CURE: Automated Patch Generation for Dynamic Software Update

Zelin Zhao, Tianxiao Gu, Xiaoxing Ma*, Chang Xu, Jian Lü
State Key Laboratory for Novel Software Technology, Nanjing University
Department of Computer Science and Technology, Nanjing University
Email: {zelinzhao1105, tianxiao.gu}@gmail.com, {xxm, changxu, lj}@nju.edu.cn

***Abstract*—Dynamic software updating (DSU) aims to patch software for fixing bugs or adding functions while it is running. Before update, developers need to make a dynamic patch ready, which includes update points, state transformers and a corresponding code patch. Existing practice mostly assumes manual preparation of dynamic patches, but this process can be both time-consuming and error-prone. Some pioneer work attempts to automate this process, but cannot guarantee the generation of safe dynamic patches for most updates. This paper presents a novel approach CURE to automatically generating safe dynamic patches. CURE takes two versions of software and their test cases as input, and automatically synthesizes state transformers and selects update points. We applied CURE to 28 updates for three real-world server software. The experimental results show that CURE generated safe dynamic patches automatically and their corresponding updates achieved an 88.7% success rate, as compared to 74.3% for TOS and 61.2% for default patches.**

I. Introduction

With the popularity of online services, an increasing number of software systems need to provide continuous and stable service. However, software systems are updated frequently in order to fix bugs and add new features. At present, most updates are applied using the stop-and-restart strategy, *i.e.*, stopping the running software to apply a patch. Obviously, this stop-and-restart strategy introduces unavoidable service interruption, which is particularly unacceptable for mission-critical software [1], [2], [3], [4], [5], [6].

Dynamic software updating (DSU) addresses these problems by patching software systems while they are running. A dynamic update needs a *dynamic patch*. A dynamic patch contains a code patch, update points and state transformers. Specifically, a code patch is a detailed description of differences between the old and new versions of a program. Update points specify the timings when a dynamic update can be applied. State transformers are used to transform the program state at the update point to a new-version.

The dynamic patch is essential in accomplishing a correct and timely dynamic update. Most existing DSU systems demand programmers to manually prepare a dynamic patch. Although recently a number of DSU systems have been proposed [7], [8], [9], [10], existing approaches to automated generating dynamic patches are really rare.

*Corresponding author: Xiaoxing Ma.

Apparently, manually preparing dynamic patches is time-consuming and error-prone. Programmers need to thoroughly understand the program logic and the evolution of program states at runtime, and then specify when and how to perform the dynamic update in the dynamic patch. Specifically, programmers should first determine the update points and then focus on providing transformers at those update points as the state of a running program changes all the time.

Many DSU systems can automatically determine the update points using a safety criterion such as activeness safety [1], [2], [7], [8], and also generate default transformers using predefined rules [7], [8], [9], [11]. However, these approaches are only effective in trivial cases. To further help programmers prepare dynamic patches for non-trivial updates, Magill *et al.* proposed TOS [12] to generate state transformers. TOS observes the corresponding runtime states of the two versions of a program under same inputs and synthesize transformers. However, TOS requires programmers to collect a set of transformation examples (*i.e.*, a group of matched old and new objects) by setting corresponding points in the two versions of program. These corresponding points are also used as update points. Therefore these points are the key factor in generating high quality transformers and dynamic updating. Incompetent human works may causes some serious consequences.

In this paper, we propose an automated approach, named CURE, aiming to help programmers generate high quality dynamic patches. CURE only needs the binary code and existing test cases of the old and new versions of a program. CURE requires no human intervention. The main challenges are generating state transformers and selecting update points.

The insight of CURE is to automatically exercise the program and collect a number of high quality transformation examples rather than the manual approach used by TOS. In addition, we also use a systematically testing approach to rule out infeasible candidate update points. Specifically, CURE executes same tests on both old and new programs, and collect a set of transformation examples at some carefully determined time points. However, this automatic step may produce a huge number of transformation examples, which brings trouble to the synthesis of transformers. Thus, CURE improves the original monolithic algorithm of TOS, so that it can synthesize the transformer in an incremental and divide and conquer way.

CURE filters candidate update points by systematically

running same test cases and applying update at reachable points with the generated state transformers. Moreover, CURE makes a set of *mixed tests*, which examine new features after updating, to further rule out infeasible candidate update points.

