Hotfixing Misuses of Crypto APIs in Java Programs

Kristen Newbury*
GitHub, Canada
knewbury01@github.com

Karim Ali
University of Alberta, Canada
karim.ali@ualberta.ca

Andrew Craik†
Oracle Labs, Australia
andrew.craik@oracle.com

ABSTRACT

Cryptography (crypto) Application Programming Interfaces (APIs) are essential for application security, but are difficult to use. Therefore, developers often do not securely integrate those APIs into their code, leading to costly security vulnerabilities. While prior work helps developers detect and fix crypto API misuses during development through software patching, those solutions cannot be applied to a vulnerable system that is already running in production. To bridge this gap, we present Hotfixer, a tool that performs automatic crypto API misuse detection and hotfixing for enterprise Java programs at run time. Hotfixer automatically transforms software patches into hotfixes that are valid to apply to a running Java program without shutting down then resuming the execution environment. Hotfixer is built on top of the high performance Eclipse OpenJ9 Java Virtual Machine (JVM), as we aim to provide its applicability and availability in an enterprise environment. Compared to traditional software patching, Hotfixer hotfixes all misuses while preserving program semantics in 98% of those cases and inducing an overall median performance overhead of only 0.2%.

ACM Reference Format:
Kristen Newbury, Karim Ali, and Andrew Craik. 2021. Hotfixing Misuses of Crypto APIs in Java Programs. In *Proceedings of (CASCON'21).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Ponemon Institute has recently estimated the average cost of a data breach to be US$3.86 million with an average time of 280 days to detect and contain [18]. To avoid such high costs, developers often use crypto APIs to deploy strong hashing techniques (e.g., to protect stored credentials) and strong encryption key generation (e.g., to protect against man-in-the-middle attacks). Despite years of research and development, various studies have shown that developers still find it hard to securely integrate crypto APIs into their code [4, 5, 9, 28, 47], invalidating the purpose of using them.

To help developers use crypto APIs securely, prior work has provided several misuse detection tools [9, 11, 26, 36, 43, 45] and tools that generate secure code to perform crypto tasks [25]. However, those tools ignore the post-distribution steps in the lifecycle of a software system and therefore only approach crypto API misuse detection and fixing from a patching perspective. In that perspective, a patch is a change to a codebase that occurs during development (i.e., does not happen at run time). When a change is applied to an application during its execution (i.e., during the maintenance phase of the software lifecycle), we call it a hotfix.

As an example of deployed software (i.e., software that is in the maintenance phase of the software lifecycle), consider a web server running a Java application with a high demand for throughput and availability. Additionally, the application may frequently serve lengthy connections to clients. This web application may also utilize non-persistent memory to maintain crucial state due to its performance advantages over persistent databases. High performance JVMs such as Eclipse OpenJ9 (a JVM that is fundamentally connected to the history and development of the IBM SDK) [12] realistically must be able to handle these nontrivial demands of high availability, high throughput and reliable security. Using existing tools to fix misuses of crypto APIs on such a server faces several challenges.

Challenge#1. Software patching requires developing a patch offline, then integrating it into the application code. We can then rebuild the new version of the application, shut down the running server, and deploy the new version of the application to the server. It is only after this point that we have a new JVM instance that runs the patched application. This scenario requires time and resources, which may leave the application vulnerable to attacks for an unacceptable window of time.

Challenge#2. To keep its services up and running, an organization may gradually roll out the patch to each server in production. However, a manually applied rollout has to be planned, managed, and executed correctly, which increases the complexity of software management and also the cost for a potential data breach [18] in the event that any step in this process goes wrong.

Challenge#3. This complex roll out process also impacts the availability and performance of the service, unless further nontrivial scheduling and computing resources are utilized. Nontrivial consideration must be given to scheduling the patch updates such that long standing client connections are either prematurely terminated (which will negatively impact application availability) or allowed to terminate at some undetermined time (which negatively impacts the timeline for when a critical security update can be applied). Additionally, (1) restarting a JVM incurs a start-up cost and (2) reducing the number of active machines during the rollout will affect the throughput performance of the service. Without additional compute resources to handle balancing this impact, the overall service performance can face unacceptable degradation.

Challenge#4. Lastly, the non-persistent state of the application will be lost upon application shutdown. While Facebook has provided a

*The work was done when the author was at the University of Alberta.
†The work was done when the author was at IBM Canada.

CASCON'21, November 22 - 26 2021, Toronto, Canada
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

solution to perform fast distributed in-memory caching [38], not all (legacy) applications have this type of solution available to them.

Hotfixing poses a solution to these challenges by allowing an application to continue running while also fixing the detected vulnerabilities. Unfortunately, existing tools that provide automatic hotfixing of crypto API misuses (e.g., Firebugs [51] and CDRep [31]) target only Android applications. Therefore, it was not suitable for our enterprise-driven use cases. Additionally, those tools detect only a fixed set of hardcoded misuse patterns, leaving running systems vulnerable to attacks that those patterns do not capture.

To address the limitations of existing solutions, we propose Hotfixer, a tool that performs automatic crypto API misuse detection and hotfixing for Java programs. To detect crypto API misuses, Hotfixer uses CogniCrypt [26], which defines the secure usage of crypto APIs in a specification language called CrySL [27]. Unlike existing tools that use hardcoded misuse patterns, CrySL allows Hotfixer to flag any deviation from the defined secure usage as a misuse. Once Hotfixer detects a misuse, it automatically converts the hand-written developer-generated patch into a hotfix. At the time of this work there is no tool that automatically generates patches for crypto API misuses (as an alternative to developer provided patches). However, we are aware that there is ongoing research focusing on that goal and when that work is complete [24], we will be able to integrate it into Hotfixer to achieve additional toolchain automation. To apply the hotfix at run time, Hotfixer uses the class redefinition functionality provided by the Java Instrumentation API [39].

While using the Java Instrumentation API provides a simple mechanism to apply a hotfix, certain code changes are rejected by the JVM as redefinition non-compliant (i.e., they cause a `java.lang.UnsupportedOperationException` exception to be thrown). In our study, we found that the two main categories of code changes present in patches that render them non-compliant are field addition and method addition. To transform non-compliant patches into redefinition-compliant hotfixes, Hotfixer constructs a set of helper classes for the classes that require hotfixing. This approach is similar to that by Kim and Tilevich [23].

We have implemented Hotfixer on top of Eclipse OpenJ9, to demonstrate the feasibility and value of hotfixing crypto API misuses in an enterprise run-time environment. We have empirically evaluated Hotfixer on two datasets: 103 microbenchmarks [45] and 27 benchmarks from 7 real-world Java applications [59]. Across both datasets, Hotfixer handles all misuses in an identical manner to a baseline develop-time patch strategy, preserving the program semantics for 98% of those cases. Compared to software patching, Hotfixer has a median performance overhead of only 0.2% at the program steady state.

