Automated Hotfixes for Misuses of Crypto APIs

by

Kristen Newbury

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Cryptographic (crypto) Application Programming Interfaces (APIs) play an important role in application security; unfortunately crypto APIs are difficult to use, which may lead to security vulnerabilities. Prior work have looked at detecting and fixing crypto APIs misuses at development time and in the setting of software patching. However, software patching for security vulnerabilities is not ideal for addressing vulnerability windows in servers in a timely manner. An alternative approach to software patching is hotfixing. In this paper, we present HOTFIXER, a tool that performs automatic crypto API misuse hotfixing at Java application runtime. To apply its fixes, HOTFIXER automatically transforms hand-crafted software patches into hotfixes that are valid to use by Java agents. We have evaluated HOTFIXER on a set of 103 microbenchmarks, and a set of 27 crypto API misuses found across 7 real-world Java applications. HOTFIXER detects and fixes all misuses in 95% of all benchmarks, in an identical manner compared to applying a develop-time patch. Additionally, we have empirically validated that HOTFIXER preserves identical application behaviour compared to software patching in 98% of all benchmarks. Compared to software patching, the performance overhead that HOTFIXER induces for all benchmarks is at most 17%.

Preface

This thesis is an original work by Kristen Newbury. We intend to publish this work in the future. The research conducted for this thesis was done as part of a collaborative project with IBM Centre for Advanced Studies (CAS) Canada. I was responsible for design decisions and implementing the system presented in this thesis, as well as the experimental setup and evaluation of our system. Andrew Craik was responsible for technical advice and contributions to concept formation and system design, as well as thesis edits. Karim Ali was the supervisory author and contributed to concept formation and system design, as well as many thesis edits.

Acknowledgements

Thank you to Karim Ali for all of your patience and effort. I would not be brave enough to write this without you, and more importantly you've always set a good example for how to be an incredibly intelligent person and also to make room for oneself to be multi-faceted.

Thank you to Andrew Craik, for always being enthusiastic about having technical discussions, even when I need to ask for clarification on how something works for the third time.

Thank you to all of my mentors, especially Nelson Amaral. You were the first professor that I ever really felt brave enough to actually talk to. The effort you put into teaching is inspiring and apparent, and I appreciate every effort that you've made to include me in group discussions and meetups.

Thank you to my mom, for having an unreasonable amount of faith in me. You always thought I could do it, buckle down and make sure I do my best.

Thank you to Amy for being my sweetest and silliest companion through all of this.

Thank you to all of my fellow classmates and lab-coworkers who have been there along the way, to bounce ideas of off and talk about work with. You all have helped me strive to be as intelligent and hard working as you.

This work was made possible with the funding support of the IBM Center for Advance Studies (CAS) Canada, the Government of Alberta, and the University of Alberta.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Detecting misuse of crypto APIs is an important component of application security. A crypto API is a library that provides a developer with functionality to perform crypto tasks such as encryption. If such crypto tasks are not performed in a secure manner, applications may accidentally leak private user data, leading to both economic losses as well as a loss of trust from the users of the affected applications.

Unfortunately, crypto APIs remain difficult to use, despite years of tool development. Some of these tools offer misuse detection [1, 2, 3, 4, 5, 6], and others help generate secure code to perform crypto tasks [7]. While existing solutions are valuable during the development of the codebase, they are insufficient to protect an application from all vulnerabilities at all times. For example, using existing tools to fix misuses of crypto APIs on a server requires shutting down the server, developing the patch offline, deploying the patched code to the server, and, finally, restarting the server. Such a scenario is impractical, for example, in the case of a long running server that cannot afford to restart at all or may only restart at well-defined intervals. Even more commonly, when considering the time and resources required to develop and deploy a patch, an unacceptable window of vulnerability may arise in the application. If any such window exists, an attacker has the opportunity to exploit the vulnerability.

A solution to this exploitation window is to apply a *hotfix* to the application. A *hotfix* is similar to a patch, in that it serves the purpose of fixing a misuse. However, instead of having the patch classes load only when the runtime is restarted, a hotfix allows the patch code to begin execution in a running application. Although some prior work provides automatic crypto API misuse hotfixing [8, 9], the techniques are limited to Android applications. To detect and fix those misuses, such tools use fixed sets of misuse patterns. In this paper, we seek to improve on the coverage of the current state of the art crypto API hotfix systems over the types of crypto API misuses that can be handled, as well as the understandability of the mechanism used to perform the hotfix. Additionally we specialize in systems where hotfixing is more beneficial than software patching.

We propose Hotfixer, a tool that performs automatic crypto API misuse detection and hotfixing for a running Java application. Given a patch, Hotfixer automatically converts this patch to a hotfix. To perform hotfixing, Hotfixer uses a Java agent, a client of the Java Instrumentation API, to modify classes in the currently running application. While using a Java agent provides us a simple mechanism to perform a hotfix, a Java agent has some limitations. In this work we use developer-provided patches as hotfixes, however, certain types of changes in patches cause an exception in the agent, which may result in Java Virtual Machine (JVM) shutdown if we attempt to use them as is. We call patches that contain any class that causes an exception redefinition non-compliant. To deal with redefinition non-compliant patches we contribute a patch adapter that is capable of detecting and fixing non-compliant changes in developer-provided patches so that a Java agent can use them to perform hotfixing. Our patch adapter contribution is a modular component of Hotfixer, therefore it can be used in a variety of settings.

Our hypothesis is that hotfixing is a viable and beneficial alternative to software patching. In this thesis, we investigate this hypothesis by answering the following research questions:

RQ1: How successful is Hotfixer at fixing crypto API misuses?

RQ2: Does Hotfixer alter the semantics of a running application?

RQ3: How does Hotfixer affect application performance?

To answer the above research questions we present the following contributions to crypto API hotfixing:

- Hotfixer, a tool that performs automatic runtime crypto API misuse detection and misuse hotfixing.
- A novel methodology for adapting patches into hotfixes that Hotfixer successfully applies to a running system.
- An evaluation of Hotfixer in its ability to detect crypto APIs and to apply hotfixes. We find that we are able to detect and hotfix crypto API misuses in 95% of benchmarks in an identical manner to the develop-time patch strategy, over two datasets, one of 103 microbenchmarks and the other a set of 27 benchmarks found across 7 real-world Java applications.
- An evaluation of Hotfixer in its ability to preserve semantics. We find that Hotfixer only alters program semantics in 2% of the total benchmarks, compared to a develop-time patch.
- An evaluation of Hotfixer in its ability to preserve application performance, again, with respect to a develop-time patch. We find that the overhead of Hotfixer not exceed 17% loss at steady state compared to software patching, for the dataset that we evaluate on.

The remainder of this thesis is organized as follows: in Chapter 2 we present necessary background information on JVMs in general, followed by specific details of Eclipse OpenJ9. We then define crypto API misuses and how software patching would address crypto API misuses. Lastly in Chapter 2, we describe Java agents. Chapter 3 presents an overview of Hotfixer, including the steps performed in each of its two phases. Chapter 4 discusses our patch adapter contribution, and the techniques that we use to deal with various types of changes in patches. Chapter 4 also contains multiple code examples to illustrate how the patch adapter in Hotfixer works. Chapter 5 describes the concluding workflow of Hotfixer and enumerates categories which patches may fall into. These categories serve as the basis for understanding the capabilities of our patch adapter, which we evaluate in Chapter 6. In our evaluation, we answer the three research questions listed above. Lastly, we present related work and our conclusion in Chapters 7 and 8.

Chapter 2

Background Material

2.1 Java Virtual Machine

A JVM is a virtual machine that can run Java programs. A JVM has many components that all work together to execute the program; some of those components include (1) an interpreter loop, and (2) a Just-In-Time (JIT) compiler. The interpreter loop is the part of the JVM that executes Java bytecode, which in a typical JVM comes directly from Java classfile format, and is introduced into the JVM by a classloader. Before execution, a Java program must be bytecode compiled by javac, a tool which translates Java source code into Java bytecode. To execute bytecode in a JVM it must be loaded into the JVM by a classloader. A classloader is responsible for managing which classes are represented in an application at a given time, by receiving a Fully Qualified Name (FQN) and depending on the delegation model (a policy on how classloaders interact and delegate amongst themselves), reading a classfile from a resource location. The classloader knows the exact resource location from 2 pieces of information: (1) FQN and (2) classpath. A FQN is comprised of the class name, as well as a package name. A classpath is a path on the machine where the JVM is executing that contains the bytecode to be executed. Once classes are loaded in the JVM, they are executed by the interpreter. This method of execution is slow however, therefore a typical modern JVM will also have a JIT. A JIT compiles Java bytecode into platform dependent native code, which is much faster than interpretation, and can be optimized. Next we discuss the importance of Java classfile format, and the

JIT, in relation to this work.

2.2 Eclipse OpenJ9

In this work we use Eclipse OpenJ9, an open-source JVM, as the base for a components of Hotfixer, and also the JVM that runs all components of Hotfixer. The JIT is important to this work because it is where we initiate the operations performed by Hotfixer. Eclipse OpenJ9 naturally sorts methods into compilation levels, based on which optimizations that the JIT performs on that method, which are determined by invocation counters. Methods that execute most frequently will be compiled at the highest level, and may be compiled at multiple levels along the way. The compilation levels in Eclipse OpenJ9 are: cold, warm, hot, very hot, and scorching. We utilize all compilation levels in this work.

In this work we utilize a unique feature of Eclipse OpenJ9 called the Shared Class Cache (SCC). The SCC is a portion of shared memory managed by Eclipse OpenJ9. The SCC holds classes in a format called RomClass, which acts as an alternative to classfile format. When a class is loaded into the SCC, some of the work that a classloader would perform is completed, and then the RomClass can be used by multiple JVMs or by the same JVM across multiple executions of the same application. This sharing and reusing enables a JVM to have better startup performance than simply using classfile format. Eclipse OpenJ9 also naturally manages which classes get stores as RomClasses and if a class is updated in the file system it will be marked as stale in the SCC and that outdated class representation will no longer be used. We have built Hotfixer to prefer to use the RomClasses from the SCC for analysis, however, if a class is not present in the SCC Hotfixer will fall back on classfile format.

We prefer to utilize RomClasses over classfile format due to one challenge of static analysis applied to runtime environments related to software versioning. Software is composed of components which evolve over time, from one version to the next; this is Lehman's Law of Continuing Change [10]. When performing static analysis, we must guarantee that the classes that we analyse are the same versions as those that are running in an application JVM, otherwise the results of that analysis are irrelevant to the application that is currently executing. One potential solution to this irrelevant result problem is to send every class to be analyzed from the JVM to Hotfixer. The analysis however may require many classes, and, does not know which it will need up front. Therefore, a solution of sending one class at a time from the JVM to Hotfixer would be tedious and expensive. Luckily, we are able to mitigate the irrelevant result problem by having Hotfixer use the SCC.

2.3 Crypto API Misuses

A crypto API misuse arises during application development not from the details of the implementation of the crypto API itself but instead from the design choices of the API whereby it is possible to use the crypto API in a way that creates a vulnerability. Figure 2.1 shows an example of an encryption task in Java that uses the Java Cryptography Architecture (JCA), which is a common framework for performing crypto tasks in Java. In Line 3 the developer has created a `Cipher` object, and specified what algorithm, mode, and, padding they want the encryption to use. In the next line they set the cipher object up to be able to do encryption, and in the last line they call `doFinal`, the method that performs the encryption on the bytes of the supplied data. In Line 3 the developer chose ECB mode for encryption; ECB uses some repetitions [11] of encryption upon blocks of plaintext data, which can result in patterns in the ciphertext and therefore a possibility that the ciphertext can leak information to an attacker. A more secure mode for encryption is CBC. Crypto API misuse detection tools can help developers to detect such misuses however, by alerting the developer to the location of the misuse and potentially suggesting ways that the developer can fix the misuse.

2.4 Software Patching and Hotfixing

As mentioned in the introduction, existing crypto API misuse detection tools are valuable during the development of the codebase, however, if an application

```java
public class Util{
  public bytes[] encrypt(...){
      Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
      cipher.init(Cipher.ENCRYPT_MODE, secretKey, paramSpec);
      return cipher.doFinal(data.getBytes());
  }
}
```

Figure 2.1: An example of a crypto API misuse.

is already deployed and is discovered to contain a vulnerability the developer has 2 choices of how to proceed to fix the misuse: (1) software patching or (2) hotfixing. In both cases the developer fixes the misuse in a local development setting as a first step, and the method in which the fix is deployed is what differs. Through software patching the developer shuts down the server where the application is running, replaces the application with the new version that contains the fix, and then restarts the server. In this case the developer does not have to worry about any state transfer of the running application, however inherently there is a cost to restarting the application. As mentioned in the introduction it may not be possible to restart the application, or it is not clear when exactly it is alright to do the restart, and inevitably this process takes time which can cause windows of vulnerability in the application. The alternative to software patching is hotfixing.

Hotfixing is when the fix for the misuse can be directly integrated into the running application, with no need for a restart. There are multiple techniques to perform hotfixing in Java, for example Aspect Oriented Weaving (AOP) [12], or as another example, Java agents. Originally, a Java agent referred to clients (written in C or C++) of the standard native API provided by Java for debugging (i.e., Java Virtual Machine Tool Interface—JVMTI) [13]. However, the term is now also commonly used to refer to clients (written in Java) of the Java Instrumentation API, because both interfaces provide similar functionality. In this work we use a Java agent written in Java to perform hotfixing. The mechanism is simple to understand; a Java agent can perform a redefinition event where the definitions of a set of specified classes are replaced with new definitions. The Java agent guarantees that subsequent invocations of methods that have been replaced will use the redefined implementation, however, if there are some methods currently running, and a redefinition of that method occurs, the old implementation of the method will continue execution until it is complete. A redefinition event does not re-run class initializers,which are run only when explicitly invoked, or in the case of static initializers, when the class is first loaded by the classloader.