This paper makes the following primary contributions:

1) We extend TOS to make it fully automated and be able to synthesize effective transformers by analysing a huge amount of transformation examples in acceptable time;
2) We present the design and implementation of the first fully automated approach to generating high quality dynamic patches;
3) We carry out convincing experiments to show the effectiveness of CURE and compare it with two existing approaches. The results show that CURE can generates much safer dynamic patches.

The rest of this paper is organized as follows. We introduce the background in Section II and present the design and implementation of CURE in Section III. Then, we show experimental results in Section IV. Section V gives some discussions about CURE, which is followed in Section VI by some related works. Section VII provides the conclusion of this paper.

II. BACKGROUND

In this section, we present background on dynamic software updating. In addition, we introduce two existing approaches and some observations.

A. Dynamic Software Updating

There are mainly two tasks during dynamic updating, *i.e.*, loading new code to patch the existing code, and transforming the program state to its new-version counterpart. A dynamic patch is used to request and guide the dynamic update. Specifically, a DSU system starts monitoring the running program just after receiving an update request. If the program reaches an update point, the DSU system starts patching the runtime state, *i.e.*, loading changed code according to the *code patch* and transforming the program state at the update point using the state transformers. Then the running program continues with the new version.

The time between receiving the update request and reaching an update point is the *waiting time*, which should be as short as possible as an update should be applied immediately. For this purpose, programmers can specify more update points in a dynamic patch. A new challenge is that the state transformers should be effective enough to be able to transform program states at these update points. However, it is time-consuming and error-prone for programmers to write such transformers.

A rigorous condition to determine the correctness of a dynamic update is that the transformed program state should be reachable in the execution of the new program starting from the scratch [13]. However, it is impossible to determine all possible runtime states of two versions of a program and precisely compare the runtime states. Therefore, we determine successful updates by a compromising approach, *i.e.*, checking and validating the rest execution (*i.e.*, runtime states) periodically after updating. The dynamic update is successful if there is no invalidation. To validate the rest execution, we can run a test on the old program and apply a dynamic update in the middle of the testing. The rest execution of the test has many sanity checks. We say that the dynamic update is successful if the testing passes all sanity checks.

B. Existing Approaches

Preparing code patch is well supported by existing tools and we do not focus on this. Many DSU systems support for generating *default patches*. They can automatically determine the update points using a timing restriction such as activeness safety [1], [2], [7], [8] and generate default transformers using predefined rules [8], [7], [9], [11]. Activeness safety is: for each thread, if there are no changed methods in stack, it is safe to update at this time. Default transformers copy unchanged fields and assign default values to changed fields, *e.g.* `null` for `String`, `0` for `int`. Obviously, default patches are not sufficient to ensure successful update [14].

Magill *et al.* proposed TOS [12] to automate the preparation of dynamic updating. TOS can compose reasonable transformers by learning some transformation examples. However, TOS requires programmers to collect these examples. Programmers should first specify a few corresponding program points in the old and new programs, and dump heap snapshots at these points during runtime. In *matching* phase, TOS extracts objects from snapshots and finds *key fields* for each changed class. Key fields can uniquely identify objects in a snapshot and correlate old and new objects between a snapshot pair. Then TOS matches old objects with new objects using key fields. In *synthesis* phase, TOS analyses the correlated objects to compose transformers. While updating, the human specified program points are update points in a *TOS patch*, which are a key factor in generating high quality transformers.

C. Observations

Most software updates are evolutionary instead of revolutionary, or else it would be meaningless to update dynamically. For evolutionary updates, most of the test cases are same and the change of test cases is also incremental [15], [16]. Moreover, there is a strong correspondence between the runtime program states at corresponding program points in adjacent versions of the program. This correspondent relationship can be observed by running same test case on the old and new programs. TOS utilize this correspondent relationship and we further exploit this feature to generate dynamic patches.

III. AUTOMATED GENERATION OF DYNAMIC PATCH

This section presents CURE's design and implementation.