In summary, we present the following contributions:

- We present Hotfixer, a tool that performs automatic run-time crypto API misuse detection and hotfixing (available at `https://github.com/themaplelab/hotfixer`).
- We empirically evaluate Hotfixer with respect to its ability to apply hotfixes while preserving program semantics and its performance overhead compared to traditional software patching.

```
1  public class SecurityUtil{
2  private static final byte[] SALT = {(byte) 0xde, ...};
3  public byte[] encrypt(String property, String externKey) {
4    String alg = "PBEWithMD5AndDES";
5    SecretKeyFactory kf = SecretKeyFactory.getInstance(alg);
6    SecretKey key = kf.generateSecret(new PBEKeySpec(externKey));
7    Cipher cipher = Cipher.getInstance(alg);
8    PBEParameterSpec spec = new PBEParameterSpec(SALT, 20);
9    cipher.init(Cipher.ENCRYPT_MODE, key, spec);
10   return cipher.doFinal(property.getBytes("UTF-8"));
11   }
12 }
```

(a) The original code containing the misuse (Line 8).

```
13 RequiredPredicateError violating CrySL rule for javax.crypto.spec
        .PBEParameterSpec
14   First parameter was not properly generated as randomized
15   at statement: specialinvoke $r13.<javax.crypto.spec.
          PBEParameterSpec: void <init>(byte[],int)>($r7, $r8)
```

(b) The CogniCrypt output for the crypto API misuse above.

Figure 1: An example of a crypto API misuse where a static hardcoded salt initializes a password-based encryption key.

2 AN EXAMPLE OF CRYPTO API MISUSE

Even when a crypto API is implemented correctly, the API exposes choices to the developer that they might misuse. Figure 1a shows an example of a crypto API misuse discovered by Wickert et al. [59] in an open-source Java project. In the example, a password-based encryption key is setup (Line 8) with a static hardcoded salt (Line 2). Although there is nothing functionally wrong with the API, using a static hardcoded salt during encryption leaves the application vulnerable to dictionary attacks [17]. A dictionary is a precomputed table that contains the output of various crypto algorithms used on common data. In this example, the attacker may use the static hardcoded salt and some common passwords to derive the encryption key. The attacker can then check against some encrypted data to see if they can obtain the plaintext for a given input message [19].

Figure 1b shows the CogniCrypt output for the misuse in Figure 1a. CogniCrypt detects the misuse as a *Required Predicate Error*, because CrySL specifies a predicate that the salt provided to the constructor of `javax.crypto.spec.PBEParameterSpec` must be randomized. Given that the example uses a static hardcoded salt, CogniCrypt flags this use as a crypto API misuse. In a typical crypto API misuse detection scenario during development, a developer would now simply fix the misuse and rebuild the application. In our work, Hotfixer uses this information to start its hotfixing process.

3 OVERVIEW OF HOTFIXER

Figure 2 shows the main workflow of Hotfixer, which consists of three main components:

- CogniCrypt_HF: our extension of CogniCrypt [26],
- OpenJ9_HF: our extension of the Eclipse OpenJ9 JVM [8],
- Soot_HF: our extension of the Soot analysis framework [56].

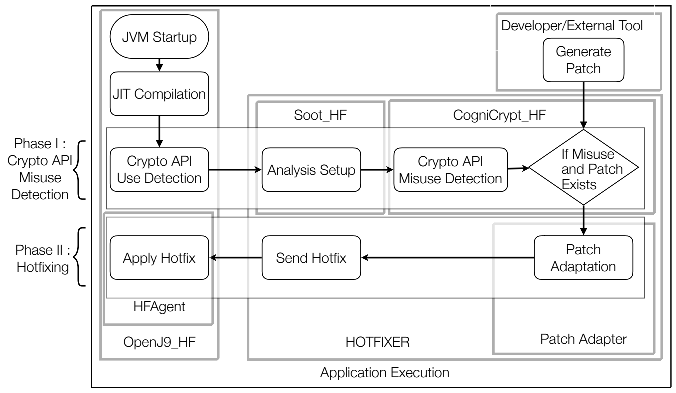

Figure 2: The main workflow of Hotfixer.

```
16 public class SecurityUtil{
17   private PBEParameterSpec pbespec; //added field
18   public byte[] encrypt(String property, String externKey) {
19     initParamSpec();
20     String alg = "PBEWithMD5AndDES";
21     SecretKeyFactory kf = SecretKeyFactory.getInstance(alg);
22     SecretKey key = kf.generateSecret(new PBEKeySpec(externKey));
23     Cipher cipher = Cipher.getInstance(alg);
24     cipher.init(Cipher.ENCRYPT_MODE, key, pbespec);
25     return cipher.doFinal(property.getBytes("UTF-8"));
26   }
27   //added method
28   private void initParamSpec(){
29     SecureRandom rand = new SecureRandom();
30     byte bytes[] = new byte[8];
31     rand.nextBytes(bytes);
32     pbespec = new PBEParameterSpec(bytes, 20);
33   }
34 }
```

Figure 3: The patch for the crypto API misuse from Figure 1.

3.1 Phase I: Crypto API Misuse Detection

Phase I consists of two main steps: crypto API use detection and analysis (i.e., misuse detection). The first step to utilizing Hotfixer is to launch CogniCrypt_HF separately. On startup CogniCrypt_HF sets up a server that waits for analysis requests. If OpenJ9_HF runs in Hotfixer mode, it connects to CogniCrypt_HF during JVM startup. CogniCrypt_HF then provides OpenJ9_HF with a set of analysis seeds, which are the names of crypto classes that CogniCrypt_HF has rules for. The Just-In-Time (JIT) compiler of OpenJ9_HF uses those seeds to check if a method currently being compiled contains any calls to crypto APIs. Eclipse OpenJ9 sorts methods into various compilation levels for optimization: `cold`, `warm`, `hot`, `veryhot`, and `scorching`. To maximize our analysis coverage, we setup the JIT compiler to search for crypto API uses in all methods at all compilation levels. Using the JIT compiler, as opposed to the JVM interpreter, allows Hotfixer to prioritize methods that are more likely to contribute to application security at runtime.