Chapter 3

Overview of Hotfixer

We built Hotfixer on top of Eclipse OpenJ9 [14], CogniCrypt [2], and Soot [15]. Eclipse OpenJ9 is an open-source JVM, CogniCrypt is a static analysis tool for crypto API misuse detection, and Soot is a bytecode analysis and optimization framework. The main workflow of Hotfixer consists of two phases: a crypto API misuse detection phase (Phase I) and a hotfix phase (Phase II). Each phase involves the major components of Hotfixer: (1) OpenJ9_Hotfixer_Version, our enhanced version of Eclipse OpenJ9 [14], (2) CogniCrypt_Hotfixer_Server, our enhanced version of CogniCrypt [2], and (3) Soot_Hotfixer_Version, our enhanced version of Soot [15]. Figure 3.1 depicts the general workflow of Hotfixer. Figure 3.2 depicts the specific events, and messages that are transmitted, between the components of Hotfixer.

3.1 Phase I: Crypto API Misuse Detection

Phase I consists of two major steps: (1) crypto API use detection and (2) analysis. Hotfixer starts when CogniCrypt_Hotfixer_Server sets up a server that waits for analysis requests. Then, when OpenJ9_Hotfixer_Version runs in Hotfixer mode, it connects to CogniCrypt_Hotfixer_Server during JVM startup. During this connection setup, CogniCrypt_Hotfixer_Server provides OpenJ9_Hotfixer_Version with a set of analysis seeds. The seeds are simply the names of classes that CogniCrypt_Hotfixer_Server has rules for. OpenJ9_Hotfixer_Version uses the analysis

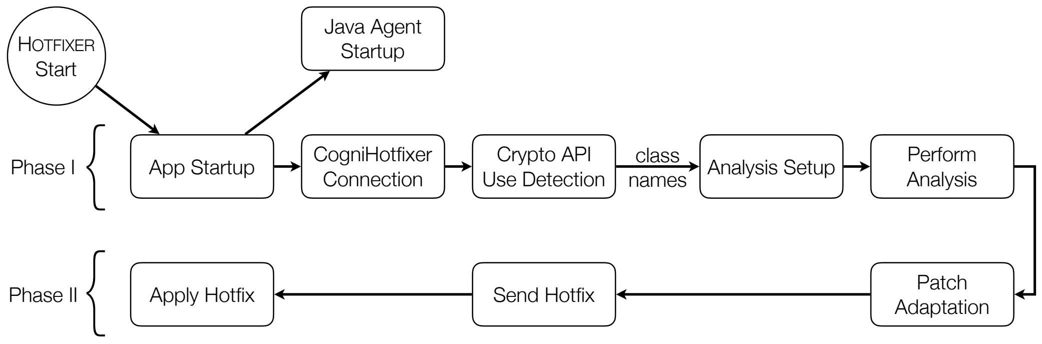


Figure 3.1: Detailed Overview of HOTFIXER Phases.

seeds to detect uses of crypto APIs in the running application. The JIT compiler of the application JVM uses the seeds to see if a currently compiling method contains any calls to crypto APIs. We setup the JIT to search for crypto API uses in all methods at all Eclipse OpenJ9 compilation levels so that we get maximum analysis coverage over all compiling methods. We use the JIT to search, as opposed to the interpreter in the JVM, to prioritize methods that are more likely to contribute to application security. This means if certain methods never compile, we will miss some uses, however, at this time we do not investigate the value of searching in methods that do not compile.

For each method under compilation, the JIT uses a check for whether each full signature of each callee in the method matches any of the seeds. We use the full signature of a callee, as opposed to just the method name, to ensure that (1) we classify invocations of methods from classes in a crypto API as uses, and (2) crypto API type parameters will trigger the detection of a use of a crypto API. For the example shown in Figure 3.3a, during the compilation of `Util.encrypt`, the JIT encounters the callsite `keyFactory.generateSecret(...)`. The JIT will then look at the callee method signature `javax.crypto.SecretKeyFactory.generateSecret:(java.security.spec.KeySpec)`. This signature will match against the seed: `javax.crypto.SecretKeyFactory`. When the JIT finds a crypto API use, it sends the name of the class that contains the method under compilation to COGNICRYPT_HOTFIXER_SERVER. In the example from Figure 3.3a, the JIT will send to COGNICRYPT_HOTFIXER_SERVER the classname: `Util` so that it can be statically analyzed. To ensure that HOTFIXER

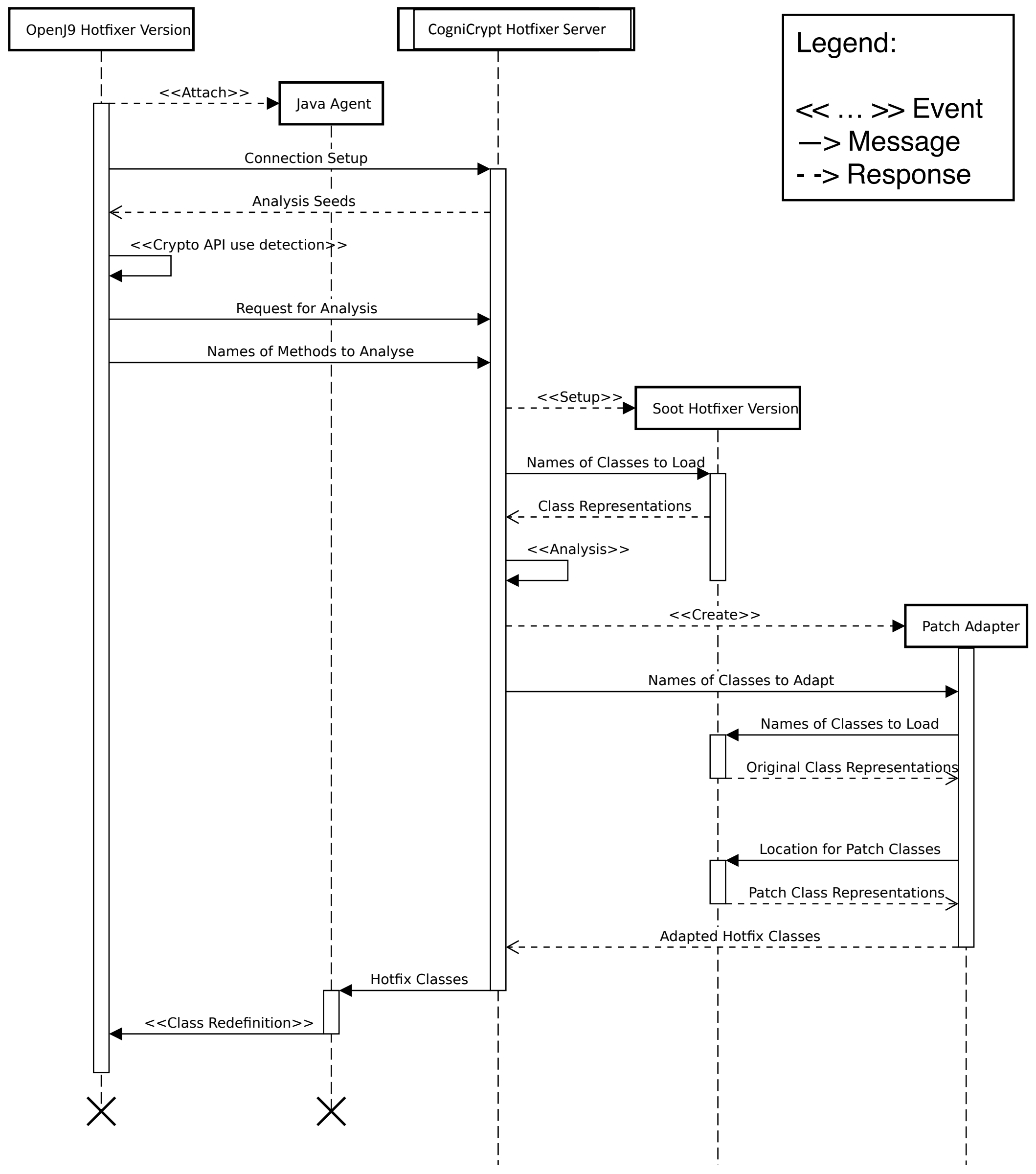


Figure 3.2: Sequence Diagram Overview of Events and Messages Between Components of HOTFIXER.

analyzes the same classes that are loaded in the running application, Soot_-Hotfixer_Version prefers to obtain the classes from Eclipse OpenJ9's SCC. If necessary classes are not found in the SCC, Soot_Hotfixer_Version uses class file format. Once Soot_Hotfixer_Version has gathered the necessary classes for the analysis, CogniCrypt_Hotfixer_Server produces a report of misuses that it detected. CogniCrypt_Hotfixer_Server then determines whether it has a patch that can be applied to the detected misuses. If CogniCrypt_Hotfixer_Server has a patch that Hotfixer can use, our system then enters Phase II.

3.2 Phase II: Hotfixing

The major step performed in Phase II is patch adaptation. CogniCrypt_-Hotfixer_Server invokes our patch adapter (Section 4), which we built on top of Soot_Hotfixer_Version. After patch adaptation, applying the hotfix occurs in OpenJ9_Hotfixer_Version. Similar to CogniCrypt_-Hotfixer_Server, to reduce any adverse effect on the normal execution of the analyzed application, both patch adaptation and hotfix receival occur asynchronously with respect to the execution of the application.

As can be seen in Figure 3.1, patch adaptation consumes a patch, and outputs a hotfix. Both a patch and a hotfix are sets of secure classes that are designed to replace insecure one(s), however, only the hotfix is applicable to a running application. To see why a hotfix can be applied to a running application, while a patch cannot, first we must present the redefinition mechanism used in Hotfixer. To apply the hotfix to the running application, we have implemented HotfixerAgent, a custom Java agent.

Although HotfixerAgent is an effective tool to modify running classes in a JVM, the types of changes that it can apply are limited. According to the Java Instrumentation API documentation [16]: *"The retransformation may change method bodies, the constant pool and attributes. The retransformation must not add, remove or rename fields or methods, change the signatures of methods, or change inheritance."* Because of these limitations, manually spec-

```
 8 public class Util{
 9   private static final byte[] SALT = "staticSalt".getBytes();
10
11   public bytes[] encrypt(String data){
12       SecretKey key = keyFactory.generateSecret(new PBEKeySpec(habridgeKey));
13       ...
14       cipher.init(Cipher.ENCRYPT_MODE, key, new PBEParameterSpec(SALT, 20));
15       ...
16   }
17 }
```

(a) The misuse that CogniCrypt detects (Line 14).

```
18 public class Util{
19
20   public bytes[] encrypt(String data){
21       SecretKey key = keyFactory.generateSecret(pbeks);
22       ...
23       cipher.init(Cipher.ENCRYPT_MODE, key, paramSpec);
24       ...
25   }
26
27   //added field
28   private PBEKeySpec pbeks;
29   private AlgorithmParameterSpec paramSpec;
30
31   //added method
32   private void initPBEKeySpec(){
33     try{
34         char[] password = new char[] {...};
35         byte bytesForKey[] = new byte[32];
36         SecureRandom secureRandom = SecureRandom.getInstance("SHA1PRNG");
37         secureRandom.nextBytes(bytesForKey);
38         pbeks = new PBEKeySpec(password, bytesForKey, 10299, 128);
39         byte bytesForSalt[] = new byte[8];
40         secureRandom.nextBytes(bytesForSalt);
41         paramSpec = new IvParameterSpec(bytesForSalt);
42     }catch(NoSuchAlgorithmException e){
43        ...
44      }
45   }
46
47
48 }
```

(b) The patch that fixes the error.

Figure 3.3: An example illustrating a patch that adds a new method to a class.

```
49 public class Util{
50   public static String alg = "DES/CBC/PKCS5Padding";
51   public static bytes[] encrypt(...){
52     Cipher cipher = Cipher.getInstance(alg);
53     ...
54   }
55 }
```

(a) The misuse that CogniCrypt detects (Line 52).

```
56 public class Util{
57   public static String alg = "AES/CBC/PKCS5Padding";
58   public static bytes[] encrypt(...){
59     Cipher cipher = Cipher.getInstance(alg);
60     ...
61   }
62 }
```

(b) The patch that fixes the error.

Figure 3.4: An example illustrating a patch that modifies a static field of a class.

ified patches may cause an exception if we attempt to use them as is. For example, Figure 3.3 illustrates a crypto API misuse that CogniCrypt detects due to a Required Predicate Error (Line 14). The error in Line 14 refers to a crypto API use that relies on another crypto API use in Line 12, that is insecure, thus making both uses insecure. To fix the error in Line 14 without causing an exception the developer must introduce a multi-statement fix. Therefore, the developer should place this fix into a new method in `Util`. However, due to these API limitations on redefinition, this patch would be redefinition non-compliant and HOTFIXER must adapt this patch before HOTFIXERAGENT uses it as in a hotfix.

Moreover, according Java Instrumentation API documentation [16]: *"redefining a class does not cause its initializers to be run"*. Initializers are methods in a class that can set the initial state for its fields. Classes have an initial state when they, for example, declare static fields. Because the redefinition mechanism does not involve re-running class initializers, if the patch changes a static field value, that change will not take effect for any current instances of classes under redefinition. For example, Figure 3.4a shows a Constraint Error in Line 52. The solution to this misuse is to use a more secure encryption algorithm, as can be seen in Line 57 in Figure 3.4b. However, since static initializers are not rerun during redefinition events, encryption using the redefined `encrypt` method, in Line 58, will not use the algorithm specified in Line 57. The only way to observe the intended effects of this patch is for our patch adapter to adjust it.

Chapter 4

Adapting a Patch into a Hotfix

A developer-provided patch may contain redefinition non-compliance because at develop-time the developer is only focused on fixing the crypto API misuse. To fix redefinition non-compliance, Hotfixer uses our patch adapter. Hotfixer ensures that the transformation preserves the semantics of the patch while making adjustments. In Section 6, our evaluation shows that our patch adapter meets these goals.

To perform a transformation, the patch adapter first identifies the differences between each class in the patch (i.e., a redefinition class) and its corresponding original version. The patch adapter assesses those differences at a non-compliant item level, as opposed to a statement or token level. The non-compliant items that Hotfixer handles are: (1) field addition and (2) method addition.