A. Overview

Figure 1 shows the overall framework of CURE. CURE takes the old program C_o, the new program C_n, the old test cases T_o and the new test cases T_n as input. First, the *code processor* analyses C_o and C_n to generates the code patch and also instruments C_o and C_n to get variants C_o', C_n' (described in Section III-C) and C_o'' (described in Section III-D). Then,the

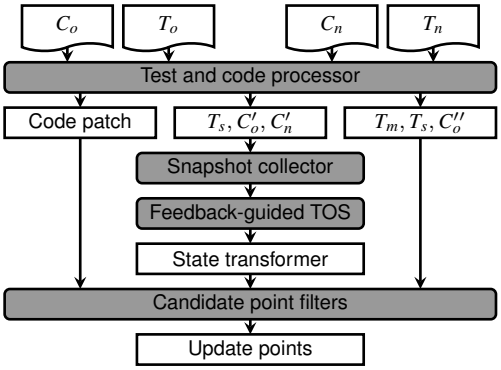


Fig. 1: The overview of CURE. T_s is used both in snapshot collector and candidate point filters.

```
1 public class DeleteTest{
2   private static File f=new File(DIR,"t.txt");
3   private static FTPClient c;
4   private void setUp(){
5     c = new FTPClient();
6     c.login("admin","password")
7   }
8   public void testDelete(){
9     f.createNewFile();
10    assertTrue(f.exists());
11    assertTrue(c.deleteFile(f.getName()));
12    assertFalse(f.exists());
13  }
14  public void tearDown(){
15    c.quit();
16  }
17 }
```

Fig. 2: A same test from Apache FtpServer, testing deleting file from FTP server.

test processor classifies test cases into *same tests* T_s and *added tests*, combines them to make *mixed tests* T_m. Next, CURE uses the *snapshot collector* to systematically execute T_s on C'_o and C'_n, dumps heap snapshots, and uses a customized TOS, *i.e.*, the *feedback-guided TOS*, to synthesize state transformers. At last, the *candidate point filters* rules out infeasible candidate update points by applying update while running T_s and T_m on C''_o, selects update points and finally generates a dynamic patch for this update.

B. Preparing Tests

CURE first compares the old tests T_o and the new tests T_n and collects a set of the same tests $T_s = T_o \cap T_n$ and a set of the added tests $T_a = T_n \setminus T_o$. After that, it composes a set of mixed tests T_m by combining tests in T_s and T_a. Here, a test is a test method written by following JUnit [17]. A typical JUnit test class has `setUp`, `tearDown` and some test methods. While executing a JUnit test method, the `setUp` runs first, then the test method and `tearDown` is the last.

Same Tests Figure 2 shows a same test `DeleteTest`, which is included in all versions of Apache FtpServer. The `setUp` method initializes an FTP client and logs in as "admin", `testDelete` method creates a file first and then deletes it and `tearDown` method disconnects the connection.

Added Tests Figure 3 shows an added test `StatTest` from Apache FtpServer 1.0.5, testing the form of a reply of the STAT command. STAT command causes a status response

```
1 public class StatTest{
2   private static FTPClient c;
3   private void setUp(){
4     c = new FTPClient();
5     c.login("admin","password")
6   }
7   public void testStat(){
8     c.stat();
9     String[] r = c.getReplyString().split("\r\n");
10    assertEquals("211-Apache␣FtpServer",r[0]);
11    assertEquals("Connected␣to␣127.0.0.1",r[1]);
12    assertEquals("Connected␣from␣127.0.0.1",r[2]);
13    assertEquals("Logged␣in␣as␣admin",r[3]);
14    assertEquals("211␣End␣of␣status.",r[4]);
15  }
16  public void tearDown(){
17    c.quit();
18  }
19 }
```

Fig. 3: An added test from Apache FtpServer 1.0.5, testing the form of a status response.

```
1 public class MixedTest{
2   private static FTPClient c;
3   private void setUp(){
4     c = new FTPClient();
5     c.login("admin","password")
6   }
7   public void mixedTest(){
8     //same test, testing delete file
9     /* omitted */
10    //added test, testing STAT
11    /* omitted */
12  }
13  public void tearDown(){
14    c.quit();
15  }
16 }
```

Fig. 4: A mixed test for Apache FtpServer 1.0.4 and 1.0.5

from server. The `testStat` method first invokes `stat` to send STAT command and then processes the reply message. From version 1.0.4 to 1.0.5, the implementation of STAT command is changed. `StatTest` passes on 1.0.5 and fails on 1.0.4.

Mixed Tests In order to examine new features after updating, CURE creates a set of mixed tests T_m by combining `setUp`, `tearDown`, a same and an added test method. For example, CURE gets `setUp`, `tearDown` and `testDelete` from `DeleteTest`, gets `testStat` from `StatTest`. Then CURE generates a mixed test, shown in Fig. 4. While running, the mixed test first executes the same test part, *i.e.* `testDelete`, then the added test part, *i.e.* `testStat`. However, some mixed tests are unreasonable, because of inconsequent program behaviors. Therefore after generating a mixed test, CURE runs it on the new program. If the mixed test fails on the new program, CURE excludes it.

