    @Override
34    public String applyOnCreation(String s) {
35        // we do not require any code here and use the default implementation
36        return s;
37    };
38
39    @Override
40    public String applyOnRead(String s) {
```

```
41          // check if data is leaked
42          StackTraceElement[] contexts = Thread.currentThread().getStackTrace();
43          for(int i = 1; i < contexts.length; i++) {
44             for(String ioPackageName: ioPackages) {
45                if(contexts[i].getClassName().startsWith(ioPackageName)) {
46                    // if harmful package is detected, return encrypted string
47                   return encrypt(s);
48                }
49             }
50          }
51          // if no harmful package detected, return the decrypted string
52          return s;
53       }
54
55       @Override
56       public boolean applyDerivationRule(DerivationRule dr) {
57           // all derived strings must receive this behavior, so we
58           // have to return true regardless of the DerivationRule
59          return true;
60       }
61
62       @Override
63       public boolean recordHistory() {
64           // we do not require a history tree for these strings, so we return false
65          return false;
66       }
67
68       @Override
69       public String description() {
70          return "A behavior that performs off-memory encryption.";
71       }
72
73        /**
74         * Helper method that takes a plain text string and encrypts it.
75         */
76       private String encrypt(String s) {
77          try{
78             Cipher enCipher = Cipher.getInstance("DES/CBC/PKCS5Padding");
79             enCipher.init(Cipher.ENCRYPT_MODE, key, iv);
80             return new String(enCipher.doFinal(s.getBytes()));
81          } catch(Exception e) {
82             e.printStackTrace();
83          }
84          return new String(value);
85       }
86    }
```

Listing 23: Working implementation of an off-memory encryption behavior

A.3 Email Notifications

The implementation of this behavior is split in two parts: the behavior itself, *i.e.*, `SendMailBehavior` and the email agent, *i.e.*, `MailAgent`. The behavior is rather simple and only checks if the stack trace contains any occurrences of warning or critical packages. If a match occurs, it performs the specified actions, *i.e.*, it immediately sends an email if a critical leak was detected, and it queues warnings whose reports are only periodically sent by email. As before, this check can easily be modified to support specific classes and methods.

```
import java.util.Arrays;
import java.util.List;
import java.util.StringJoiner;

// importing the mail agent
import MailAgent;

public class SendMailBehavior implements IStringBehavior {
    // packages that can trigger the behavior
   private String[] criticalPackages = { "java.net" };
   private String[] warningPackages = { "java.io" };

   @Override
   public String applyOnCreation(String s) {
       // we do not require any code here and use the default implementation
       return s;
   };

   @Override
   public String applyOnRead(String s) {
       // check if data is leaked
      StackTraceElement[] contexts = Thread.currentThread().getStackTrace();
      for(int i = 1; i < contexts.length; i++) {
         for(String errorPackageName: criticalPackages) {
            if(contexts[i].getClassName().startsWith(errorPackageName)) {
               // if data is leaked in a critical package:
               // immediately send an email notification
               MailAgent.sendError(
                  "Sensitive data was leaked at:\n"
                  + stackTraceToString(contexts)
               );
            }
         }
         for(String warningPackageName: warningPackages) {
            if(contexts[i].getClassName().startsWith(warningPackageName)) {
               // if data is accessed in a warning package:
               // add message to a buffer that is periodically sent as report
               MailAgent.queueWarning(
```

```
                    "Data was stored to disk at:\n"
                    + stackTraceToString(contexts)
                );
            }
        }
    }
    // continue execution without altering the string value
    return s;
  }

  @Override
  public boolean applyDerivationRule(DerivationRule dr) {
      // all derived strings must get this behavior, so we
      // have to return true regardless of the DerivationRule
    return true;
  }

  @Override
  public boolean recordHistory() {
      // we do not require a history tree for the strings, so we return false
    return false;
  }

  @Override
  public String getDescription() {
    return "A behavior that can send emails when issues are detected.";
  }