4.1 Handling Field/Method Addition

Addition of either fields or methods represents the more complex scenarios that our patch adapter handles. To handle addition of either a field or a method, we follow the same general approach. For each, the patch adapter moves the added field or method in a completely new class and then updates all references to that field or method. However, there are key differences between handling fields and handling methods.

4.1.1 Field Addition

In the case where the patch contains field addition, Hotfixer first moves the added field into a completely new class. For each class in a patch, Hotfixer constructs a corresponding new class that hosts the added fields, upholding the semantics of the patch while transforming it into a redefinition compliant hotfix. To minimize the changes that this addition introduces on the field visibility, Hotfixer constructs the new class in the same package as the patch class. After moving the field, Hotfixer also adapts its field references in the patch. In Java, a field reference consists of the field name and the owner class of the field. If the moved field is static, Hotfixer simply replaces its references to match the new owner of the field. If the moved field is non-static (i.e., an instance field), Hotfixer categorizes all its references in the patch as either uses or definitions. Hotfixer then replaces these references with calls to newly constructed getters and setters to access the field. For consistency, we use this replacement strategy regardless of the field visibility. However, for private fields, we must expose a protected getter/setter to the added field, enabling access to that added field from the patch class once we move it into a new class.

Figure 4.1 illustrates an example for the case where instance field addition requires further adjustment. In the patch, the base of the field reference `p` is a variable of the type of the redefinition class `Util` that the field was added to (Line 76). However, in the hotfix, when Hotfixer moves the field to the new class `UtilNew`, simply changing the field reference's owner to match its new owner is not sufficient. Hotfixer must also produce a variable that matches the new type that is required to access the field. To fix this issue, Hotfixer creates two static hashtable fields in the new class `UtilNew` that maintain a bidirectional mapping of instances of redefinition class objects to corresponding new class instances (Lines 94–95). Hotfixer populates both hashtables in the constructor of the redefinition class (Lines 103–104). For each instance of the redefinition class, our hotfix creates an instance of the new class that corresponds to the redefinition class `Util` and stores it in the

```
63 public class User{
64   public void main(){
65     Util p = new Util();
66
67   }
68 }
69
70 public class Util{
71
72 }
```

(a) The original code.

```
73 public class User{
74   public void main(){
75     Util p = new Util();
76     p.aField = "Hello";
77   }
78 }
79
80 public class Util{
81   public String aField;
82 }
```

(b) The developer patch.

```
84 public class User{
85   public void main(){
86     Util p = new Util();
87     UtilNew us = UtilNew.redefToNew.get(p);
88     us.setAddedField("Hello");
89   }
90 }
91
92 public class UtilNew{
93   public String aField;
94   public static Hashtable<Util, UtilNew> redefToNew;
95   public static Hashtable<UtilNew, Util> newToRedef;
96   public String getAddedField(){ return aField; }
97   public void setAddedField(String s){ aField = s; }
98 }
99
100 public class Util{
101   Util(){
102     UtilNew us = new UtilNew();
103     UtilNew.redefToNew.put(this, us);
104     UtilNew.newToRedef.put(us, this);
105   }
106 }
```

(c) Our generated hotfix.

Figure 4.1: An example illustrating a developer patch that adds an instance field to a class and the hotfix that our patch adapter generates by moving the added field to a new class and replacing its references with calls to getters and setters.

hashtable. For each instance field reference, HOTFIXER adjusts the reference location by using the redefinition class variable to lookup the corresponding new class instance. HOTFIXER then uses that returned object to refer to the copied field (Line 87).

4.1.2 Method Addition

Similar to handling field addition, HOTFIXER moves an added method into the new class that corresponds to the redefinition class. If the moved method is `private`, HOTFIXER relaxes its visibility to `protected` such that it can be accessed from the patch class. After moving the method, HOTFIXER modifies all of the references to the moved method. Unlike our solution to field references, HOTFIXER additionally considers dynamic dispatch for method references. If we simply update moved method references to refer to the new class method, the intended semantics of the patch may change. In Java, a method reference consists of a class identifier and a method signature. Statically replacing the class identifier in the method reference would disallow dynamic dispatch from functioning correctly. If the reference's class identifier is for the new class, but the runtime type for receiver of the call is actually a parent class of the redefinition class, which is not related to the new class by inheritance, then the call could not be resolved to the intended target. To handle dynamic dispatch, HOTFIXER first determines the targets that methods may resolve to. To achieve that, HOTFIXER uses a call graph constructed by SOOT_HOTFIXER_VERSION. For each method reference, there are two cases that may occur that the patch adapter addresses.

Monomorphic Call Sites Following the convention in traditional inlining strategies [15], we refer to a call site as monomorphic [17] if it is guaranteed to have only one explicit target. For such call sites, it is safe to perform an outright replacement of the original method invocation with a reference to the moved method in our hotfix.

For example, in Figure 4.2, the method invocation to `emitMsg()` (Line 122) may resolve only to `Util.emitMsg()`. For this invocation, our adaptation strategy replaces the invocation outright with a reference to the moved method (Line 134). For instance method references, we use the hashtable (Line 139) in the new class `UtilNew` to lookup the correct object reference to use to invoke the moved method `emitMsg()` (Line 133). This lookup is similar to our approach for handling references to instance field.

Polymorphic Call Sites In this case, the method invocation has multiple potential targets. Figure 4.3 shows an example of a polymorphic call site (Line 180). When HOTFIXER adds `emitMsg()`, that call site can only be re-

```java
public class User{
  public void main(){
    Util p = new Util();


  }
}

public class Util{

}
```

(a) The original code.

```java
public class User{
  public void main(){
    Util p = new Util();
    // monomorphic call
    p.emitMsg("Hello");
  }
}

public class Util{
  public void emitMsg(String str){...}
}
```

(b) The developer patch.

```java
public class User{
  public void main(){
    Util p = new Util();
    UtilNew us = UtilNew.redefToNew.get(p);
    us.emitMsg("Hello");
  }
}

public class UtilNew{
  public static Hashtable<Util, UtilNew> redefToNew;
  public static Hashtable<UtilNew, Util> newToRedef;
  public void emitMsg(String str){...}
}

public class Util{
  Util(){
    UtilNew us = new UtilNew();
    UtilNew.redefToNew.put(this, us);
    UtilNew.newToRedef.put(us, this);
  }
}
```

(c) Our generated hotfix.

Figure 4.2: An example illustrating a developer patch that adds a method to a class and the hotfix that our patch adapter generates by moving the added method to a new class and replacing its monomorphic reference.

solved at runtime through dynamic dispatch. To handle this case, HOTFIXER constructs a runtime check that uses the Java `instanceof` operator. The check maps the runtime type of the call receiver `p` to the new class `Child` that hosts the added method (Line 197). To discover if a method call may resolve to a moved method, HOTFIXER uses the underlying call graph in SOOT_HOTFIXER_VERSION. By default, SOOT_HOTFIXER_VERSION builds a call graph using Class Hierarchy Analysis (CHA) [18]. Since the CHA call graph is conservative, if HOTFIXER has copied all possible targets of the method call into new classes, it will construct a map between each possible runtime type and the appropriate method call. If not all targets are for methods that HOTFIXER has moved, the runtime check makes the original method reference as the default case (Line 200).

The patch adapter generates its runtime checks in a child to parent ordering. Because Java's `instanceof` operator succeeds on an object of a (subclass of) class, building the checks in a child to parent order allows the patch adapter to mimic the way that dynamic dispatch determines the target of the call. The resulting, sorted set of conditional method calls enables the JVM to handle the resolution of the type and corresponding method call as it normally would. Similar to how HOTFIXER handles instance field addition, for added instance methods, HOTFIXER includes in each runtime check a lookup of the appropriate invocation object to use for the adjusted method call (Line 198).

Once HOTFIXER moves a method into the appropriate new class, it transforms references in the moved method that use the `this` keyword in Java. In the developer patch, both fields and methods accessed by `this` are located in the redefinition class. However, a moved method that contains any reference to a field or method that originally (i.e., before the patch) existed in the class now falsely refers to non-existent fields and methods. Our patch adapter detects these references, and uses the reverse hashtable to fetch the correct object reference that accesses the pre-existing fields and methods. Figure 4.4 shows an example of this scenario. The figure is an extension of Figure 4.3 and, therefore, does not depict the code for `User` and `Util`. Compared to Figure 4.3, the addition to Figure 4.4 is Line 234, where a use of the `this` keyword

```
public class User{
  public void main(){
    Util p;
    if(...){
      p = new Util();
    }else{
      p = new Child();
    }
    // -> Util.emitMsg()
    p.emitMsg("Hello");
  }
}

public class Util{
  public void emitMsg(){...}
}

public class Child extends Util{

}
```

(a) The original code.

```
public class User{
  public void main(){
    Util p;
    if(...){
      p = new Util();
    }else{
      p = new Child();
    }
    //polymorphic call
    p.emitMsg("Hello");
  }
}

public class Util{
  public void emitMsg(){...}
}

public class Child extends Util{
  public void emitMsg(){...}
}
```

(b) The developer patch.

```
public class User{
  public void main(){
    Util p;
    if(...){ p = new Util(); }
    else{ p = new Child(); }

    if(p instanceof Child){
      ChildNew us = ChildNew.redefToNew.get(p);
      us.emitMsg("Hello");
    }else{
      p.emitMsg("Hello");
    }
  }
}

public class ChildNew{
  public static Hashtable<Child, ChildNew> redefToNew;
  public static Hashtable<ChildNew, Child> newToRedef;
  public void emitMsg(){...}
}

public class Child extends Util{
  Child(){
   ChildNew us = new ChildNew();
   ChildNew.redefToNew.put(this, us);
   ChildNew.newToRedef.put(us, this);
  }
}
```

(c) Our generated hotfix.

Figure 4.3: An example illustrating a developer patch that adds a method to a class and the hotfix that our patch adapter generates by moving the added method to a new class and updating its polymorphic references.

```
219 public class Util{ }
220
221 public class Child extends Util{
222   public String f;
223
224
225
226
227 }
```

(a) The original code.

```
228 public class Util{ }
229
230 public class Child extends Util{
231   public String f;
232
233   public void emitMsg(){
234     System.out.println(this.f);
235   }
236 }
```

(b) The developer patch.

```
237 public class ChildNew{
238   public static Hashtable<Child, ChildNew> redefToNew;
239   public static Hashtable<ChildNew, Child> newToRedef;
240
241   public void emitMsg(){
242     Child p = ChildNew.newToRedef.get(this);
243     System.out.println(p.f);
244   }
245 }
246
247 public class Child extends Util{
248   public String f;
249   Child(){
250     ChildNew us = new ChildNew();
251     ChildNew.redefToNew.put(this, us);
252     ChildNew.newToRedef.put(us, this);
253   }
254 }
```

(c) Our generated hotfix.

Figure 4.4: An example illustrating a developer patch where an added method refers to an existing field using `this` and the hotfix that our patch adapter generates by correctly updating that reference using its reverse hashtable.

requires an adjustment from our patch adapter. HOTFIXER performs this adjustment by looking up the correct object reference used for the field access (Lines 242–243). In this work we do not consider accesses to fields or calls to methods via reflection.

Chapter 5

Phase II: Hotfixing

When CogniCrypt_Hotfixer_Server has a hotfix to deliver, CogniCrypt_Hotfixer_Server connects to HotfixerAgent. For the example in Figure 3.3, the hotfix consists of `Util`. When HotfixerAgent performs a redefinition, it guarantees that subsequent invocations of methods that have been replaced will use the redefined implementation. Despite this guarantee, if there are some methods currently running, and a redefinition of that method occurs, the old implementation of the method will continue execution until it is complete. Hotfixer provides this guarantee only on any future runs of that method that the new implementation will execute.

There is one more limitation of the redefinition mechanism that cannot be solved by the patch adapter; hotfixing applies to a single instance of the running application. In a client-server environment, there is at least one server and many clients. When there are multiple actors involved in an interaction, they must coordinate patch changes among each actor. For example, if a client sends a server message encrypted with RSA [19], the server must know that RSA was used, so that it can decrypt the message accordingly. Thus, if the Hotfixer patch changes an encryption algorithm used by the client, and we do not patch the server, the server will not properly decrypt the message that it received.

To address the limitation above, and also those limitations discussed in Section 3.2, we have devised four categories that a patch may fall into, when it becomes a hotfix:

1. **Natural Hotfix:** If every class in a patch can be used for redefinition without causing errors, we refer to these classes as redefinition compliant and the entire patch is a natural hotfix.

2. **Adjusted Hotfix:** If some classes in the patch may cause an error when used for redefinition due to the checks that the Instrumentation API performs before performing redefinition, then HOTFIXER must adjust these classes before using them in the hotfix. To deal with redefining non-compliant classes, we have devised a patch adaptation tool that takes a patch, detects the components in each class that are non-compliant, and then adapts those components so that the patch can be used as a hotfix, i.e., it expresses the intended semantics of the fix and also only contains redefinition compliant classes. The patch adaptation tool presented in Section 4 addresses this category of patches.

3. **JVM-Assisted Hotfix:** If some classes in the patch are redefinition non-compliant and cannot be adjusted by our patch adapter, HOTFIXER may require assistance from the JVM to apply this patch as a hotfix.

4. **Multi-JVM Hotfix:** If the semantics of the changes in the patch require modifying multiple running applications, we classify this as a multi-JVM hotfix.

The Natural Hotfix and Adjusted Hotfix categories are mutually exclusive, as well as Natural Hotfix and JVM-Assisted Hotfix categories. It would be possible, however, to have a Natural Hotfix that falls into the Multi-JVM Hotfix category. For this work, we focused on addressing the Natural Hotfix and Adjusted Hotfix categories. Natural hotfixes do not require any special handling, and simply go through the process described in Figure 3.1. However, adjustment are necessary when the patch falls into the Adjusted Hotfix category, as previously discussed in Section 4.

Chapter 6

Evaluation

In this section, we evaluate the ability of HOTFIXER to fix misuses, as well as the effect that HOTFIXER has on program semantics and program performance. Our baseline for comparison is a manually applied develop-time patch strategy. Our evaluation aims at answering the following research questions:

RQ1: How successful is HOTFIXER at fixing crypto API misuses?