C. Synthesizing State Transformers

CURE collects snapshots while running same tests, matches them, and synthesizes transformers by analyzing objects in these snapshots.

Collecting Snapshot Pairs CURE runs T_s on C'_o and C'_n. At runtime, CURE dumps a snapshot δ after modifying an object of a changed class's. We call this *Writing Object Point* (WOP).

Take an example in Fig. 5. Method `cMet` in `ClassOne` is changed and other three methods are not. All fields should be transformed while updating. CURE dumps snapshots in

```
1 public class ClassOne{
2   private int f1;
3   private String f2;
4   /* other fields, omitted */
5   public void ucMet1(){
6     UpdatePoints.ID("CP1")
7     /* omitted */
8     this.f1++;
9     WOP.ID("WOP1");
10    /* omitted */
11  }
12  public void ucMet2(String str){
13    /* omitted */
14    this.f2 = str;
15    WOP.ID("WOP2");
16    /* omitted */
17  }
18  public void ucMet3(){
19    /* omitted */
20    cMet();
21    /* omitted */
22  }
23  public void cMet(){
24    /* omitted */
25    ucMet2();
26    /* omitted */
27  }
28 }
```

Fig. 5: Setting CPs and WOPs in unchanged methods

unchanged methods, because program points in these methods are corresponding between the old and new programs. In Fig. 5, CURE sets WOPs at line 9 and 15. In **ucMet1**, **f1** is changed at line 8, so CURE sets a WOP after this line and assigns it with ID *WOP1*, which is unique to other WOPs. Also, CURE sets a WOP in **ucMet2** at line 15.

While reaching a WOP, **WOP.ID** generates a snapshot and names it uniquely, *e.g.*, *WOP2-3*. Finally, CURE collects a set of snapshot pairs Δ based on their names from all snapshots collected while running T_s on C'_o and C'_n.

Synthesizing Transformers TOS needs to first determine key fields to match objects in snapshot pairs. If there are too many objects in snapshot pairs, searching would take unacceptable memory and time, We improved TOS to propose feedback-guided TOS (FGTOS), as shown in Algorithm 1. Note that Δ is the set of snapshot pairs. For a changed class `C`, FGTOS analyses `n` snapshot pairs in each iteration. In the previous iteration, FGTOS analyses a set of snapshot pairs Δ_{pre} and finds key fields K_{pre}. `T` is the set of transformers for `C` synthesized by FGTOS.

The first time calling `FGTOS`, Δ_{pre}, K_{pre} and `T` are initialized empty. `FGTOS` randomly gets `n` snapshot pairs from Δ at line 9 and rules out them from Δ at line 10. At line 12, `FGTOS` calls TOS to find key fields K_{now} by analysing Δ_{now}. If it fails to find key fields (*i.e.* K_{now} is **null**), `FGTOS` starts next iteration at line 14. Otherwise, `FGTOS` calls itself at line 16 (in the first time, K_{pre} is **null**).

In following iterations, `FGTOS` adds Δ_{pre} to Δ_{now} at line 11 and finds new key fields K_{now} at line 12. If K_{now} equals to K_{pre}, `FGTOS` calls TOS at line 18 to match objects using K_{now} and synthesize transformer for `C`. `FGTOS` adds the transformer to *TR* and starts next iteration with empty Δ_{pre} and null K_{pre} at line 19. If the K_{now} does not equal to K_{pre}, FGTOS calls itself at line 16, passing Δ_{now} and K_{now} to next iteration.

Algorithm 1: The feedback-guided TOS (FGTOS)

```
1  Function FGTOS_CALLER(Δ, n, C)
       Input: Δ, a set of snapshot pairs, n, a number, C, a changed class
2      Δ_pre ← ∅, K_pre ← null, TR ← ∅
3      FGTOS(Δ,Δ_pre,K_pre,n,C,TR)
4  Function FGTOS(Δ,Δ_pre,K_pre,n,C,TR)
5      if Δ = ∅ then
6          if Δ_pre ≠ ∅ then
7              TR.add(TOS_synMatch_syn(Δ_pre,C))
8          return merge(TR)
9      Δ_now ← random_get_snapshot(Δ,n)
10     Δ ← Δ- Δ_now
11     Δ_now ← Δ_now ∪ Δ_pre
12     K_now ← TOS_find_keyfields(Δ_now,C)
13     if K_now = null then
14         FGTOS(Δ,Δ_now,K_pre,n,C,TR)
15     else if K_pre = null or K_now ≠ K_pre then
16         FGTOS(Δ,Δ_now,K_now,n,C,TR)
17     else if K_pre = K_now then
18         TR.add(TOS_keyMatch_syn(Δ_now,K_now,C))
19         FGTOS(Δ,∅,null,n,C,TR)
```