For each method under compilation, the JIT compiler checks if the full signature of each callee in the method matches any of the seeds. Figure 1 shows an example where, during the compilation of `SecurityUtil.encrypt()`, the JIT compiler encounters a callsite (Line 5) whose callee method signature matches the seed `javax.crypto.SecretKeyFactory`. When the JIT compiler finds a crypto API use, it sends the name of the class that contains the method under compilation (i.e., `SecurityUtil`) to CogniCrypt_HF. To ensure that Hotfixer analyzes the same classes that are loaded in the running application, Soot_HF first tries to obtain the classes from the Eclipse OpenJ9 Shared Class Cache (SCC). If Soot_HF does not find a class in the SCC, it falls back to available class files (i.e., regular bytecode format). If multiple versions of the same application are deployed to the server, this process enables Hotfixer to analyze the correct version of the application while still serving the analysis to multiple application clients. Once Soot_HF has gathered the necessary classes to analyze, CogniCrypt_HF produces a report of the detected misuses. CogniCrypt_HF then determines whether it has been provided with an applicable patch. If CogniCrypt_HF has such a patch, Hotfixer then enters Phase II.

3.2 Phase II: Hotfixing

To create a hotfix, CogniCrypt_HF invokes the patch adapter, which we built on top of Soot_HF. The patch adapter consumes the hand-written developer generated patch and outputs a hotfix. To hotfix a running application, we implemented HFAgent, a custom Java agent that uses the Java Instrumentation API to redefine classes at runtime. The redefinition mechanism exposed in the Java Instrumentation API is a JVM-agnostic interface that performs class redefinition. Due to the nature of the patches relevant to this work, our design must consider two well known limitations of the Java Instrumentation API.

First, *"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."* [39]. To fix the misuse in Figure 1 (Line 8) without causing an exception, the developer must introduce a new method in `SecurityUtil`, which is shown in Figure 3 (Line 28). However, due to the limitations on redefinition, this patch is non-compliant and Hotfixer must adapt it before applying it as a hotfix.

Second, the Java standards [39] state that *"redefining a class does not cause its initializers to be run"*. Initializers are special methods that set initial class state and are run by the JVM the first time a class is loaded. For example, classes have an initial state when they declare static fields. Because the redefinition mechanism does not involve re-running class initializers, changes to static initial field value will not take effect for any classes being redefined. Figure 4 shows an example misuse (Line 38) that may be fixed by using a more secure encryption algorithm (Line 43). Since static initializers are not rerun during redefinition events, encryption using the redefined method `encrypt()` (Line 44) will not use the newly specified algorithm. To observe the intended effects of this patch, our patch adapter must transform the patch; this case requires a well known Dynamic Software Updating (DSU) technique called object transformation [10]. Hotfixer handles object transformation for static fields eagerly by duplicating field value assignment statements in helper classes so that the new values are observed once the redefinition event has occurred.

Given these limitations, we have identified 3 categories of patches:

```
35 public class SecurityUtil{
36   public static String alg = "DES/CBC/PKCS5Padding";
37   public byte[] encrypt(String data){
38       Cipher cipher = Cipher.getInstance(alg);
39       ...
40   }
41 }
```

(a) The original code containing a misuse (Line 38).

```
42 public class SecurityUtil{
43   public static String alg = "AES/CBC/PKCS5Padding";
44   public byte[] encrypt(String data){
45       Cipher cipher = Cipher.getInstance(alg);
46       ...
47   }
48 }
```

(b) The patch that fixes the misuse.

Figure 4: An example illustrating a patch that changes a static field value.

(1) **Adjusted Hotfix:** If any class in a patch is redefinition non-compliant, then Hotfixer' patch adapter, given a patch, must generate a semantically equivalent, redefinition compliant hotfix.
(2) **JVM-Assisted Hotfix:** If a class in the patch cannot be adjusted by our patch adapter, Hotfixer may require assistance from the JVM to apply this patch as a hotfix. An example of this category would be a patch requiring object instance field values of already allocated objects to be updated.
(3) **Multi-JVM Hotfix:** If the semantics of the changes in the patch require modifying multiple running applications, we classify this as a multi-JVM patch. Changing the encryption protocol being used by an open connection would be an example of this category of change.

For brevity we omit discussion of the default case where a patch can be used as a hotfix without adjustment (i.e., naturally does not cause errors). In this paper, we focus on addressing the Adjusted Hotfix category.

4 EVALUATION

To evaluate Hotfixer, we compare it to applying a develop-time patch. Our evaluation aims at answering the following questions:

RQ1: Does Hotfixer correctly fix crypto API misuses?
RQ2: Does Hotfixer alter program semantics?
RQ3: How does Hotfixer affect application performance?

4.1 Experimental Setup

We have conducted our experiments using two datasets: a dataset constructed by Sharmin et al. [1] comprising of 103 benchmarks (CryptoGuardBench) and a dataset constructed by Wickert et al. [59] comprising of 27 benchmarks across 7 open-source Java projects (WickertBench). Each benchmark contains a misuse and its respective patch. We ran all of our experiments in a Docker container [7] version 19.03.5 on an x86_64 Ubuntu 18.04.4 machine with two 2.4GHz AMD EPYC 7351 16-Core processors running OpenJ9_HF.

4.2 Data Processing

4.2.1 CryptoGuardBench. Sharmin et al. [1] provide 14 correct use examples of crypto APIs (for the purpose of detecting true negatives in crypto API misuse detection tools), which we utilize as patches for Hotfixer. We specialized these general patch classes to each individual benchmark in the dataset.

4.2.2 WickertBench. To further evaluate Hotfixer, we obtain the exact versions of all projects in the original dataset by Wickert et al. [59]; we refer to this as the original version. For this dataset, the authors had manually crafted fixes for each misuse, and contributed each fix in a code snippet separate from the original benchmark. For each benchmark, we created a devpatch version by manually integrating each provided fix into the application at the misuse location indicated by the dataset authors. The majority of the contributed corrections (34/44) could not be used as corrections in isolation. For example, to correctly accomplish an encryption task, a common setup component must be used in both encryption and decryption. To resolve these issues in the original fixes, we merged the related corrections into a single fix. To integrate these merged corrections into the code, we followed two principles. For intra-procedural single-statement changes, we applied the correction as that single-statement change. For all other changes, we applied them in newly added methods. We also added invocations for those new methods in pre-existing instance or static methods. We maintained the original schema for identifying the fixes, and when we merged fixes, we label the result as misuseXandY, where X and Y are the original identifiers of the fixes. This process yielded 27 benchmarks.

4.2.3 Crypto API Misuse Detection. In preparation for answering our research questions, we confirmed that, despite utilizing classes from the SCC, Hotfixer is as effective at detecting crypto API misuses as CogniCrypt. For CryptoGuardBench, Hotfixer finds the same misuses as CogniCrypt, except for 6 benchmarks where Hotfixer detects one extra misuse. For the 27 benchmarks in WickertBench, the findings of Hotfixer and CogniCrypt are identical for the original version of each project.