RQ2: Does HOTFIXER alter the semantics of a running application?

RQ3: How does HOTFIXER affect application performance?

6.1 Experimental Setup

To answer our research questions, we utilize two datasets: one constructed by Sharmin et al. [20] comprising of 144 relevant misuses across 103 benchmarks and the second is constructed by Wickert et al. [21] comprising of 44 crypto API misuses across 7 open-source projects. We refer to the datasets as CryptoGuard micro-benchmark and Wickert benchmark, respectively.

We ran all of our experiments in a Docker container [22] using Docker version 19.03.5, on an x86_64 Ubuntu 18.04.4 machine with two 2.4GHz AMD EPYC 7351 16-Core processors. We ran all experiments on OPENJ9_-HOTFIXER_VERSION, which is based on the OpenJDK build jdk8u252-b09. Table 6.1 lists the SHAs for the corresponding commits for each base component in HOTFIXER.

Component	**Based On**	**SHA**
OpenJ9_Hotfixer_Version	Eclipse OMR [23, 24]	`d4365f371ce896bead71bc601cbdb53cc35ab47b`
OpenJ9_Hotfixer_Version	Eclipse OpenJ9 [14]	`05fa2d3611f757a1ca7bd45d7312f99dd60403cc`
OpenJ9_Hotfixer_Version	OpenJDK [25]	`8a27fe4d476a8ca2263c4c031264e204d2e3d259`
CogniCrypt_Hotfixer_Server	CogniCrypt [2]	`97b22050519bd5a5dbc3c364f5fadb981d043fa6`
Soot_Hotfixer_Version	Soot [15]	`ca72d20db7921415ecd9310f3c5208f20c669991`

Table 6.1: The components in Hotfixer and their corresponding commit hashes.

6.2 Data Processing

6.2.1 CryptoGuard micro-benchmark

To answer our research questions, we run Hotfixer on 103 benchmarks from CryptoGuard micro-benchmark. As CogniCrypt_Hotfixer_Server is an extension of CogniCrypt, first we had to assure that CogniCrypt can detect misuses in all benchmarks in CryptoGuard micro-benchmark, otherwise Hotfixer will not be able to also detect misuses and therefore perform hotfixing. Although the version of CryptoGuard micro-benchmark that we accessed contains 195 benchmarks, CogniCrypt does not detect misuses in all benchmarks in the suite. From the 92 benchmarks that do not contain any misuse, 14 of them serve the purpose of detecting true negatives (or false positives) in the findings of crypto API misuse detection tools. This means these 14 benchmarks contain correct uses of crypto APIs, which we can utilize as patches for Hotfixer. We adapt each of these general patch classes to each individual benchmark, as we require a corresponding specific patch for each benchmark for Hotfixer to run on. We changed the provided patches in 7 benchmarks that store passwords in `String` objects, which CogniCrypt correctly flags as insecure. We are left with 103 benchmarks from CryptoGuard micro-benchmark to test Hotfixer once we omit both the patch classes and classes where no misuse is discovered. We leave it to future work to utilize CryptoGuard [5], the tool that the benchmark was designed around, to enable additional initial misuse detection.

6.2.2 Wickert benchmark

To further evaluate Hotfixer, we obtain the exact versions of all projects in the dataset that was studied in the work by Wickert et al. [21]; we refer to this as original version. For this benchmark, the authors had manually crafted fixes for each misuse, and contributed each fix in a code snippet separate from the corresponding project. For each benchmark program, we created dev-patch version by integrating each provided correct usage code snippet into the application at the location indicated by the dataset to contain the misuse. The majority of the contributed corrections (34/44) do not classify as corrections in isolation. For example, to correctly accomplish an encryption task, the same initialization vector must be used in both encryption and decryption. Encrypted text that is decrypted with a different cipher or initialization vector is simply invalid. To correct these runtime errors in the original fixes, we merged the related corrections into a single experimental configuration. To integrate these merged corrections into the project's code base, we followed two basic principles. For single statement-level changes, we generally applied the correction as that single-statement change. However, for changes (single or multi statement) that have an interprocedural impact, we applied the changes in new, added methods; in this case, we additionally added invocations for these added methods in pre-existing instance or static methods. Column 1 of Table 6.2 summarizes the results of our consolidation efforts. We maintain the original schema for identifying the fixes, and in cases where we have merged the fixes, we simply label them as misuseXandY, where X and Y are the components of their original fixes. This consolidation leaves us with a total of 27 misuses to perform our evaluation on.

6.2.3 Crypto API Misuse Detection

Lastly, in preparation for answering our research questions, we assured that Hotfixer is as effective at detecting crypto API misuses as standalone CogniCrypt, despite performing static analysis at runtime and utilizing classes from the SCC. Across all of CryptoGuard micro-benchmark CogniCrypt_-

HOTFIXER_SERVER finds the same misuses that CogniCrypt detects, with the exception of 6 benchmarks, where COGNICRYPT_HOTFIXER_SERVER detects one extra misuse, which we discuss in further detail in Section 6.3. For the 27 misuses in Wickert benchmark dataset, the findings of COGNICRYPT_-HOTFIXER_SERVER are identical to CogniCrypt when run standalone on the ORIGINAL version of each project. Additionally, we manually verified for each experiment that the initial analysis that COGNICRYPT_HOTFIXER_SERVER performs is successful in utilizing the class under test from the SCC.

6.3 How successful is Hotfixer at fixing crypto API misuses? (RQ1)

6.3.1 CryptoGuard micro-benchmark Results

To confirm that each misuse is fixed after hotfixing, we add (for experimental purposes only) an additional post-hotfix run of CogniCrypt to HOTFIXER. Using this technique, we have verified that HOTFIXER fixes the same misuses that a develop-time patch strategy fixes on the entire CryptoGuard micro-benchmark suite. During this post-hotfix CogniCrypt analysis, HOTFIXER finds an additional 6 misuses compared to CogniCrypt run on develop-time patch. However, these extra misuses are also detected by COGNICRYPT_-HOTFIXER_SERVER in the application before hotfixing. In all 6 cases, the detected misuse is a constraint error with an identical error message. We observe that this error is reported in association with an extra upcast statement that is present in the class during hotfixing, compared to the class when used in develop-time patch. The extra upcast is found between the return value of a call to `java.security.KeyPair.getPublic()`, and the argument to `javax.crypto.Cipher.init(intopmode,java.security.Keykey)`. We observe that this upcast statement is only present in the analyzed class when it is either loaded from the SCC or after it has gone through our patch adapter. Further investigation has showed that this extra upcast is generated by Soot to represent a stack location. In this case, the generated local has the declared type (`java.security.Key`) of the argument to `javax.crypto.Cipher.`

Table 6.2: The results of running HOTFIXER on Wickert benchmark.

Benchmark	Misuse	CogniCrypt Misuse Type	Fixed?	# Introduced Misuses	# Passing Tests	# Failed Tests
HA-BRIDGE	1and5	Constraint	✓	0	221	0
	2and7	Constraint	✓	0	213	0
	3and8	Required Predicate	✓	0	182	0
	4and6	Forbidden Method	✓	0	795	0
INSTAGRAM4J	1	Required Predicate	✓	0	8,658	0
JEESUITE-LIBS	1and4	Required Predicate	✓	0	288	0
	2and5	Constraint	✓	0	21	0
	3	Required Predicate	✓	0	21	0
	6and7	Constraint	✓	0	129	0
	8	Constraint	✓	0	8,283	0
	9	Constraint	✓	0	7,722	0
NETTYGAMESERVER	1	Constraint	✓	0	1,492	0
	2and3	Constraint	✓	0	1	2
	4	Constraint	✓	0	8,038	0
SMART	1and6	Required Predicate	✓	0	2	0
	2and5	Required Predicate	✓	0	16	0
	3	Required Predicate	✓	0	243	0
	4and7	Required Predicate	✓	0	16	0
	8	Constraint	✓	0	248	0
WHATSMARS	1and3	Constraint	✓	0	235	5
	2and4	Required Predicate	✓	0	226	0
	5and9	Constraint	✓	0	245	0
	6and11	Required Predicate	✓	0	2,518	0
	7and12	Required Predicate	✓	0	253	0
	8and10	Required Predicate	✓	0	259	0
	13	Constraint	✓	0	159	0
DRAGONITE-JAVA	1	Required Predicate	✓	0	2	0
			Total	0	40,486	7

`init(intopmode,java.security.Keykey)`. Since the misuse is present before and after hotfixing, we do not consider it to be an erroneous behaviour caused by HOTFIXER.

6.3.2 Wickert benchmark Results

Table 6.2 depicts the result of our experiment. The first column of the table shows the 7 project names comprising the Wickert benchmark. The second column depicts the exact benchmarks in the entire Wickert benchmark dataset, and the third column describes the misuse type that was addressed in that benchmark. The misuse types described in the third column originate from the categorization that CogniCrypt uses. As shown in the fourth column, across all of Wickert benchmark, HOTFIXER fixes the same misuses as the develop-time patch strategy. HOTFIXER also does not introduce any additional misuses compared to the develop-time patch strategy, as seen in the fifth column of Table 6.2. We explain the results presented for the remaining two columns of Table 6.2 shortly, in Section 6.4.3.

Hotfixer fixes crypto API misuses in all of the benchmarks in CryptoGuard micro-benchmark and Wickert benchmark, without introducing any additional misuses. Hotfixing is as viable of a solution as software patching, with respect to fixing misuses.

6.4 Does Hotfixer alter the semantics of a running application? (RQ2)

Similar to prior software patching work [9, 26, 27, 28], we use regression testing to assess whether Hotfixer alters the intended original functionality of the application after applying our generated hotfix.

6.4.1 Regression Test Setup

For each patched benchmark in CryptoGuard micro-benchmark and the devpatch version of each project in Wickert benchmark, we generated a set of regression tests using Randoop, an automatic test generation framework [29]. To avoid generating irrelevant tests, we provided Randoop with the classes that we know contain the misuses. When Randoop is provided with a set of names of classes to generate tests for, it only uses those classes in the tests. We configured Randoop with a 60-second time limit for the test generation. This limit is more than double the typical suggested time limit [30]. For CryptoGuard micro-benchmark, Randoop generated a total of 78488 tests (min: 2, max: 5,000, median: 6), whereas for Wickert benchmark, Randoop generated a total of 40493 tests (min: 2, max: 8,658, median: 240).

Randoop only generates tests for methods accessible to the test suite (i.e., public methods). In the cases where a misuse is located in a non-public method, we changed the visibility of the method to assure that Randoop could test it. Because Randoop generates tests for the exact set of methods provided, and only that set, if we change a method from non-public to public, we can see that this does not alter the effects of that exact method under test. This change does affect the entire application, however, it is an unavoidable change that we must make to allow for regression test generation.

To answer RQ2, we create 2 different setups from the Randoop tests, for CryptoGuard micro-benchmark and Wickert benchmark. For CryptoGuard micro-benchmark, we assess the output of one full iteration of a regression test suite once the redefinition event has clearly taken place. To ensure that the redefinition event has occurred (and completed) before the regression tests run, we create a setup method in the setup class of each regression test suite. Normally Randoop generates an empty setup class that serves only the purpose of assuring that all classes containing tests will run, however we take advantage of Junit `BeforeClass` [31] annotation to assure that our setup method will run first. The setup consists of: (1) an invocation of the method(s) relevant to the experiment, such that we perform a logical task, for example an encryption followed by a decryption, (2) a loop that ensures that those methods of interest are to be compiled, (3) a pause until the HotfixerAgent has completed the redefinition event, and (4) a repetition of the same task that was initially performed in the method. It is after this setup that the regression tests begin to run.

For Wickert benchmark we do not run the test suite after inducing the redefinition event, instead, we continuously iterate the test suite for some large number of iterations (i.e., a test window), and then observe whether test failures occur after redefinition occurs. Each test window is thus comprised of: (1) original subwindow: some number of iterations where the original application executes, (2) a redefinition event, and (3) hotfixed subwindow: some number of iterations where the hotfixed application executes. The number of iterations required to allow for a sufficient test window is variable between each benchmark. We determine these values experimentally but have opted for not presenting them, because the exact values are not relevant to answering RQ2.

Due to several factors, we are unable to utilize the previously mentioned setup to answer RQ2 for NettyGameServer misuse2and3 and smart misuse1and6. These two benchmarks take more than three minutes to execute each testsuite iteration. Additionally, JUnit's reporting behaviour, by default, is only able to report test outcomes at the end of a complete test run, i.e., the full test window duration. In other words, we can only determine if Hotfixer affects application behaviour after the full test window has completed. Lastly, redefinition points are non-deterministic (with respect to testsuite iteration number) due to the behaviours of the JIT and JVM which vary across application executions; therefore we cannot pre-emptively pick a number of test iterations for the test window size to accommodate the especially long patch runtime. Due to all of these factors, for these two benchmarks only, we use the setup described for CryptoGuard micro-benchmark to answer RQ2.

Furthermore, Randoop generated 161 flaky tests for each of (ha-bridge misuse3and8, jeesuite-libs misuse1and4, jeesuite-libs misuse6and7, and, whatsmars misuse13). We observed that those tests fail at either both original subwindow and the hotfixed subwindow, or just in the hotfixed subwindow. In all 4 benchmarks, we are able to replicate the presence of failures in the iterated devpatch version version of the application. Unfortunately, in all 4 cases, the number of failures is non-constant across multiple runs of the application. We refer to such tests that fail on some runs and not others as *flaky*, similar to many other research works and software professionals [32, 33]. Flaky tests confound our ability to determine if Hotfixer introduces errors into the application compared to software patching, therefore, we omit the flaky tests from the testsuites of these 4 benchmarks.