If `FGTOS` cannot find same K_{now} and K_{pre} or cannot find K_{now} until stop, `FGTOS` would fallback to use the *synthesis match* (*i.e.*, the function `TOS_synMatch_syn`) of TOS [12] to match objects without key fields and then synthesize transformers at line 7. The synthesis match can match objects by a transformation function that can maps old objects to new objects. The synthesis match takes more time than key field matching and gets inaccurate results. Fortunately, such a case is very rare and did not happen in our experiments.

D. Filtering Candidate Points

CURE excludes infeasible candidate points by various filters.

Static Filter CURE only installs *candidate points* (CP) at the entrance of an unchanged method. This is because (1) applying dynamic update is unsafe while changed methods are actively running [7], [9], [18], [19], [20]. (2) program states at these points are simpler than other points, (3) and we can avoid checking other ineffective points. In Fig. 5, CURE sets a candidate point with ID *CP1*, at line 6 in **ucMet1**.

In some cases, CURE installs too many CPs and filtering would takes many time. Therefore, CURE excludes candidate update points within unsafe *method distance*. Method distance between A and B is the steps that A(B) takes to invoke B(A). In Fig. 5, method distance between **cMet** and **ucMet2** is one, between **ucMet3** and **ucMet2** is two. If distance between an unchanged method and any changed methods is less than a threshold, CURE excludes this unchanged method. If threshold is one in Fig. 5, both **ucMet2** and **ucMet3** are unsafe.

Same Tests Filter CURE runs each same test on C''_o for many times and performs update at each CP. Before this step, CURE needs to record runtime information to guide the filter process. CURE runs T_s on C''_o and records runtime information for each test. Take Fig. 5 as an example, if running a test calls **ucMet1** for 3 times and the stack is safe (*i.e.* no changed methods are

running) at the first and third time, CURE records *CP1-1-safe*, *CP1-2-unsafe*, *CP1-3-safe*.

Then CURE executes this test for two times. For one execution, CURE applies update when the first time reaching *CP1* and applies update when the third time reaching *CP1* for another execution. After each execution, CURE examines the test result. If the test passed correctly, the update is applied successfully at the candidate point for this time.

If a test encounters a CP too many times, CURE may takes much time to examine it. For saving time, CURE randomly chooses some safe records. A CP is excluded if it has any update failure.

Mixed Tests Filter CURE uses mixed tests to examine new features after applying update. While running a mixed test on C_o'', CURE updates it at the same test part and this process is same as Section III-D. After updating, the mixed test continues running to test new features. New features are performed correctly after updating if the mixed test passed correctly. CURE iterates this process on all mixed tests and further excludes infeasible candidate points.

Although we show WOP and CP in Fig. 5 at the same time, CURE sets them separately for time efficiency.

E. Implementation

To create a mixed class, CURE first copies a same test class, extracts an added test method, and adds the method at the end of the same test method. In addition, CURE also automatically adds all necessary dependencies, in order to compile the mixed test and check whether it is reasonable.

CURE uses a hash map to store times of encountering each WOP. While dumping a snapshot, CURE only dumps live objects. In a few cases, running a same test on the old and new programs produces different number of snapshots. CURE excludes snapshots that cannot be matched.

CURE uses FGTOS to generate state transformers and Javelus to validate dynamic patches. Javelus provides a public method `invokeDSU`, whose parameter is the path to a dynamic patch. CURE can perform update by calling `invokeDSU`. Moreover, CURE can record runtime information into a log file. By setting some environment variables, CURE can examine a specified point, while filtering candidate points, or verifying several update points. While recording runtime information, CURE checks whether the stack is safe or not and uses a hash map to store the information.

CURE is implemented in Java and Python, comprises roughly 2200 lines of code.

IV. Experimental Evaluation

We applied CURE to generate dynamic patches for three widely used applications to show its effectiveness. All experiments run on a computer equipped with 2.4 GHz CPU (8 logical cores, 4 physical), 8GB of RAM, with Kubuntu 14.04 (kernel 3.13.0-45).