4.3 Fixing Crypto API Misuses (RQ1)

4.3.1 CryptoGuardBench. To confirm that Hotfixer addresses the present misuses, we added a post-hotfix run of CogniCrypt to our setup. Using this technique, we verify that Hotfixer fixes the same misuses that a develop-time patch strategy can fix for the entire suite. During this post-hotfix CogniCrypt analysis, Hotfixer finds 6 misuses more than CogniCrypt. However, these extra misuses are also detected by Hotfixer in the application before hotfixing (as mentioned above in Section 4.2.3). All 6 cases are reported at an extra upcast statement in the class during hotfixing compared to the class when used in the develop-time patch. Further investigation suggests that there is either a seemingly superfluous difference in the class format obtained from the SCC compared to a regular classfile, or that Hotfixer generates a redundant upcast statement while reading SCC format compared to a regular classfile.

Table 1: The results of running HOTFIXER on WICKERTBENCH. Each test either passes (P) or fails (F).

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

Since the misuse is present before and after hotfixing, we do not consider it a misuse caused by HOTFIXER.

4.3.2 WICKERTBENCH. Across all 27 benchmarks, HOTFIXER fixes the same misuses as the baseline develop-time patch strategy, without introducing any additional misuses.

HOTFIXER fixes all crypto API misuses in CRYPTOGUARDBENCH and WICKERTBENCH, without introducing any additional misuses.

4.4 Altering Program Semantics (RQ2)

Similar to prior work on software patching [3, 34, 51, 57], we use regression testing to assess whether applying HOTFIXER to a program alters its intended original functionality.

4.4.1 Regression Test Setup. For each patched benchmark in CRYPTOGUARDBENCH and the DEVPATCH version of each benchmark in WICKERTBENCH, we generated a set of regression tests using Randoop [41], an automatic test generation framework. We configured Randoop with a 60-second time limit for the test generation. This limit is more than double the typical suggested time limit [52]. For CRYPTOGUARDBENCH, Randoop generated a total of 78,488 tests (min: 2, max: 5,000, median: 6), whereas for WICKERTBENCH, Randoop generated a total of 40,493 tests (min: 2, max: 8,658, median: 240).

To answer RQ2, we created two setups from the Randoop tests. For CRYPTOGUARDBENCH, we assess the output of an iteration of a regression test suite once the redefinition event has taken place. To detect the occurrence of the redefinition event we manually created a setup method (using the Junit `BeforeClass` annotation [21]) in the setup class of each regression test suite. Our setup consists of: an invocation of the method(s) affected by the misuse (and therefore the patch) such that we perform a logical task (e.g., an encryption followed by a decryption), a loop that ensures that the JIT compiler compiles those methods, a pause until HFAGENT has completed the redefinition event, and a repetition of the same task that was initially performed in the method. After this setup, the regression tests begin to run. For WICKERTBENCH we create a setup that more closely simulates how a longstanding application runs. We rerun the test suite for a large number of iterations (i.e., a test window). We then observe whether test failures occur after redefinition (or in the case of 2 tests in NETTYGAMESERVER whether tests expected to fail, in the baseline condition, continue to fail). Each test window is thus comprised of: 1) an original subwindow, the iterations where the original application executes, 2) a redefinition event, and 3) a hotfixed subwindow, the iterations where the hotfixed application executes. Since a sufficient test window varies between benchmarks, we identified its value experimentally.

Furthermore, Randoop generated a total of 161 flaky tests [46] for HA-BRIDGE MISUSE3AND8, JEESUITE-LIBS MISUSE1AND4, JEESUITE-LIBS MISUSE6AND7, and WHATSMARS MISUSE13. We observed that those tests fail at random in the original subwindow and the hotfixed subwindow. To ensure that we do not confound our ability to determine if HOTFIXER introduces errors into the application with the results of those flaky tests, we removed them from our testsuite.

4.4.2 CRYPTOGUARDBENCH. We observed only 2 test failures across 78,488 tests: STATICINITIALIZATIONVECTORABICASE2 and STATICSALTSABICASE2. Both failures are due to tests that check against values of `public static final` fields of the redefined class. Since static initializers are not rerun during redefinition events, to observe the value changes, HOTFIXER must redefine the modified static variable. However, when `javac` compiles the tests alongside the original benchmark class (which simulates how a real application would be compiled before deployment), the original constant value of the field propagates to all its uses. Because the tests are for the DEVPATCH version of the benchmark, the value that the test checks against is the field value in the patched class. The `javac` constant propagation optimization guarantees that the test will fail. Our patch adapter does not reverse that optimization, because this case falls into the currently unsupported JVM-Assisted Hotfix category.

4.4.3 WICKERTBENCH. Table 1 shows the results of running the 40,493 tests, where the pass category refers to whether the outcome preserves the expected semantics (i.e., if a test passes in the DEVPATCH version condition and during hotfixing, or the test fails in the DEVPATCH version condition and during hotfixing). All tests pass except for in benchmark WHATSMARS where we observe 5 test failures in MISUSE1AND3 that occur because the patch introduces a

Table 2: The results of running pre-existing tests in WickertBench. Each test passes (P), fails (F), or errs (E).

Benchmark	# Tests	original			Misuse	devpatch			Hotfixed		
		P	F	E		P	F	E	P	F	E
ha-bridge	9	9	0	0	1and5	9	0	0	9	0	0
					2and7	9	0	0	9	0	0
					3and8	9	0	0	9	0	0
					4and6	9	0	0	9	0	0
instagram4j	0	N/A			1	N/A					
jeesuite-libs	20	1	0	19	1and4	1	0	19	1	0	19
					2and5	1	0	19	1	0	19
					3	1	0	19	1	0	19
					6and7	1	0	19	1	0	19
					8	1	0	19	1	0	19
					9	1	0	19	1	0	19
NettyGameServer	37	0	0	37	1	0	0	37	0	0	37
					2and3	0	0	37	0	0	37
					4	0	0	37	0	0	37
smart	0	N/A			1-8	N/A					
whatsmars	15	12	0	3	1and3	12	0	3	11	0	4
					2and4	12	0	3	11	0	4
					5and9	12	0	3	11	0	4
					6and11	12	0	3	11	0	4
					7and12	12	0	3	11	0	4
					8and10	12	0	3	11	0	4
					13	12	0	3	11	0	4
dragonite-java	5	5	0	0	1	5	0	0	5	0	0

new static field that must only be initialized at specific points in the application. For those tests, the program accesses the state before initialization. This case falls into the currently unsupported JVM-Assisted Hotfix category, because it requires the JVM to determine application points where performing the hotfix avoids all conflicts with the logical state transitions during all tasks.

While this case legitimately impacts the correctness of hotfixing, if there were persistent (e.g., database) effects based on the introduced state (e.g., hashed passwords stored in a database that were either statically salted or not salted at all), even software patching techniques would need to give further nontrivial consideration to the implications of these patch change types.