Finally, the last testsuite size adjustment that we performed was in benchmark smart misuse4and7. In this case, Randoop originally generated only 4 tests in the testsuite, resulting in an extremely short runtime of each iteration (1ms or less). As we measure our iteration runtimes at the precision of milliseconds, any variation in a sample of iteration times of less than 1ms results in an inflated coefficient of variation (COV), which is the ratio of standard deviation to the mean. To make it easier and more meaningful to compare both the COV and throughput of this experiment to the baseline, we replicated the testsuite such that each iteration now contains those original 4 tests duplicated 4 times, and measured that instead.

```
String str0 = cryptoGuardBench.STATIC_FINAL_FIELD;
org.junit.Assert.assertTrue(str0.equals("abcde"));
```

Figure 6.1: An example illustrating the cause of CryptoGuard micro-benchmark test failures.

6.4.2 CryptoGuard micro-benchmark Results

We observed two test failures across 78488 tests, in StaticInitializationVectorABICase2 and StaticSaltsABICase2. Both of the failures observed were caused by tests that check against values of public static final fields of the redefined class. Figure 6.1 shows an example of such a check. In these two benchmarks, the patch modifies the value of a static final field. As explained in Section 5, static initializers are not rerun during redefinition events, therefore to observe the value changes, our patch adapter must redefine that static variable (as long as it is simply static and not static final). An important complication to this is that when `javac` compiles the tests alongside the original benchmark class, the static final field's original constant value is propagated to all use locations. In the bytecode corresponding to Figure 6.1, the reference to `str0` is replaced by the constant value held in the original `cryptoGuardBench.STATIC_FINAL_FIELD`. Because the tests were generated over the patched version of the benchmark, the value that the test checks against (in Figure 6.1 this is `"abcde"`) is the static final field value of the patched class. The constant propagation optimization performed by javac guarantees that the test will fail, and currently our patch adapter cannot reverse the constant propagation that `javac` does in this scenario, because this testcase falls into the JVM-Assisted Hotfix category.

6.4.3 Wickert benchmark Results

We ran a cumulative total of 40493 tests, across all projects, with the exact number of tests for each setup shown in the sixth column of Table 6.2. As the sixth column of Table 6.2 shows, all tests in all experiments pass, except for in 2 benchmarks. In the first case we observe two test failures in NettyGameServer misuse2and3. These test failures are due to testing a result of encryption; in this benchmark the patch adds a random component to an encryption task, and these particular tests assert over the result of encryption. Therefore, these tests will fail on every run, in both DEVPATCH version and in HOTFIXER, i.e., the failures are not caused by HOTFIXER.

The second case occurs in WHATSMARS MISUSE1AND3, where we observe 5 test failures caused by HOTFIXER. All of the failures occur for the same reason; in WHATSMARS MISUSE1AND3, the patch introduces a static field that did not previously exist in the application. That static field is only initialized at specific points in the application, and in this case 5 tests attempted to access that state before it had become initialized. This testcase also falls into the JVM-Assisted Hotfix category because it requires further JVM intervention to determine application points where performing the hotfix guarantees to avoid all conflict with the logical state transitions during all tasks.

In 15 benchmarks, some expected test failures occur during the original subwindow; failures are expected due to the previously mentioned fact that the testsuites are generated for the DEVPATCH version of the benchmarks, not the ORIGINAL version. To answer RQ2, we need to observe whether all failures in test window are expected, i.e., the behaviour of the application during test window is equivalent to expected behaviour of a combination of ORIGINAL version (over an original subwindow duration) and DEVPATCH version (over an hotfixed subwindow duration). We track the exact duration of each original subwindow in terms of number of full testsuite iterations, plus one partial iteration, where some number of tests from the latest iteration have executed. We then run ORIGINAL version over the testsuite for the original subwindow duration to determine the exact number of expected failures. In 14 of those 15 benchmarks, we observe that the the number of failures reported in HOTFIXER test window exactly matches that of the original application, i.e., HOTFIXER does not cause these failures. The last 1 of those 15 benchmarks is the case we previously mentioned, WHATSMARS MISUSE1AND3, where HOTFIXER does cause the failures.

Hotfixer preserves program semantics in 98% of the analyzed benchmarks. The other 2% belong to a patch category that Hotfixer currently does not support. Hotfixing requires additional consideration compared to software patching, when considering it as a feasible solution to crypto API misuses in applications running in servers.

6.5 How does Hotfixer affect application performance? (RQ3)

6.5.1 Performance Test Setup

To measure the application performance overhead of using Hotfixer, we focus on 4 metrics: (1) runtime of patch adapter, (2) duration until the JIT has sufficiently recovered from the redefinition event in Hotfixer, (3) relative throughput performance of Hotfixer compared to develop-time patch, and, (4) recovery profile of application. We answer RQ3 for Wickert benchmark only, because it consists of real-world applications where performance is a relevant aspect of application execution. To answer RQ3 we use the same setup for RQ2.

To collect patch adapter running time, we measure the duration between the point in time where CogniCrypt_Hotfixer_Server receives the analysis request for the class that causes redefinition and the point at which the JVM observes that the redefinition has taken place. To collect recovery duration, we first define sufficient JIT recovery by inspecting the activities of the JIT during execution; as a result of the redefinition event, the JIT will typically experience an increase in the number of compilations that it must perform. We define sufficiently recovered as the subsequent testsuite iteration after which the size of the compilation request queue for the JIT has sustained at 2 requests or less, for a duration of 2 seconds. To compare throughput performance, we measure 10 contiguous testsuite execution times, after this recovery point has been observed.

For each benchmark application, we begin our sample from the same recovery point (iteration number) for the baseline as was used for the test window recovery point. This approach maintains consistency across our experiments.

The only exception is 2 long-running benchmarks (NettyGameServer misuse2and3 and smart misuse1and6), where it would take upwards of 4,000 hours to reach the recovered iteration number in the patch; for these two benchmarks we define our baseline runtimes using 70 and 100 iterations of the testsuite, respectively, and sample a window of 10 from iteration 55 and 85, respectively, because these represent a stable point in the application which is the most fair window for comparison.

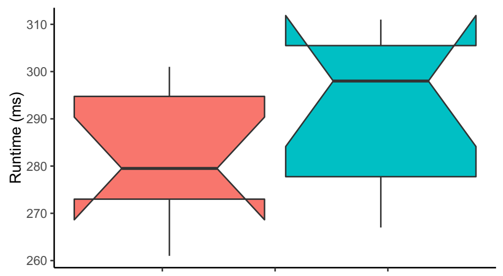
To inspect the recovery profile of the application when Hotfixer is run we observe a window of 30 iterations of each testsuite for each configuration. In the 30 iteration window we utilize the first 9 iterations at the point before the HotfixerAgent has observed the redefinition event, the 10-12th iteration range are where the event occurs and then the remaining iterations depict the immediate runtimes of the recovering system. We additionally plot, in Figure 6.4, the runtimes of the develop-time patch baseline over that iteration window as well, again, with exception of the long-running benchmarks NettyGameServer misuse2and3 and smart misuse1and6, where we take a sample window starting roughly in the middle of the trial (from iteration 31-60 and 61-90 respectively). In benchmarks instagram4j, dragonite-java, jeesuite misuse8, and smart misuse4and7 we were required to collect the windows from a trial where the OpenJ9_Hotfixer_Version heap size is set to 4g. In jeesuite misuse8 we used a heap size of 6g. We increased the heap limit in these experiments to allow for an observation window devoid of garbage collection events. We are required to avoid garbage collection events in the observation window, as they create an increase in an iteration runtime, which makes it difficult to observe the isolated effects of the redefinition event.

6.5.2 Results

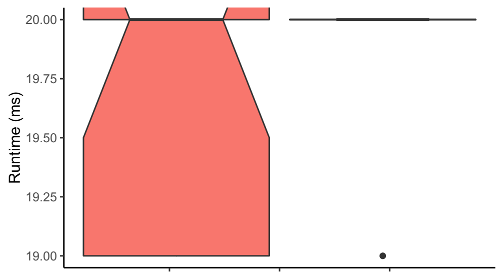
Over the entire Wickert benchmark, the median of the runtime of the patch adapter is 32.8 seconds (min: 23.3 seconds, max: 47.9 seconds, standard deviation: 5.8 seconds). Across the benchmark, we observed an median recovery time of 4.3 seconds (min: 2.1 seconds, max: 128.6 seconds, standard deviation: 28.8 seconds), which represents a 3.6% (min: 0.5, max: 14.3) of the total runtimes across all of Wickert benchmark. We note that 2 recovery times stand out in particular, for NettyGameServer misuse2and3 and smart misuse1and6 with recovery times of 128.6 and 93.0 seconds, respectively. These benchmarks present an outstanding recovery time due to the average runtime of each testsuite iteration being over 550× and 240× respectively, of the runtime of the next longest testsuite runtime from the entire Wickert benchmark. As the runtime of the each iteration is so long, it spreads out the JIT discovery of compilation tasks such that even if other benchmarks take a number of testsuite iterations (which represent some number of times encountering methods affected by the redefinition) to recover, in the case of NettyGameServer misuse2and3 and smart misuse1and6, that period is simply longer.

Figure 6.2 presents the results of our comparison of throughput performance of Hotfixer against develop-time patch, where the notches in the boxplots represent the 95% confidence interval. Each subfigure in Figure 6.2 contains paired boxplots, such that the leftmost boxplot shows the distribution of the runtime of the iterations in our sample window for the baseline develop-time patch strategy, and the rightmost is the distribution for Hotfixer. The line in each boxplot represents the median of the data, and each boxplot's shape describes the distribution of the data points around that median. We performed a Shapiro-Wilk [34] normality test and found that 31 of the 54 total windows represent distributions that are not normal. We additionally create histograms, for each benchmark, for the differences between the runtimes in the Hotfixer window and the baseline window. From these histograms we observe that the differences are not symmetrically distributed around the medians for most of the benchmarks.

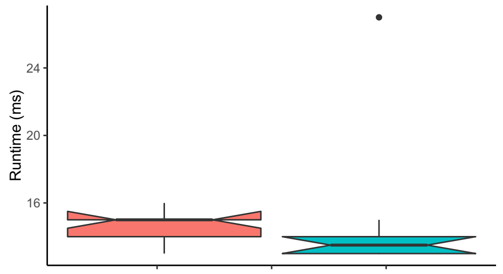
We performed a Paired Sign Test [35, 36], using a significance level of 0.05, to determine whether any difference exists between the runtimes of the iterations in the sample window of Hotfixer compare to the baseline develop-time patch strategy, within each benchmark, i.e., the median of the paired differences is not equal to zero. In the benchmarks ha-bridge misuse2and7, jeesuite misuse3, jeesuite misuse6and7, jeesuite misuse9, NettyGameServer misuse4, smart misuse8, whatsmars misuse6and11,