A. Subjects

All subjects were collected as follows. First, we focused on evaluating CURE on open source standalone server applications or components of such applications. Apache FtpServer is a popular FTP server. Apache Empire-db is a database abstraction layer. Apache Commons Net is a library that includes various useful Internet protocols. We considered 6 updates to FtpServer, 9 updates to Empire-db and 13 updates to Commons Net. Second, these applications should have a set of high quality test cases. We downloaded source code of all subjects from GitHub [21] and built each version and their corresponding test cases. Third, users expect continuous and stable service from these applications. For FtpServer and Empire-db, dynamic updates can avoid disruption and downtime. Many applications use Commons Net to provide Internet connection service, while updating them, Commons Net also should be updated if a new version is available. The lines of code increased from 20K+ to 21K+ for FtpServer, from 23K+ to 26K+ for Empire-db and from 34K+ to 38K+ for Commons Net.

In column Changed of table I, C, F and M show the number of changed classes, fields and methods. In column Test, S and M show the number of same tests and mixed tests. Some updates have no added test, therefore the number of mixed tests are 0 and CURE did not use mixed tests filter for them, *e.g.* update 1.0.2/1.0.4 of FtpServer.

B. State Transformers

The column Transformer shows results of generating state transformers. CURE sets same number of WOPs in C_o' and C_n', which is shown in column WOP . In update 3.5-rc1/3.5-rc2 of Commons Nets, only one class is changed and there are no fields in it. Therefore CURE did not set WOP for this update.

Column SPT shows the number of snapshots dumped after running T_s on C_o' and C_n'. For some updates of Commons Net, CURE created 0 snapshots, because running same tests does not reach any WOPs. Therefore, CURE used default transformers in these updates. For such updates, programmers can add some test cases on purpose with little efforts to help CURE generates state transformers. For other updates, FGTOS can analyse a huge amount of snapshot pairs and generate high quality transformers, but TOS cannot.

C. Update Points

Column Candidate points shows results of filtering candidate points. Column SF shows the number of remained CPs after static filter. As we mentioned in Section III-D, CURE excludes some CPs by the threshold of unsafe method distance. Considering time efficiency, if there are too many CPs and test cases, CURE sets higher threshold. CURE set threshold 3 for FtpServer, 1 for Empire-db. For Commons Net, CURE set threshold as different values in the range 1 to 6.

Column STF shows the number of remained candidate points after same tests filter. For update 3.0/3.0.1 of Commons Net, CURE excluded most candidate points, because of the default transformers. For update 3.2/3.3 of Commons Net,

TABLE I: Information of each update

Application	Updates (old/new)	Changed			Test		Transformer		Candidate Points			Time(hour)	
		C	F	M	S	M	WOP	SPT	SF	STF	MTF	TR	CP
Apache FtpServer	1.0.0/1.0.1	12	43	20	586	1,172	63	74,118	361	29	3	2.62	12.33
	1.0.1/1.0.2	5	45	8	588	1,764	33	228,392	425	46	2	8.88	10.05
	1.0.2/1.0.3	2	3	3	591	0	2	1,182	499	147	-	0.76	11.8
	1.0.3/1.0.4	6	15	5	583	583	13	53,798	396	103	4	1.82	14.17
	1.0.4/1.0.5	16	90	27	584	5,256	115	255,538	386	26	4	6.07	8.53
	1.0.5/1.0.6	18	47	35	588	4,116	44	103,380	368	86	0	3.04	13.6
Apache Empire-db	2.0.5/2.0.6	65	191	151	85	340	296	478	547	86	38	0.08	0.46
	2.0.6/2.0.7	19	48	20	81	810	298	308	799	83	70	0.05	0.9
	2.0.7/2.1.0	82	333	248	84	924	116	3,922	420	86	56	0.27	1.18
	2.1.0/2.2.0	102	348	272	80	1,040	396	310	482	86	0	0.63	0.41
	2.2.0/2.3.0	67	299	88	86	516	405	1,788	702	74	69	0.72	1.01
	2.3.0/2.4.0	59	268	83	92	184	366	5,476	729	87	84	0.25	1.06
	2.4.0/2.4.1	42	204	90	94	282	313	4,508	851	72	61	0.66	1.66
	2.4.1/2.4.2	46	201	46	95	95	248	1,428	848	77	72	0.23	1.91
	2.4.2/2.4.3	15	58	17	95	0	244	4,802	1,118	65	-	0.48	1.45
Apache Commons Net	3.0-rc1/3.0-rc2	6	61	10	143	143	58	3,064	967	87	87	0.14	1.56
	3.0-rc2/3.0-rc3	5	48	8	144	0	40	0	975	200	-	0.03	0.53
	3.0-rc3/3.0	1	17	1	143	429	18	0	1,528	89	70	0.03	3.35
	3.0/3.0.1	2	30	2	146	0	23	0	1,143	2	-	0.03	0.49
	3.0.1/3.1-rc1	45	344	105	141	3,102	379	16,916	878	95	19	1.43	4.49
	3.1-rc1/3.1	29	236	75	149	447	290	11,766	1,117	87	55	0.35	2.91
	3.1/3.2	29	241	63	149	2,086	290	11,760	1,119	89	38	1.35	1.68
	3.2/3.3	23	152	100	162	648	216	3,867	994	0	-	3.19	0.53
	3.3/3.4-rc1	48	344	108	149	14,304	291	12,500	1,016	80	19	1.39	0.42
	3.4-rc1/3.4-rc2	6	36	10	245	0	27	48	1,204	26	-	0.05	0.92
	3.4-rc2/3.4	5	9	9	245	0	8	0	1,072	26	-	0.05	0.91
	3.4/3.5-rc1	109	856	510	186	11,532	710	21,630	510	53	12	0.62	1.43
	3.5-rc1/3.5-rc2	1	0	2	248	0	0	0	1,621	311	-	0	0.95