4.4.4 WickertBench Pre-existing Tests. For each project in WickertBench, Table 2 shows the results of running the pre-existing test set for original version, devpatch version, and the hotfixed application. To run the pre-existing tests, we constructed a test setup that ran the pre-existing tests after the redefinition event has occurred (i.e., in the same JVM instance that the redefinition event occurred in). Additionally, jeesuite-libs and whatsmars contain tests across multiple Maven/Gradle modules in the project. For each project, to run those tests in the same JVM instance, we constructed a custom test runner that invokes the pre-existing tests after the redefinition event is guaranteed to occur.

Across all benchmarks, for the 161 errors that occur in the Hotfixer condition, we manually verified that the cause is the same as for the corresponding errors observed in the patched application condition. In whatsmars we could not replicate the exact test setup that Maven would construct, which caused 2 errors originating from an XML source configuration issue. We manually verified that these 2 errors are not seen in the devpatch version condition. In other words, for those 2 errors, we are unable to guarantee that Hotfixer would not cause any unintended program behaviour.

The results of running the pre-existing tests for each project corroborate our findings from our generated regression test suites. Empirically, Hotfixer has a high success rate in terms of not altering program semantics in an unintended manner. For all tests where we were able to identically replicate the test setup (84 out of 86 across all benchmarks), we observe a 100% success rate in terms of not altering program semantics.

Hotfixer preserves program semantics in 98% of the analyzed benchmarks. The other 2% belong to a patch category that Hotfixer currently does not support.

4.5 Effect on Application Performance (RQ3)

4.5.1 Performance Test Setup. To measure the performance overhead of using Hotfixer, we focus on three metrics: runtime of patch adapter, duration until the JIT compiler has sufficiently recovered from the redefinition event, and relative throughput performance of Hotfixer modified code compared to develop-time patch. We answer RQ3 only for WickertBench, because it consists of real-world applications where performance is a relevant aspect of application execution and the realistic OpenJ9_HF setup allows us to assess the production relevant impacts of Hotfixer. To answer RQ3, we use the same setup for RQ2.

To collect the patch adapter running time, we measure the duration between CogniCrypt_HF receiving the analysis request for the class that causes redefinition and the JVM observing that the redefinition has taken place. To collect recovery duration, we first define sufficient JIT compiler recovery by inspecting the activities of the JIT compiler during execution; as a result of the redefinition event, the JIT compiler 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 compiler has received at most 2 requests or less, for a duration of 2 seconds. A required queue size of 0 would be too strict and unrealistic for prolonged amounts of time. To compare throughput performance, we measure 10 contiguous testsuite execution times after this recovery point.

For each benchmark application, we begin our sample from the same recovery point (i.e., 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 two 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 benchmarks, we define our baseline runtimes using 70 and 100 iterations of the testsuite, respectively. We then begin the sample window from iteration 55 and 85, respectively, because these represent a stable point in the application as the most fair window for comparison.

4.5.2 Results. Over the entire WickertBench, 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 a median recovery time of 4.3 seconds (min: 2.1 seconds, max: 128.6 seconds, standard deviation: 28.8

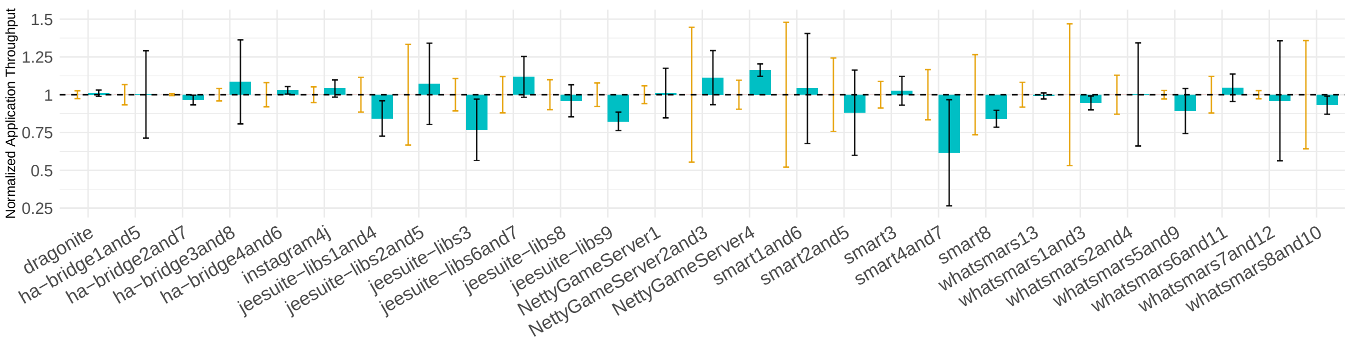

Figure 5: The application throughput of WickertBench using Hotfixer, normalized to that of using the baseline develop-time patch strategy. Yellow error bars represent COV of the baseline, while black error bars are COV of using Hotfixer.

seconds), which represents a median 3.6% (min: 0.5, max: 14.3) of the total runtimes across all of WickertBench.

Two recovery times particularly stand out, for NettyGameServer misuse2and3 and smart misuse1and6 with recovery times of 128.6 and 93 seconds, respectively. These outstanding recovery times are due to the average runtime of each testsuite iteration being over 570× and 270×, respectively, of the runtime of the next longest testsuite runtime in WickertBench. Redefinition will necessarily undo some of the prior work of the JIT compiler but the application will recover once the JIT compiler can perform more compilation. As the runtime of each testsuite iteration in the benchmarks NettyGameServer misuse2and3 and smart misuse1and6 is very long, the JIT compiler's discovery of compilation tasks is delayed until enough invocations of the affected methods occur again to trigger compilation by the JIT compiler. Even if other benchmarks take some number of testsuite iterations to recover after a redefinition event, in the case of NettyGameServer misuse2and3 and smart misuse1and6, that period is much longer.

Figure 5 compares the application throughput when using Hotfixer against develop-time patch. Each bar represents the geometric mean across the iterations in our sample window normalized to the baseline develop-time patch strategy. A ratio of 1 means that they perform the same, values above 1 mean that using Hotfixer induces an overhead compared to patching, and values below 1 mean that using Hotfixer improves the application throughput. Across all benchmarks, the overall median effect is 0.2% overhead compared to the baseline. The maximum overhead induced in any benchmark is 16.3% and the maximum speedup induced is 38.4%.

To determine whether these results are statistically significant, we performed a Paired Sign Test [55] (using a significance level of 0.05). We observe significant overhead in the benchmarks jeesuite misuse6and7, NettyGameServer misuse4, and whatsmars misuse6and11. We observe significant speedup in the benchmarks ha-bridge misuse2and7, jeesuite misuse3, jeesuite misuse9, and smart misuse8. The overall median of these 7 benchmarks is a 3.5% speedup (max overhead: 16.3% and max speedup: 23.2%).