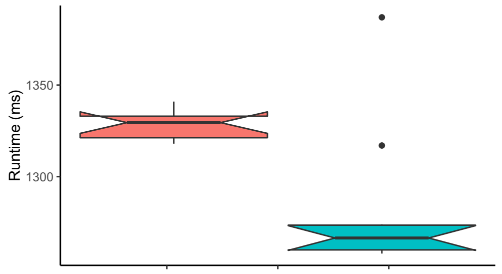
((1)) instagram4j



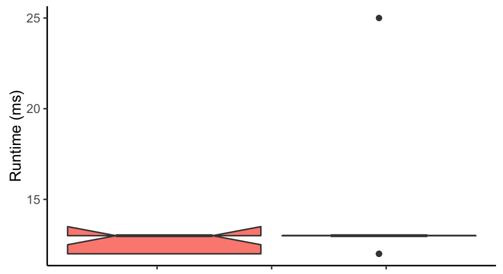
((2)) dragonite-java



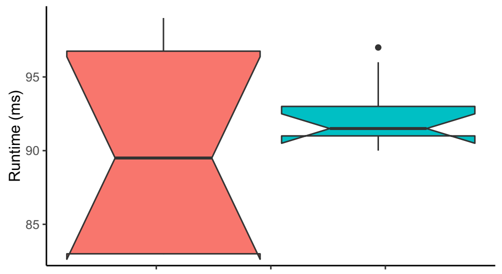
((3)) ha-bridge1and5



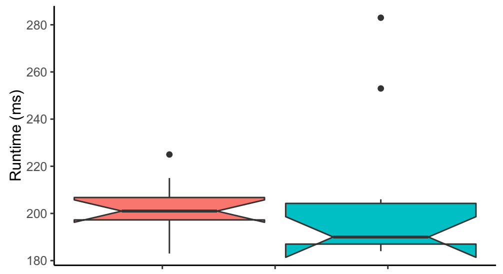
((4)) ha-bridge2and7



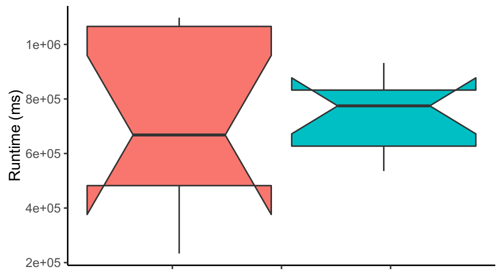
((5)) ha-bridge3and8



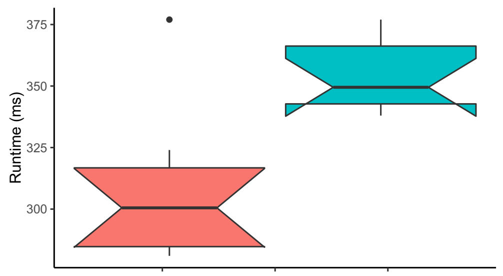
((6)) ha-bridge4and6



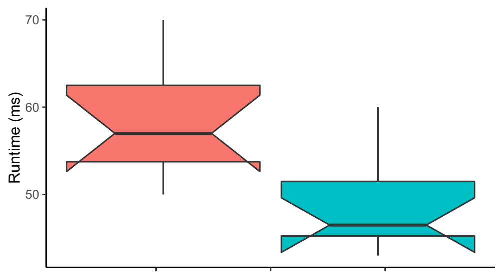
((7)) netty1



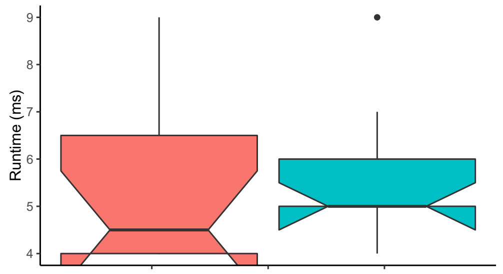
((8)) netty2and3



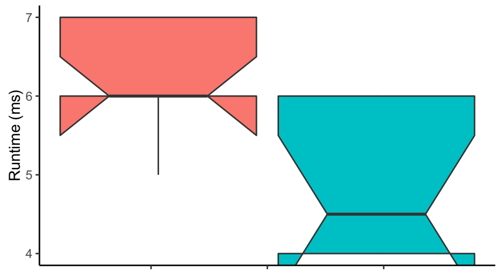
((9)) netty4



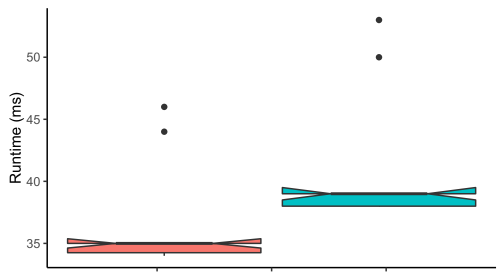
((10)) jeesuite1and4



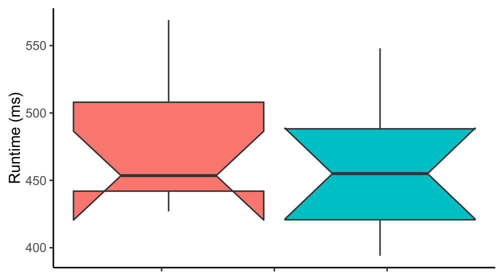
((11)) jeesuite2and5



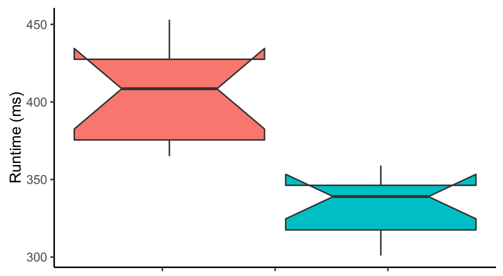
((12)) jeesuite3



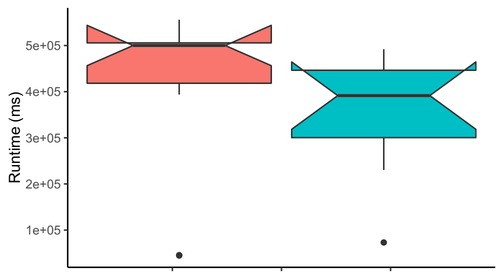
((13)) jeesuite6and7



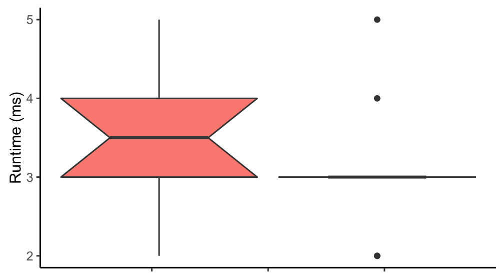
((14)) jeesuite8



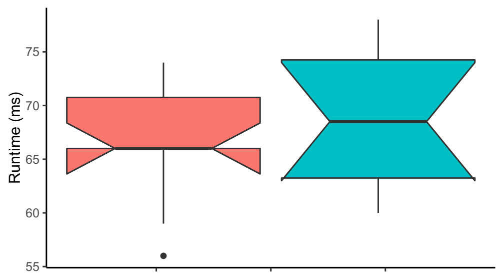
((15)) jeesuite9



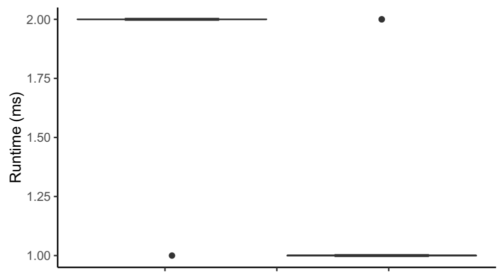
((16)) smart1and6



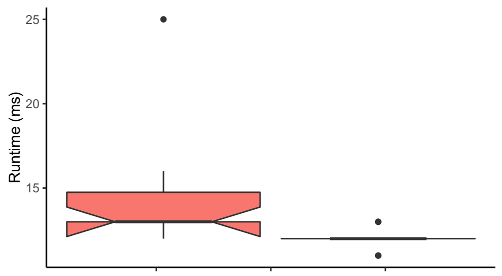
((17)) smart2and5



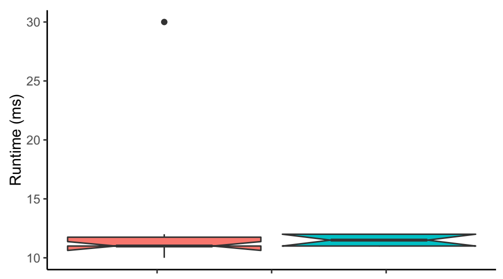
((18)) smart3



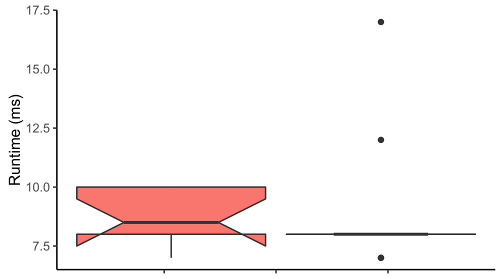
((19)) smart4and7



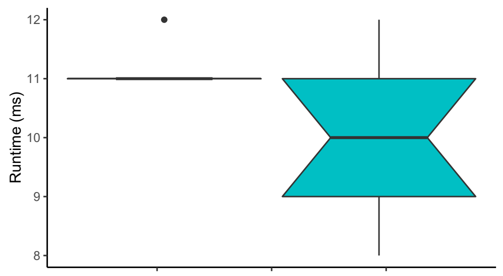
((20)) smart8



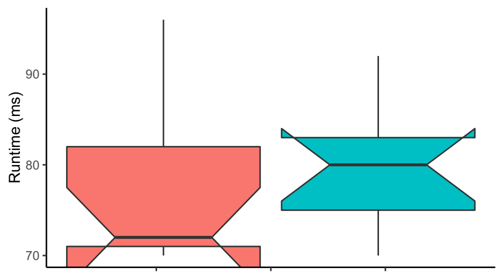
((21)) whatsmars1and3



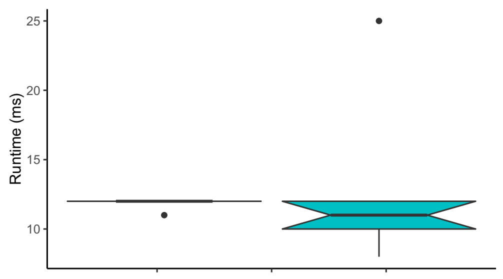
((22)) whatsmars2and4



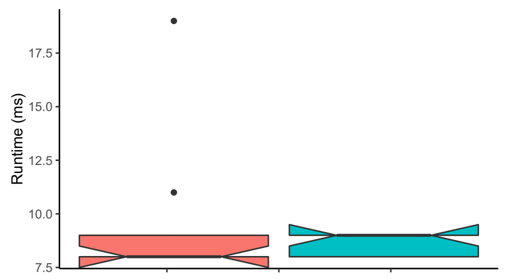
((23)) whatsmars5and9



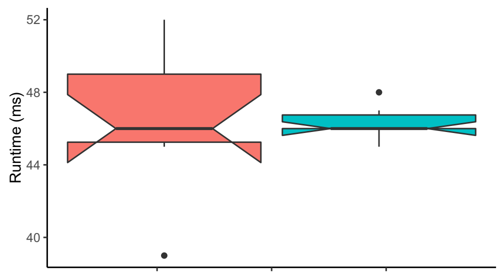
((24)) whatsmars6and11



((25)) whatsmars7and12



((26)) whatsmars8and10



((27)) whatsmars13

Legend:

🟥 Baseline (Software Patch)

🟦 Hotfixer

Figure 6.2: Throughput of Wickert Dataset. Notches in boxplot represent a 95% Confidence Interval.

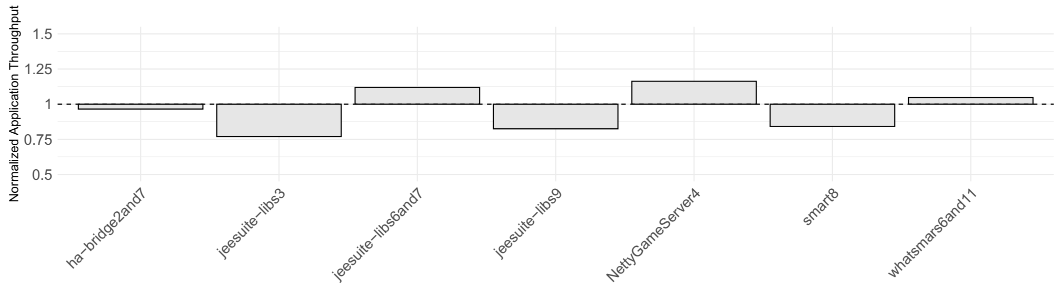


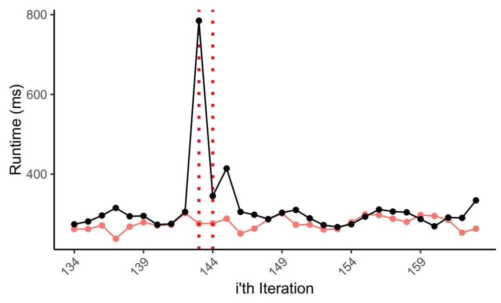
Figure 6.3: The application throughput using HOTFIXER, normalized to that of using develop-time patch strategy, of Wickert benchmark.

we observe that the median of the HOTFIXER window does differ from the median of the baseline window.

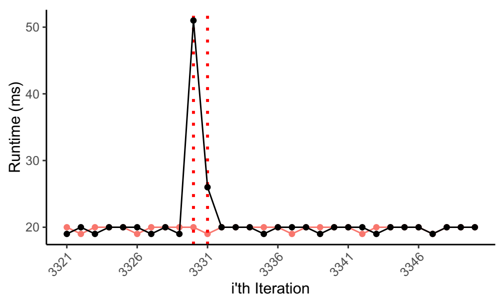
Additionally, across Wickert benchmark, we calculate the relative effect as the geometric mean of the runtime of the iterations in our sample window normalized to the baseline develop-time patch strategy. Figure 6.3 presents the relative effect, as defined above, for the 7 benchmarks where the samples were found to differ. We find that the overall median effect is 0.2% overhead, across all benchmarks, while the overall median of just the 7 benchmarks where the samples are found to be different is a 3.5% speedup. The maximum overhead induced in any benchmark is 16.3% and the maximum speedup induced is 38.4%. Considering only the benchmarks where the medians of the differences were found by the Sign Test to be non-zero, the maximum overhead is 16.3% and the maximum speedup induced is 23.2%.

However, the runtimes presented for our baseline do not encompass the cost that we estimate to be associated with fully stopping and restarting an application to apply develop-time patch. These results, in conjunction with the overall small recovery times suggest to us that HOTFIXER presents a beneficial alternative to develop-time patch.

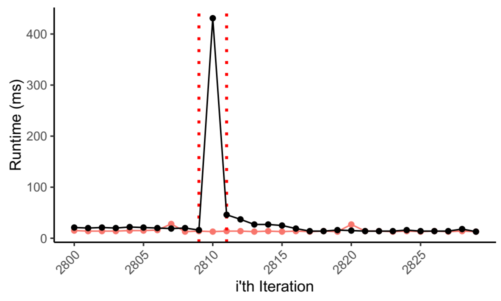
Figure 6.4 shows the results of our comparison of the recovery iteration window of HOTFIXER compared to develop-time patch. In all plots, the black line is the times of HOTFIXER and the red line is the times of develop-time patch. The vertical, red-dotted lines show the iteration (or in some cases a span of two iterations) where the redefinition event took place. In almost all