Column Updates shows old and new version numbers; In column Changed, C, F and M are the number of changed classes, fields and methods; In column Test, S and M are the number of same tests and mixed tests; In column Transformer, WOP is the number of writing object points in the old program (the same in the new program) and SPT is the number of snapshots dumped after run same tests on two versions of the program. In column CP, SF, STF and MTF are the number of remained candidate points after static filter, same test filter and mixed test filter; In column Time, TR and CP is the time of generating transformers and filtering update points.

CURE excluded all candidate points and programmers should pay more attention to such updates.

Column MTF shows the number of remained points after mixed tests filter. For 1.0.5/1.0.6 of FtpServer and 2.1.0/2.2.0 of Empire-db, CURE excluded all candidate points, because they are too complex for FGTOS to generate effective transformers. In such situations, the success rate of each candidate point in mixed tests filter can still help programmers and we show this in Section IV-E.

For most updates, CURE excluded infeasible candidate points effectively and the remained points are update points.

D. Efficiency

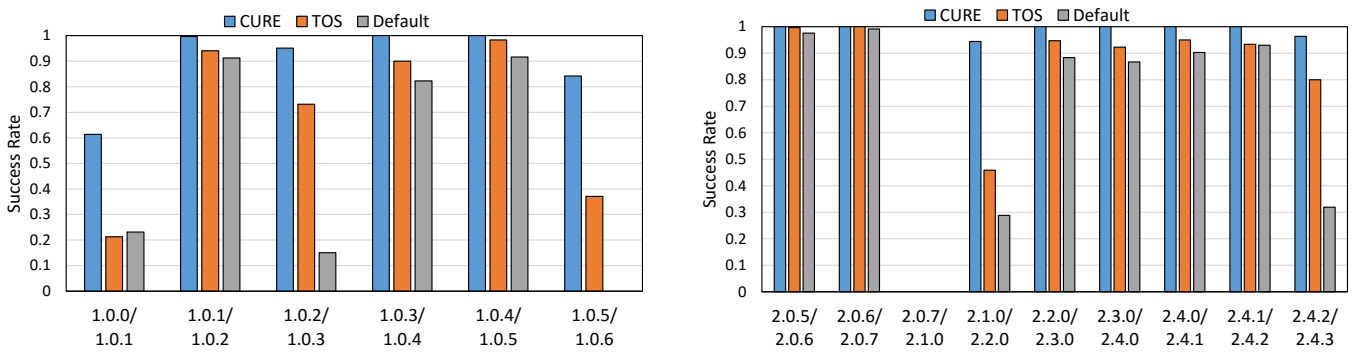
In column Time, TR shows the time of generating state transformers and CP shows the time of filtering candidate points. Obviously, CURE spent more time on FtpServer than other two programs. There are two reasons. First, CURE dumped much more snapshots for FtpServer and took longer time to synthesize state transformers. On average, CURE spent 3.87 hours for FtpServer, 0.37 hours for Empire-db and 0.67 hours for Commons Net. Second, filtering candidate points took very long time too. FtpServer has more test cases and each one takes more time to run. On average, CURE spent 11.75 hours for FtpServer, 1.12 hours for Empire-db and 1.55 hours for Commons Net.