Since performance is not a critical factor in demonstrating Hotfixer's overall viability, we deem it sufficient at this time to speculate that differences in JIT compilation plans are the cause. Additionally, 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 a develop-time patch. These results along with the overall short recovery times suggest that Hotfixer is a feasible alternative to develop-time patch.

Across all benchmarks, the throughput performance overhead that Hotfixer induces is a median of 0.2%.

5 THREATS TO VALIDITY

There are two main threats to the validity of the results that we present in this paper. First, a threat to the construct validity is that we rely on regression testing to approximate typical application behaviour. Such tests may not encompass all possible behaviours of an application. Therefore, we can only show 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. Similar to much of the work on software patching, we also rely on the established practice of evaluating regression tests.

Second, a threat to the internal validity of our evaluation is that we do not ensure that garbage collection (GC) does not occur during the execution of the application, which may cause a lag in application execution time. To address this threat, we monitored for deviations in any test iteration time over all trials of every setup. When these deviations occurred, we increased the JVM heap size such that GC is less likely to occur during test windows.

6 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 prior work on software patching, which relates to the general principles of Phase II of Hotfixer, and dynamic software updating, which relates to patch adaptation that also occurs in Phase II.

6.1 Crypto API Misuse Detection

General approachs to fault localization [2, 20] may not be well suited to detect crypto API misuses. Such generic detection techniques often rely on test suites that are available for a particular application. While automatic test generation frameworks (e.g., Randoop [52] and EvoSuite [13]) may generate unit tests that express the correctness of method output, these tests cannot check whether a certain method call occurred during execution, or whether an object used a constant during its initialization, both of which are common reasons of crypto API misuses [28]. Therefore, to detect crypto API misuses, it would take a considerable effort to augment automatically generated test suites.

Alternatively, researchers have developed specialized static analysis tools to help developers with crypto API misuses detection. CryptoLint [9], MalloDroid [11], and FixDroid [36] are specific to the Android platform, while other tools such as CryptoGuard [45], and CRYLOGGER [43] detect misuses in both Android and Java apps. Hotfixer relies on CogniCrypt [26] 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 [27], which enables extensible definitions of misuses that are easy to update. Using CrySL allows 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 [15] and BouncyCastle [29].

6.2 Software Patching

Software patching tools provide a foundation for 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 correctness. Automation in software patching may occur at three points in the process: fault detection, patch generation, and patch application (i.e., program repair). The state of the art tools in various levels of automated end-to-end software patching include AutoPAG [30], SapFix [35], SemFix [37], AE [58], and ASAP [42].

AutoPAG [30] uses static analysis to detect out-of-bounds errors. SapFix [35] relies on static analysis to validate the ability of candidate patches to fix null pointer exceptions. Similarly, Hotfixer uses static analysis to perform crypto API misuses detection and to evaluate the correctness of the generated hotfixes. SapFix [35] uses software testing to narrow down candidate patches during a patch selection process. SemFix [37] also uses software testing during the patch generation process. However, both tools use tests to create a set of constraints that the patch must satisfy as it is generated. ELIXIR [49] additionally uses machine learning to narrow down its generated patches.

Unlike all generate-and-validate tools, Hotfixer does not generate the patch used in its hotfix and instead uses a hand-written developer generated patch. Additionally, Hotfixer uses CogniCrypt to ensure that crypto API misuses are truly fixed in the generated hotfix and regression testing to validate the correctness of the hotfix after applying it to the analyzed program. Our evaluation has shown that Hotfixer is more successful than existing general program repair tools that have lower success rates such as SemFix [37] (53.3% or 48/90), PAR [22] (22.6% or 27/119), and AE [58] (50.5% or 53/105). Furthermore, this performance of general program repair tools would only comparable to that of Hotfixer if they were modified to have a fault localization ability for crypto API misuses.

6.3 Dynamic Software Updating

The concept of DSU has been formalized at least since 1976 [10]. In a survey conducted in 2012, Seifzadeh et al. [50] shows that there were at least 23 DSU publications prior to that year. DSU is such an established topic in software engineering that here we will discuss only the relevant work that focus on providing or enhancing DSU functionality in a JVM.

Orso et al. [40] create a swapping-enabled application via an introduction of proxy classes. Kim and Tilevich [23] describe an approach that combines binary refactoring [54] and virtual superclasses to also create a swapping enabled application. Additionally, Rubah [44] uses bytecode rewriting as a key component of its process. The aforementioned techniques all rely on a pre-processing step that require the set of classes containing misuses to be known prior to the execution of the application, which is infeasible for Hotfixer. Hotfixer hotfixes applications under the assumption that the exact set of classes containing crypto API misuses is unknown prior to deploying the application. Otherwise, those misuses would already have been patched during development.

Tools such as DCEVM [60], Jvolve [53], dReAM [14], JDRUMS [48] and the work by Malabarba et al. [33] present an alternative approach to bytecode rewriting by customizing a JVM to facilitate class redefinition. Unlike these tools, Hotfixer performs its patch adaptation phase asynchronously with respect to the execution of the application. This means that our application JVM (OpenJ9_HF) is not customized with respect to hotfixing behaviour (i.e., OpenJ9_HF performs hotfixing the same as any stock JVM), which minimizes the potential overhead incurred by the patch adaptation phase.

Lastly, an important component of DSU in object-oriented languages is management of the transition of state (i.e., field addition, removal, or value changes), which is commonly known as object transformation. Tools such as PASTA [62], TOS [32] and AOTES [16] attempt to handle all possible state changes listed. However, in our enterprise-driven scenarios, we have only encountered patches that make changes to static fields. Therefore, Hotfixer was only required to currently support a subset of the object transformation techniques discussed in that related work. We recognize that, for a more theoretically complete tool, Hotfixer would likely be need to be extended in the future to accommodate additional patch change types.

6.4 Security Hotfixes for Android

Recent work has focused on security-related hotfixing in Android apps. AppSealer [61] prevents injection and information leakage attacks in Android apps. To automatically detect vulnerabilities, AppSealer uses a combination of static analysis and run-time instrumentation. If AppSealer detects a potential vulnerability in a running application, it simply alerts the user to restart the app. Similar to AppSealer, Hotfixer utilizes static analysis for fault detection, and we also automate most of the patching process. Unlike AppSealer, Hotfixer never restarts the application; it instead applies a hotfix on the executing code when it detects a vulnerability.

InstaGuard [6] is a fully automated hotfix tool for Android apps. InstaGuard avoids adding new code to the app, and instead disallows vulnerability by terminating the app when it detects an insecure condition. Insecure conditions are defined via modular rulesets called GuardRules. Similar to InstaGuard, Hotfixer uses modular specifications, in our case CrySL, for fault detection. Unlike InstaGuard and AppSealer, Hotfixer enables continued program execution at all times. While InstaGuard targets generic security vulnerabilities and AppSealer targets component hijacking vulnerabilities, Hotfixer performs targeted program repair for crypto API misuses.