   /*
    * Helper method to transform StackTraceElement[] to a multiline string.
    */
  private static String stackTraceToString(StackTraceElement[] context) {
    StringJoiner sj = new StringJoiner("\n");
    for(int i = 5 ; i < context.length ; i++){
      sj.add(context[i].toString());
    }
    return sj.toString();
  }
}
```

Listing 24: Working implementation of a behavior that supports email reports

The main purpose of the `MailAgent` class is the submission of emails. In addition, it can aggregate warnings into reports that it can send periodically. In order to make it work, it must be configured with valid credentials, *i.e.*, a user name and password as well as a sender and receiver email address, and the SMTP server parameters of the email server. It is further possible to specify a time interval that will be used for the periodical reports.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Properties;
import java.util.StringJoiner;

import javax.mail.Authenticator;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Multipart;
import javax.mail.PasswordAuthentication;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeBodyPart;
import javax.mail.internet.MimeMessage;
import javax.mail.internet.MimeMultipart;

public class MailAgent {
    // sender and receiver email addresses
    private final String receiverEmail = "admin@some.domain.com";
    private final String senderEmail = "server@some.domain.com";

    // default report interval is 24 hours
    private final int reportIntervalInSeconds = 60*60*24;

    // list of collected warnings since last report
    private List<String> warningList = new ArrayList<>();

    /**
     * Initializes a new thread that handles the periodic report submission.
     *
     */
    public MailAgent() {
        new Thread() {
            public void run() {
                try {
                    Thread.sleep(reportIntervalInSeconds * 1000);

                    // if any warnings are queued for the report
                    if(!warningList.isEmpty()){
                        // collect all warnings into one String
                        StringJoiner sj = new StringJoiner("\n\n");
                        for(String warningMessage: warningList)
                            sj.add(warningMessage);

                        // send the report
                        sendMessage("Warning report", sj.toString());
                    }
                } catch (InterruptedException e) {
```

```
50                      e.printStackTrace();
51                  }
52              }
53          }.start(); // start the report thread
54      }
55
56      /**
57       * Gets email server properties.
58       */
59      private Properties getProps() {
60          Properties props = new Properties();
61          props.put("mail.smtp.auth", true);
62          props.put("mail.smtp.starttls.enable", "true");
63          props.put("mail.smtp.host", "smtp.mailtrap.io");
64          props.put("mail.smtp.port", "25");
65          props.put("mail.smtp.ssl.trust", "*");
66          return props;
67      }
68
69      /**
70       * Creates a session to the email server.
71       */
72      private Session getSession() {
73          Session session = Session.getInstance(getProps(), new Authenticator() {
74              @Override
75              protected PasswordAuthentication getPasswordAuthentication() {
76                  return new PasswordAuthentication(senderEmail, "password");
77              }
78          });
79          return session;
80      }
81
82      /**
83       * Adds a warning message to the report queue.
84       */
85      public void queueWarning(String warningDescription){
86          warningList.add(warningDescription);
87      }
88
89      /**
90       * Sends error emails immediately.
91       */
92      public boolean sendError(String errorDescription){
93          return sendMessage("Critical security alert", errorDescription);
94      }
95
96      /**
97       * Sends an email with the specified title and content.
98       */
```

```
99      private boolean sendMessage(String title, String content) {
100         Message message = new MimeMessage(getSession());
101         try {
102             // set header information of the message
103             InternetAddress[] receivers = new InternetAddress().parse(receiverEmail);
104             InternetAddress[] senders = new InternetAddress().parse(senderEmail);
105             message.setFrom(senders);
106             message.setRecipients(Message.RecipientType.TO, receivers);
107             message.setSubject(title);
108 
109             // set the message body
110             String msg = content;
111             msg = msg.replace("\n", "<br>");
112 
113             MimeBodyPart mimeBodyPart = new MimeBodyPart();
114             mimeBodyPart.setContent(msg, "text/html; charset=utf-8");
115 
116             Multipart multipart = new MimeMultipart();
117             multipart.addBodyPart(mimeBodyPart);
118 
119             message.setContent(multipart);
120 
121             // send email to the SMTP server
122             Transport.send(message);
123 
124             return true; // message was sent successfully
125         } catch (MessagingException e) {
126             e.printStackTrace();
127             return false; // something went wrong during submission
128         }
129     }
130 }
```

Listing 25: Implementation of the email agent