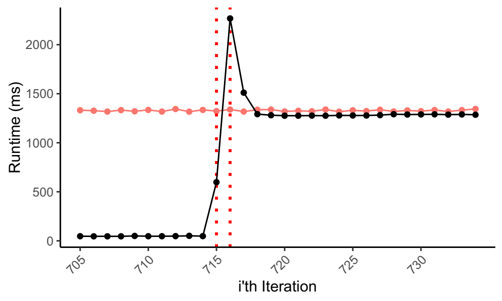
((1)) instagram4j



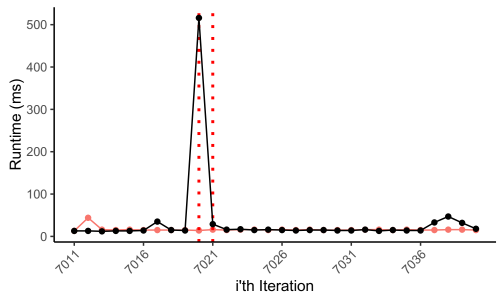
((2)) dragonite-java



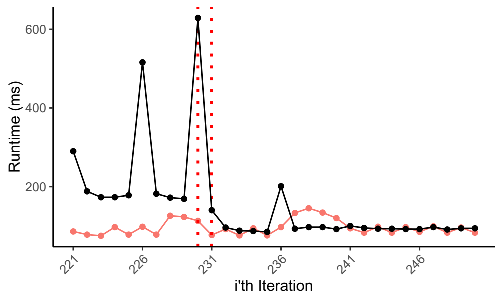
((3)) ha-bridge1and5



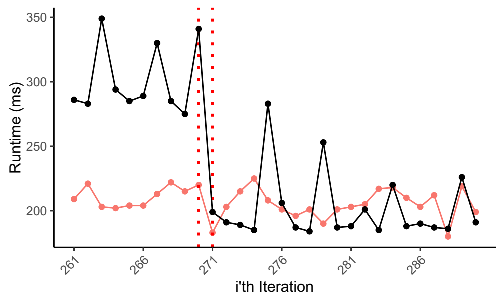
((4)) ha-bridge2and7



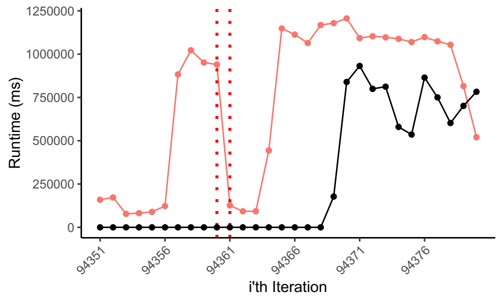
((5)) ha-bridge3and8



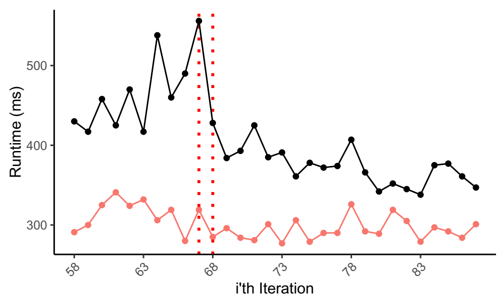
((6)) ha-bridge4and6



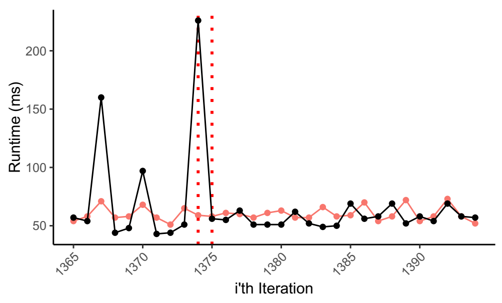
((7)) netty1



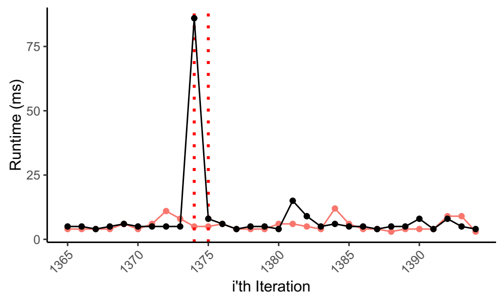
((8)) netty2and3



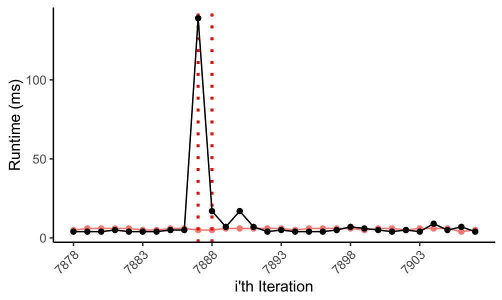
((9)) netty4



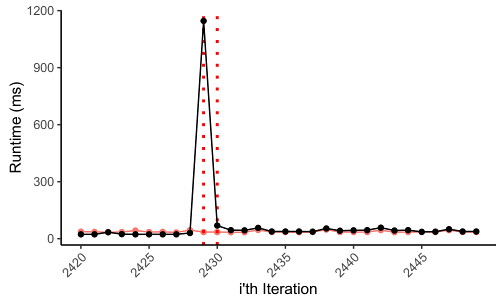
((10)) jee1and4



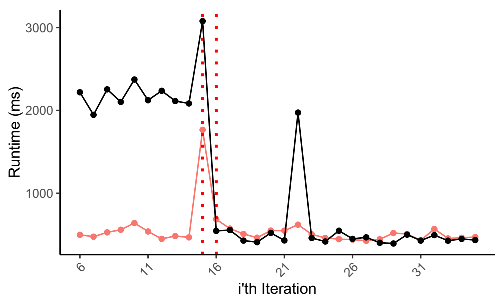
((11)) jee2and5



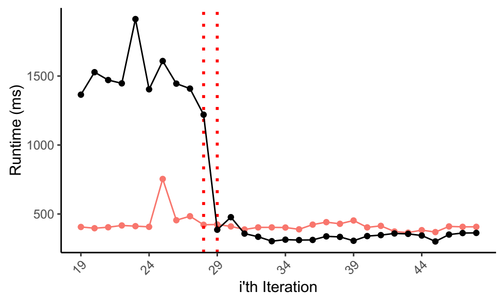
((12)) jee3



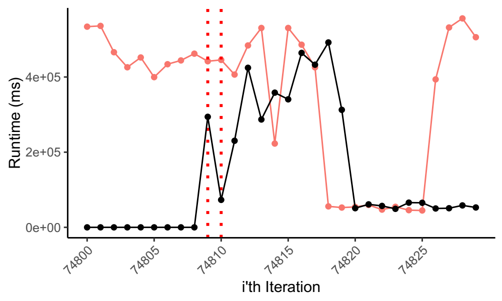
((13)) jee6and7



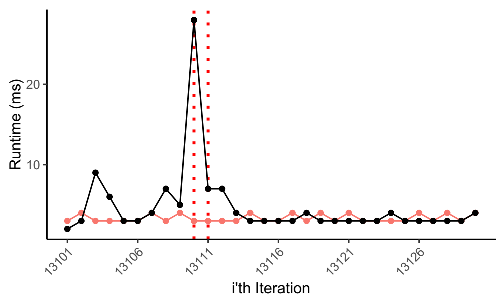
((14)) jee8



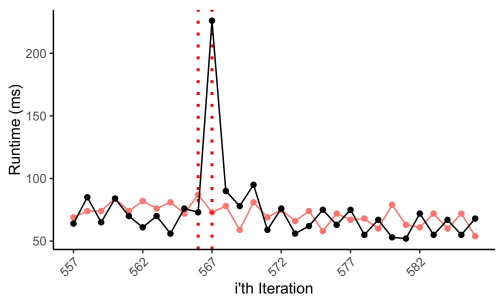
((15)) jee9



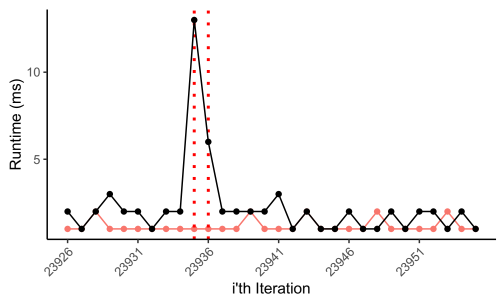
((16)) smart1and6



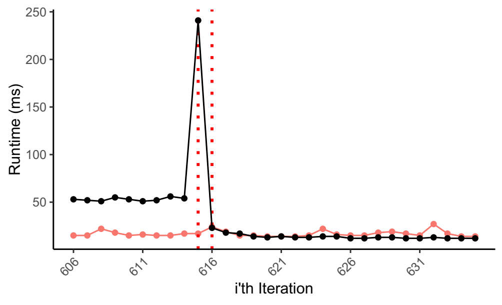
((17)) smart2and5



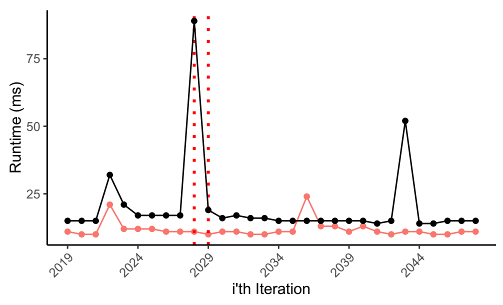
((18)) smart3



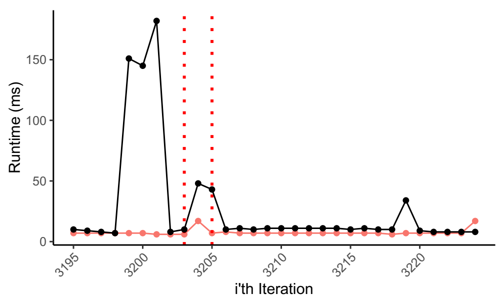
((19)) smart4and7



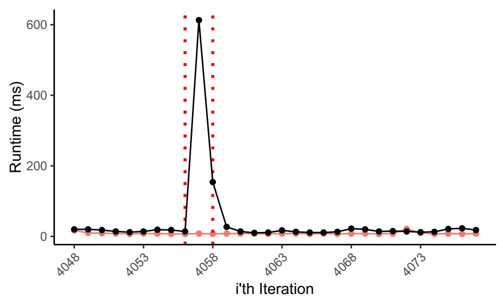
((20)) smart8



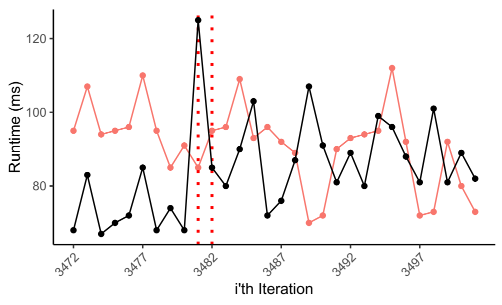
((21)) whatsmars1and3



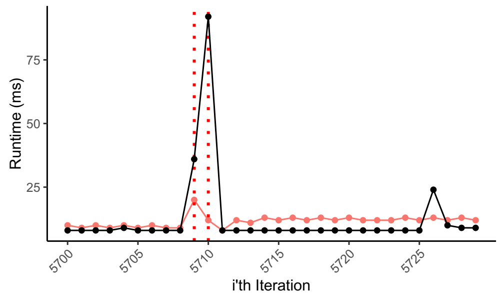
((22)) whatsmars2and4



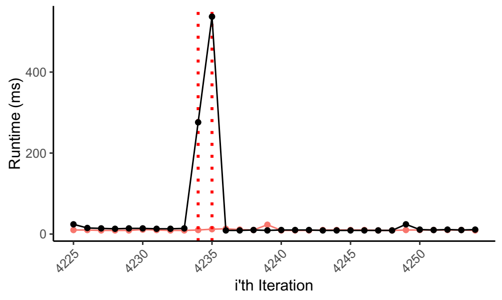
((23)) whatsmars5and9



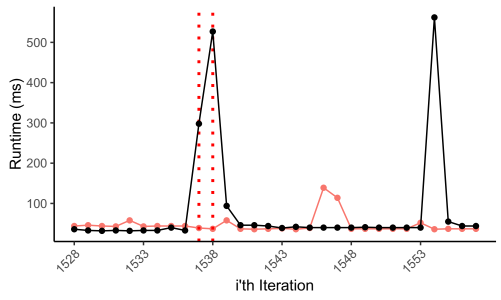
((24)) whatsmars6and11



((25)) whatsmars7and12



((26)) whatsmars8and10



((27)) whatsmars13

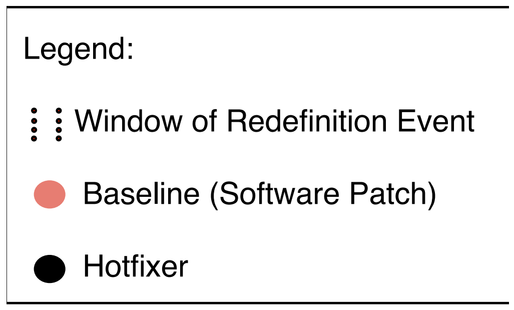


Figure 6.4: Recovery Curves of Wickert Dataset

plots we observe an expected peak during the iteration(s) where the redefinition event takes place, with the exception of NETTY MISUSE2AND3, JEESUITE MISUSE9, SMART MISUSE1AND6, and WHATSMARS MISUSE2AND4. Peaks are expected at the moment when the redefinition can take place in the JVM as it is required to briefly halt the execution of the application so that the definition of a class can be replaced. The delayed peaks in NETTYGAMESERVER MISUSE2AND3 and SMART MISUSE1AND6 occur due to the longer runtimes of each iteration, where it is likely that the JVM cannot immediately perform the necessary pause, because active stack frames of the old definitions of the redefined methods are executing. In JEESUITE MISUSE9 that peak is less prominent due to the fact that the patch simplifies the application, and therefore the peak appears as a smaller runtime than the normal runtimes of the original application. Overall the plots show that in most cases the application runtime stabilizes within the next 15-20 iterations, and that the largest single effect that the redefinition event has on the runtime is when the actual event occurs. As this is a necessary result of performing redefinition, these results show that using HOTFIXER is still beneficial compared to the undeniable application halt and restart that would occur when using develop-time patch.

Our patch adapter has a median runtime of 32.8 seconds and applications in Wickert benchmark experienced a median recovery time of 4.3 seconds. The throughput performance overhead that HOTFIXER induces across all benchmarks is at most 17%. The immediate effect on the application runtime is largely shortlived and non-concerning.

6.6 Threats to Validity

One threat to the conclusion validity of the evaluation of HOTFIXER is that we rely upon regression testing to serve guide for typical application behaviour. It is possible that tests do not encompass all possible behaviours of an application, and in that case we can only assure that HOTFIXER does not alter program behaviour with respect to the tests that we have utilized. We mitigate this threat by using an established test generation tool, Randoop, for our regression tests, and rely upon the established practice of evaluating regression tests, as does much of software patching.

One threat to the internal validity of our evaluation is that we do not control for all external and internal events in our experiments. For example we do not assure that garbage collection does not occur during the execution of the application, which is an event that would cause a lag in application execution time. To reduce the possibility of this effect we did monitor for unexplainable deviations in any one test iteration time over all trials of every setup. This monitoring led to the JVM heap increases, as described in Section 6.5.1.

Lastly, one threat to the external validity of our evaluation is that CogniCrypt_Hotfixer_Server in Hotfixer is built on CogniCrypt. We can only assume that we can extend our work to applications where CogniCrypt is capable to detect a crypto API misuse, i.e., where CogniCrypt does not have any false negatives. To mitigate this threat to validity we rely upon the ongoing work of CogniCrypt that is extending the number of crypto API providers that are covered by CrySL rules, as well as the high precision and recall of CogniCrypt presented in previous evaluations of CogniCrypt [37].

Chapter 7

Related Work

In this section, we discuss prior work that relates to crypto API misuses detection and hotfixing (i.e., the two phases of HOTFIXER). We additionally discuss software patching work that relates to the general principles of Phase II of HOTFIXER.