The most relevant work in the literature to ours is CDRep [31], a tool that automatically repairs crypto API misuses in Android apps. CDRep automatically detects 7 specific misuses by utilizing and extending misuse patterns defined by CryptoLint [9]. To fix misuses, CDRep automatically applies a handwritten template patch to the bytecode of an Android application. Unlike Hotfixer, CDRep does not handle the issue of hotfixing a running application and instead focuses on automatically creating a patched application that would need to be deployed instead of the existing one. FireBugs [51] 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 both CDRep and FireBugs use fixed sets of misuse patterns, Hotfixer uses definitions of the secure usage of crypto APIs in a specification language called CrySL [27]. This approach enables Hotfixer to flag any deviation from those definitions as a misuse, without resorting to a fixed, hardcoded set of misuse patterns.

7 CONCLUSION

The secure integration of crypto APIs into production code is crucial for the overall security of enterprise applications. While prior work has provided several misuse detection tools for use in development environments, there are currently no tools to aid application maintenance teams with fixing crypto API misuses in running enterprise Java applications.

In this paper we have presented Hotfixer, a tool that automatically hotfixes crypto API misuse in a running Java application. Hotfixer offers a beneficial alternative to software patching in scenarios where it is nontrivial to restart servers to apply patches due to the extensive management and compute resources required to maintain application performance and availability. Our evaluation has shown that Hotfixer hotfixes all misuses 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, while inducing a median performance overhead of only 0.2% at steady state compared to software patching.

The features that Hotfixer covers were driven by the demands of the specialized scenarios that we aimed to handle. Through this evaluation, we have shown that Hotfixer presents a simple and effective technique to enhance application security.

ACKNOWLEDGMENTS

This material is based upon work supported by IBM Canada Advanced Studies (CAS).

REFERENCES

[1] Sharmin Afrose, Sazzadur Rahaman, and Danfeng Yao. 2019. CryptoAPI-Bench: A Comprehensive Benchmark on Java Cryptographic API Misuses. In *2019 IEEE Cybersecurity Development (SecDev)*. IEEE, 49–61.

[2] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. 1995. Fault localization using execution slices and dataflow tests. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95.* 143–151. https://doi.org/10.1109/ISSRE.1995.497652

[3] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated Software Transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015).* ACM, New York, NY, USA, 257–269. https://doi.org/10.1145/2771783.2771796

[4] VeraCode (CA). 2017. State of Software Security 2017. https://info.veracode.com/report-state-of-software-security.html.

[5] Alexia Chatzikonstantinou, Christoforos Ntantogian, Georgios Karopoulos, and Christos Xenakis. 2015. Evaluation of Cryptography Usage in Android Applications. In *EAI International Conference on Bio-inspired Information and Communications Technologies.* 83–90. http://dl.acm.org/citation.cfm?id=2954820

[6] Yaohui Chen, Yuping Li, Long Lu, Yueh-Hsun Lin, Hayawardh Vijayakumar, Zhi Wang, and Xinming Ou. 2018. InstaGuard: Instantly Deployable Hot-patches for Vulnerable System Programs on Android. In *NDSS*.

[7] Docker. [n. d.]. Docker Homepage. https://www.docker.com/

[8] Eclipse. 2018. Eclipse OpenJ9. https://www.eclipse.org/openj9/

[9] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13).* ACM, New York, NY, USA, 73–84. https://doi.org/10.1145/2508859.2516693

[10] R S Fabry. 1976. How to design a system in which modules can be changed on the fly.. In *In 2nd Intnl. Conf. on Software Eng. (ICSE),.* IEEE-CS Press., 470–476.

[11] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. 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 (CCS '12).* ACM, New York, NY, USA, 50–61. https://doi.org/10.1145/2382196.2382205

[12] Eclipse Foundation. [n. d.]. More about OpenJ9. https://www.eclipse.org/openj9/about.

[13] Gordon Fraser and Andrea Arcuri. 2017. EvoSuite at the SBST 2017 Tool Competition. In *10th International Workshop on Search-Based Software Testing (SBST'17) at ICSE'17.* 39–42.

[14] Bashar Gharaibeh, Danny Dig, Tien Nguyen, and J Chang. 2007. dReAM: Dynamic refactoring-aware automated migration of Java online applications. (01 2007).

[15] Google. [n. d.]. Google Tink. https://github.com/google/tink

[16] Tianxiao Gu, Xiaoxing Ma, Chang Xu, Yanyan Jiang, Chun Cao, and Jian Lu. 2018. Automating Object Transformations for Dynamic Software Updating via Online Execution Synthesis. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs))*, Todd Millstein (Ed.), Vol. 109. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 19:1–19:28. https://doi.org/10.4230/LIPIcs.ECOOP.2018.19

[17] Hacksplaining. [n. d.]. Dictionary Attacks. https://www.hacksplaining.com/glossary/dictionary-attacks

[18] Ponemon Institute. [n. d.]. Cost of a Data Breach Report 2020. https://www.capita.com/sites/g/files/nginej146/files/2020-08/Ponemon-Global-Cost-of-Data-Breach-Study-2020.pdf

[19] Javamex. [n. d.]. Password-based encryption in Java: salt and key derivation. https://www.javamex.com/tutorials/cryptography/pbe_salt.shtml

[20] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02).* Association for Computing Machinery, New York, NY, USA, 467–477. https://doi.org/10.1145/581339.581397

[21] JUnit. [n. d.]. BeforeClass javadoc. https://junit.org/junit4/javadoc/latest/org/junit/BeforeClass.html

[22] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-Written Patches. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13).* IEEE Press, 802–811.

[23] Dong Kwan Kim and Eli Tilevich. 2008. Overcoming JVM HotSwap Constraints Via Binary Rewriting. In *ACM Workshop on Hot Topics in Software Upgrades (HotSWUp).* ACM. https://doi.org/10.1145/1490283.1490290

[24] Stefan Krüger. 2020. *CogniCrypt - the secure integration of cryptographic software.* Ph.D. Dissertation. University of Paderborn. https://doi.org/10.17619/UNIPB/1-1039

[25] Stefan Krüger, Karim Ali, and Eric Bodden. 2020. CogniCrypt$_{GEN}$: generating code for the secure usage of crypto APIs. In *CGO '20: 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO).* ACM, 185–198. https://doi.org/10.1145/3368826.3377905

[26] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and et al. 2017. CogniCrypt: Supporting Developers in Using Cryptography. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017).* IEEE Press, 931–936.

[27] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. 2018. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs.. In *ECOOP (LIPIcs)*, Todd D. Millstein (Ed.), Vol. 109. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 10:1–10:27. http://dblp.uni-trier.de/db/conf/ecoop/ecoop2018.html#KrugerS0BM18

[28] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. 2014. Why Does Cryptographic Software Fail? A Case Study and Open Problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems (APSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 7. https://doi.org/10.1145/2637166.2637237

[29] Legion of the Bouncy Castle Inc. [n. d.]. BouncyCastle. https://www.bouncycastle.org/

[30] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, Bing Mao, and Li Xie. 2007. 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 (ASIACCS '07)*. ACM, New York, NY, USA, 329–340. https://doi.org/10.1145/1229285.1267001

[31] Siqi Ma, David Lo, Teng Li, and Robert H. Deng. 2016. CDRep: Automatic Repair of Cryptographic Misuses in Android Applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16)*. ACM, New York, NY, USA, 711–722. https://doi.org/10.1145/2897845.2897896

[32] Stephen Magill, Michael Hicks, Suriya Subramanian, and Kathryn S. McKinley. 2012. Automating Object Transformations for Dynamic Software Updating. *SIGPLAN Not.* 47, 10 (Oct. 2012), 265–280. https://doi.org/10.1145/2398857.2384636

[33] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. 2000. Runtime Support for Type-Safe Dynamic Java Classes. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP '00)*. Springer-Verlag, Berlin, Heidelberg, 337–361.

[34] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott. 2019. SapFix: Automated End-to-End Repair at Scale. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '19)*. IEEE Press, 269–278. https://doi.org/10.1109/ICSE-SEIP.2019.00039

[35] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott. 2019. SapFix: Automated End-to-End Repair at Scale. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '19)*. IEEE Press, 269–278. https://doi.org/10.1109/ICSE-SEIP.2019.00039

[36] Duc Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. 2017. A Stitch in Time: Supporting Android Developers in WritingSecure Code. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 1065–1077. https://doi.org/10.1145/3133956.3133977

[37] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, 772–781.

[38] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 385–398. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[39] Oracle. [n. d.]. Instrumentation documentation. https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/Instrumentation.html#redefineClasses-java.lang.instrument.ClassDefinition...-

[40] Alessandro Orso, Anup Rao, and Mary Jean Harrold. 2002. A Technique for Dynamic Updating of Java Software. In *International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 649–658. https://doi.org/10.1109/ICSM.2002.1167829

[41] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-Directed Random Testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 815–816. https://doi.org/10.1145/1297846.1297902

[42] Mathias Payer and Thomas R. Gross. 2013. Hot-patching a web server: A case study of ASAP code repair. In *Eleventh Annual International Conference on Privacy, Security and Trust, PST 2013 (PST'13)*, Jordi Castellà-Roca, Josep Domingo-Ferrer, Joaquín García-Alfaro, Ali A. Ghorbani, Christian D. Jensen, Jesús A. Manjón, Iosif-Viorel Onut, Natalia Stakhanova, Vicenç Torra, and Jie Zhang (Eds.). IEEE Computer Society, 143–150. https://doi.org/10.1109/PST.2013.6596048

[43] Luca Piccolboni, Giuseppe Di Guglielmo, Luca P. Carloni, and Simha Sethumadhavan. 2020. CRYLOGGER: Detecting Crypto Misuses Dynamically. arXiv:cs.CR/2007.01061

[44] Luís Pina, Luís Veiga, and Michael Hicks. 2014. Rubah: DSU for Java on a Stock JVM. *SIGPLAN Not.* 49, 10 (Oct. 2014), 103–119. https://doi.org/10.1145/2714064.2660220

[45] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. 2019. CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-Sized Java Projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. ACM, New York, NY, USA, 2455–2472. https://doi.org/10.1145/3319535.3345659

[46] Randoop. [n. d.]. Randoop generated flaky tests. https://randoop.github.io/randoop/manual/#flaky-tests

[47] Siegfried Rasthofer, Steven Arzt, Robert Hahn, Max Kohlhagen, and Eric Bodden. 2015. (In)Security of Backend-as-a-Service. In *BlackHat Europe 2015*.

[48] Tobias Ritzau and Jesper Andersson. 2002. Dynamic Deployment of Java Applications. (10 2002).

[49] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. ELIXIR: Effective Object Oriented Program Repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, 648–659.

[50] Habib Seifzadeh, Hassan Abolhassani, and Mohsen Sadighi Moshkenani. 2013. A survey of dynamic software updating. *Journal of Software: Evolution and Process* 25, 5 (2013), 535–568. https://doi.org/10.1002/smr.1556 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1556

[51] Larry Singleton, Rui Zhao, Myoungkyu Song, and Harvey Siy. 2019. FireBugs: Finding and Repairing Bugs with Security Patterns. In *Proceedings of the 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft '19)*. IEEE Press, 30–34.

[52] Nastassia Smeets and Anthony J H Simons. 2010. Automated Unit Testing with Randoop, JWalk and µJava versus Manual JUnit Testing. http://staffwww.dcs.shef.ac.uk/people/A.Simons/research/reports/jwalksmeets.pdf

[53] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. 2009. Dynamic Software Updates: A VM-Centric Approach. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1542476.1542478

[54] Eli Tilevich and Yannis Smaragdakis. 2005. Binary Refactoring: Improving Code behind the Scenes. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*. Association for Computing Machinery, New York, NY, USA, 264–273. https://doi.org/10.1145/1062455.1062511

[55] Statistics How To. [n. d.]. Sign Test: Step by Step Calculation. https://www.statisticshowto.com/sign-test/#:~:text=The%20sign%20test%20compares%20the%20sizes%20of%20two%20groups.&text=The%20sign%20test%20is%20an,difference%20between%20medians%20is%20zero.

[56] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. 2000. 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 (Lecture Notes in Computer Science)*, David A. Watt (Ed.), Vol. 1781. Springer, 18–34. https://doi.org/10.1007/3-540-46423-9_2

[57] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. 2010. Automated Fixing of Programs with Contracts. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA '10)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/1831708.1831716

[58] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. 2013. Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. IEEE Press, 356–366. https://doi.org/10.1109/ASE.2013.6693094

[59] Anna-Katharina Wickert, Michael Reif, Michael Eichberg, Anam Dodhy, and Mira Mezini. 2019. A Dataset of Parametric Cryptographic Misuses. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19)*. IEEE Press, Piscataway, NJ, USA, 96–100. https://doi.org/10.1109/MSR.2019.00023

[60] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. 2010. Dynamic Code Evolution for Java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ '10)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/1852761.1852764

[61] Mu Zhang and Heng Yin. 2014. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *NDSS*.

[62] Zelin Zhao, Yanyan Jiang, Chang Xu, Tianxiao Gu, and Xiaoxing Ma. 2021. Synthesizing Object State Transformers for Dynamic Software Updates. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*. https://zelinzhao.github.io/pasta/artifact/pasta-icse2021.pdf