7.1 Crypto API Misuse Detection

Many static analysis tools have been proposed to aid developers with crypto API misuses detection. Some tools such as CryptoLint [1], MalloDroid [3], and, FixDroid [4] are specific to the Android platform, while other tools such as CryptoGuard [5], and CRYLOGGER [6] detect misuses in both Android and Java apps. HOTFIXER relies on CogniCrypt [2] to detect misuses of crypto APIs. To define a misuse, tools other than CogniCrypt employ pattern-based rules relating to various components of crypto APIs. On the other hand, CogniCrypt uses a specification language called CrySL [37] that enables extensible definitions of misuses that are easy to update. Using CrySL allows HOTFIXER to apply almost all possible techniques for detecting misuses, such as parametric misuse checking, block-listing, typestate analysis and also, predicates over the interaction of crypto API objects; these techniques allow HOTFIXER to check for misuses in 39 classes in the Java Cryptographic Architecture (JCA) API, as well as classes in crypto APIs from other providers such as Tink [38] and BouncyCastle [39].

7.2 Software Patching

Software patching tools provide a foundation for the principles of the implementation and evaluation of Hotfixer: automating the patching process, using static analysis during various stages of the patching process, and applying regression testing to evaluate the correctness of the hotfix.

Automation in software patching can occur at three points in the process: fault detection, patch generation, and patch application (i.e., program repair). The state of the art in various levels of automated end-to-end software patching can be found in the following tools: AutoPAG [40], SapFix [41], SemFix [42], AE [43], and ASAP [44].

AutoPAG [40] uses static analysis to detect out-of-bounds errors. SapFix [41] relies on static analysis to validate the ability of candidate patches to fix null pointer exceptions. Similarly, to perform crypto API misuses detection (Phase I), Hotfixer also uses static analysis. We also use static analysis to evaluate the correctness of Hotfixer (Section 6).

SapFix [41] uses software testing to narrow down candidate patches during a patch selection process. SemFix [42] also uses software testing during the patch generation process, but in their work, tests are used to create a set of constraints that the patch must satisfy as it is built. Unlike those tools, Hotfixer does not use regression testing to generate its hotfix. Instead, we use regression testing to validate the correctness of the hotfix that Hotfixer has generated after applying it to the analyzed program.

7.3 Hotfixing

Recent work has focused on security-related hotfixing in Android apps. For example, AppSealer [45] prevents injection and information leakage attacks in Android apps. AppSealer automatically detects vulnerabilities via a combination of static analysis and runtime instrumentation. If AppSealer detects a potential vulnerability in a running application, the app displays an alert to the user to restart the application. Similar to AppSealer, Hotfixer utilizes static analysis for fault detection, and we also automate portions of the patching process.

InstaGuard [46], is another fully automated hotfix tool that focuses on Android app security. InstaGuard avoids adding any new code to the app, and, instead disallows vulnerability by terminating the app when an insecure condition is detected. Insecure conditions are defined via modular rulesets, called GuardRules. Similar to InstaGuard, HOTFIXER also uses modular specifications, in our case CrySL, for fault detection. Unlike InstaGuard and AppSealer, HOTFIXER enables continued program execution at all times. Moreover, HOTFIXER performs targeted program repair for crypto API misuses specifically, compared to InstaGuard that targets generic security vulnerabilities, and AppSealer that targets component hijacking vulnerabilities.

The most relevant work in the literature is CDRep [8], a tool that automatically repairs crypto API misuses in Android apps. CDRep automatically detects 7 specific scenarios denoting a misuse by utilizing and extending misuse patterns defined by CryptoLint [1]. To fix misuses, CDRep automatically applies a handwritten template patch. FireBugs [9] presents a semi-automated crypto API misuses detection and repair tool for Android apps. Similar to HOTFIXER, FireBugs uses static analysis to detect misuses and regression testing to assess the correctness of the patch. While FireBugs also applies the patch to the running program, it uses Aspect Oriented Programming (AOP) to achieve this, compared to HOTFIXER that utilizes a Java agent. We believe that, compared to AOP, the Java agent interface is a simpler redefinition mechanism to understand and maintain. More importantly, while CDRep and FireBugs use fixed sets of misused patterns, HOTFIXER uses modular CrySL specifications during misuse detection.

Chapter 8

Conclusion

In this paper we present HOTFIXER, a tool to perform automatic crypto API misuse hotfixing at Java application runtime. HOTFIXER offers a beneficial alternative to software patching in scenarios where it is nontrivial to restart servers to adopt software patches, and more importantly, in scenarios where delays in patch deployment lead to windows of vulnerability in an application. We additionally contribute (as a component of HOTFIXER) a novel patch adaptation technique to transform hand written developer patches into hotfixes that a Java agent can use. Our evaluation has shown that HOTFIXER is able to fix all misuses in 95% of the benchmarks in an identical manner to a baseline develop-time patch strategy. Furthermore we have shown that HOTFIXER preserves program behaviour in 98% of the benchmarks, and, that not only does the overhead of HOTFIXER not exceed 17% loss at steady state compared to software patching, often HOTFIXER outperforms a software patched version at steady state, for the Wickert benchmark. Through this evaluation we have shown that HOTFIXER presents a simple and effective technique to enhance application security.

References

[1] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 73–84. [Online]. Available: https://doi.org/10.1145/2508859.2516693

[2] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler, and et al., "Cognicrypt: Supporting developers in using cryptography," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. IEEE Press, 2017, p. 931–936.

[3] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why eve and mallory love android: An analysis of android ssl (in)security," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 50–61. [Online]. Available: https://doi.org/10.1145/2382196.2382205

[4] D. C. Nguyen, D. Wermke, Y. Acar, M. Backes, C. Weir, and S. Fahl, "A stitch in time: Supporting android developers in writingsecure code," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1065–1077. [Online]. Available: https://doi.org/10.1145/3133956.3133977

[5] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, and D. D. Yao, "Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 2455–2472. [Online]. Available: https://doi.org/10.1145/3319535.3345659

[6] L. Piccolboni, G. D. Guglielmo, L. P. Carloni, and S. Sethumadhavan, "Crylogger: Detecting crypto misuses dynamically," 2020.

[7] S. Krüger, K. Ali, and E. Bodden, "Cognicrypt$_{gen}$: generating code for the secure usage of crypto apis," in *CGO '20: 18th ACM/IEEE International Symposium on Code Generation and Optimization, San Diego, CA, USA, February, 2020.* ACM, 2020, pp. 185–198. [Online]. Available: https://doi.org/10.1145/3368826.3377905

[8] S. Ma, D. Lo, T. Li, and R. H. Deng, "Cdrep: Automatic repair of cryptographic misuses in android applications," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 711–722. [Online]. Available: https://doi.org/10.1145/2897845.2897896

[9] L. Singleton, R. Zhao, M. Song, and H. Siy, "Firebugs: Finding and repairing bugs with security patterns," in *Proceedings of the 6th International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '19. IEEE Press, 2019, p. 30–34.

[10] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[11] NIST, "Des modes of operation." [Online]. Available: https://csrc.nist.gov/csrc/media/publications/fips/81/archive/1980-12-02/documents/fips81.pdf

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP'97 — Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 220–242.

[13] Oracle, "Jvmti doc." [Online]. Available: https://www.oracle.com/technical-resources/articles/javase/jvm-tool-interface.html

[14] Eclipse, "Eclipse openj9," 2018. [Online]. Available: https://www.eclipse.org/openj9/

[15] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing java bytecode using the soot framework: Is it feasible?" in *Compiler Construction, 9th International Conference, CC 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*, ser. Lecture Notes in Computer Science, D. A. Watt, Ed., vol. 1781. Springer, 2000, pp. 18–34. [Online]. Available: https://doi.org/10.1007/3-540-46423-9_2

[16] Oracle, "Instrumentation documentation." [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/Instrumentation.html#redefineClasses-java.lang.instrument.ClassDefinition...-

[17] V. Sundaresan, L. J. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, "Practical virtual method call resolution for java," in *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000), Minneapolis, Minnesota, USA, October 15-19, 2000*, M. B. Rosson and D. Lea, Eds. ACM, 2000, pp. 264–280. [Online]. Available: https://doi.org/10.1145/353171.353189

[18] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *ECOOP'95 - Object-Oriented Programming, 9th European Conference, Århus, Denmark,*

August 7-11, 1995, Proceedings, ser. Lecture Notes in Computer Science, W. G. Olthoff, Ed., vol. 952. Springer, 1995, pp. 77–101. [Online]. Available: https://doi.org/10.1007/3-540-49538-X_5

[19] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, p. 120–126, Feb. 1978. [Online]. Available: https://doi.org/10.1145/359340.359342

[20] S. Afrose, S. Rahaman, and D. Yao, "Cryptoapi-bench: A comprehensive benchmark on java cryptographic api misuses," in *2019 IEEE Cybersecurity Development (SecDev)*. IEEE, 2019, pp. 49–61.

[21] A.-K. Wickert, M. Reif, M. Eichberg, A. Dodhy, and M. Mezini, "A dataset of parametric cryptographic misuses," in *Proceedings of the 16th International Conference on Mining Software Repositories*, ser. MSR '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 96–100. [Online]. Available: https://doi.org/10.1109/MSR.2019.00023

[22] Docker, "Docker homepage." [Online]. Available: https://www.docker.com/?utm_source=google&utm_medium=cpc&utm_campaign=dockerhomepage&utm_content=namer&utm_term=dockerhomepage&utm_budget=growth&gclid=EAIaIQobChMI_snm98785wIVFo_ICh3F0wRuEAAYASAAEgI8SPD_BwE

[23] Eclipse, "Eclipse openj9," 2018. [Online]. Available: https://www.eclipse.org/omr/

[24] ——, "Eclipse omr clone," 2018. [Online]. Available: https://github.com/eclipse/openj9-omr

[25] IBM, "Ibmruntimes github." [Online]. Available: https://github.com/ibmruntimes/openj9-openjdk-jdk8

[26] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "Sapfix: Automated end-to-end repair at scale," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '19. IEEE Press, 2019, p. 269–278. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2019.00039

[27] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 257–269. [Online]. Available: https://doi-org.login.ezproxy.library.ualberta.ca/10.1145/2771783.2771796

[28] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 61–72. [Online]. Available: https://doi.org/10.1145/1831708.1831716

[29] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for Java," in *OOPSLA 2007 Companion, Montreal, Canada.* ACM, Oct. 2007.

[30] N. Smeets and A. J. H. Simons, "Automated unit testing with randoop, jwalk and µjava versus manual junit testing," 2010. [Online]. Available: http://staffwww.dcs.shef.ac.uk/people/A.Simons/research/reports/jwalksmeets.pdf

[31] JUnit, "Beforeclass javadoc." [Online]. Available: https://junit.org/junit4/javadoc/latest/org/junit/BeforeClass.html

[32] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, S. Cheung, A. Orso, and M. D. Storey, Eds. ACM, 2014, pp. 643–653. [Online]. Available: https://doi.org/10.1145/2635868.2635920

[33] M. Fowler, "Eradicating non-determinism in tests," 2011. [Online]. Available: https://martinfowler.com/articles/nonDeterminism.html

[34] S. S. SHAPIRO and M. B. WILK, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3-4, pp. 591–611, dec 1965. [Online]. Available: https://doi.org/10.1093/biomet/52.3-4.591

[35] J. Arbuthnot, "An argument for divine providence, taken from the constant regularity observ'd in the births of both sexes. by dr. john arbuthnott, physitian in ordinary to her majesty, and fellow of the college of physitians and the royal society," vol. 27, p. 186–190, 1710. [Online]. Available: https://martinfowler.com/articles/nonDeterminism.html

[36] W. Conover, *Practical nonparametric statistics*, 3rd ed., ser. Wiley series in probability and statistics. New York, NY [u.a.]: Wiley, 1999. [Online]. Available: http://gso.gbv.de/DB=2.1/CMD?ACT=SRCHA&SRT=YOP&IKT=1016&TRM=ppn+24551600X&sourceid=fbw_bibsonomy

[37] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "Crysl: An extensible approach to validating the correct usage of cryptographic apis." in *ECOOP*, ser. LIPIcs, T. D. Millstein, Ed., vol. 109. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, pp. 10:1–10:27. [Online]. Available: http://dblp.uni-trier.de/db/conf/ecoop/ecoop2018.html#KrugerS0BM18

[38] Google, "Google tink." [Online]. Available: https://github.com/google/tink

[39] Legion of the Bouncy Castle Inc, "Bouncycastle." [Online]. Available: https://www.bouncycastle.org/

[40] Z. Lin, X. Jiang, D. Xu, B. Mao, and L. Xie, "Autopag: Towards automated software patch generation with source code root cause identification and repair," in *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 329–340. [Online]. Available: https://doi.org/10.1145/1229285.1267001

[41] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "Sapfix: Automated end-to-end repair at scale," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '19. IEEE Press, 2019, p. 269–278. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2019.00039

[42] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, p. 772–781.

[43] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'13. IEEE Press, 2013, p. 356–366. [Online]. Available: https://doi.org/10.1109/ASE.2013.6693094

[44] M. Payer and T. R. Gross, "Hot-patching a web server: A case study of ASAP code repair," in *Eleventh Annual International Conference on Privacy, Security and Trust, PST 2013, 10-12 July, 2013, Tarragona, Catalonia, Spain, July 10-12, 2013*, J. Castellà-Roca, J. Domingo-Ferrer, J. García-Alfaro, A. A. Ghorbani, C. D. Jensen, J. A. Manjón, I. Onut, N. Stakhanova, V. Torra, and J. Zhang, Eds. IEEE Computer Society, 2013, pp. 143–150. [Online]. Available: https://doi.org/10.1109/PST.2013.6596048

[45] M. Zhang and H. Yin, "Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications," in *NDSS*, 2014.

[46] Y. Chen, Y. Li, L. Lu, Y.-H. Lin, H. Vijayakumar, Z. Wang, and X. Ou, "Instaguard: Instantly deployable hot-patches for vulnerable system programs on android," in *NDSS*, 2018.