We also used the original TOS when generating transformers. However, TOS took over 12 hours and cannot generate state transformers for most updates. In many cases, an out of memory error happened. Our FGTOS can effectively reduce the running time and memory usage.

E. Validation

We validated CURE's dynamic patches and compared it with TOS and default patches. In our experiments, we used activeness safety and default transformers in default patches. The dumped snapshots were too many for TOS to generate transformers, therefore we used CURE's transformers in TOS patches and randomly chose 10 update points each time updating. This can represent the average of human interventions. We executed the C_o'' for N times and updated it with a dynamic patch. Then we record the number M of finishing update successfully. The success rate of the dynamic patch was the ratio of M and N. Commons Net is a library and implements many internet protocols. There is no long-time running script to test each protocol, hence we only validated dynamic patches on FtpServer and Empire-db.

For FtpServer, we wrote a script in Python. This script first initializes an FTP server and six users connect to it. Each user transfers 10 files to the FTP server, then renames, downloads and deletes the 10 files. Moreover, all of them use each FTP command [22] for random times. Between two operations, each thread sleeps for a random time to simulate operation intervals of users. Executing this script takes about 30 seconds and produces about 3000 operations. The script checks the result of each operation and an update fails if any result is wrong. For each update of FtpServer, we run the script for



(a) Apache FtpServer

(b) Apache Empire-db

Fig. 6: Success rate of updating with CURE, TOS and default dynamic patches.

300 times and requested update with a dynamic patch at each second for 10 times.

There are several examples provided along with each version of Empire-db. We use the *empire-db-example-basic* example in our experiment. This example utilizes many core functions of Empire-db and we extended it. For each update, we run this example 500 times and requested the dynamic update at each 100 milliseconds for 5 times.

In our experiment, CURE excluded all candidate points for 1.0.5/1.0.6 of FtpServer and 2.1.0/2.2.0 of Empire-db. For such updates, the success rate of each candidate point can still help programmers. We used the top 10 candidate points with the highest success rate in CURE's dynamic patches.

Figure 6 shows the average success rate. The update 1.0.5/1.0.6 of FtpServer, each time updating with the default patch was failed. Although CURE excluded all candidate points for this update and 2.1.0/2.2.0 of Empire-db, the top 10 points can also achieve great success rate. The update 1.0.0/1.0.1 of FtpServer, CURE's dynamic patch only achieved 61.3% success rate, but it is still about 40% higher than other two approaches. Update 2.0.7/2.1.0 of Empire-db is too complex for FGTOS to generate state transformers, hence we used default transformers in CURE's and TOS patches. The success rates of this update were 0%. For most updates, updating with CURE's dynamic patches are much safer than TOS and default patches. In total, the average of success rate is 88.7% for CURE, 74.3% for TOS and 61.2% for default patches.

V. Discussion

In this paper, we proposed CURE for generating dynamic patches automatically. From the results of our experiments, we can see that CURE can effectively generate dynamic patches in acceptable time.

A. Test Cases

CURE utilizes test cases to generate transformers and exclude candidate points. The efficiency and effectiveness of CURE is highly bound up with them. There may be some redundant test cases. In the future, we can improve CURE by excluding these redundant tests first.

Test cases may be insufficient to validate a dynamic patch in few cases in our experiments. If there are no tests, we cannot use CURE to generate dynamic patch. If tests are prepared insufficiently, CURE may generate mediocre dynamic patch. In Fig. 6, there are a few failures while updating directly with CURE's dynamic patches. The main reason is that test cases cannot represent real behaviors of users. Moreover, sanity checks in test cases are not designed for dynamic updates. However, using test cases is a cost-effective and time-saving way. Programmers can add some representative test cases to improve CURE's performance. Besides, they can improve the dynamic patch according to their experience and priori knowledge with little efforts. In future work, we plan to utilize not only test cases but also real behaviour of users. We may try to use some existing record and replay tools [23], [24], [25], [26] to record the execution traces while users using programs.

B. Generating State Transformers

CURE dumps snapshots at WOPs in unchanged methods. If there are no snapshot, CURE cannot generate any state transformer. In our experiments, CURE did not generate a transformer for several updates of Commons Net due to insufficient test cases. Moreover, if objects are created and changed only in changed methods, CURE cannot generate transformers too. It is difficult to map corresponding program points between changed methods. We plan to use execution indexing and access path [27], [28] to map objects directly.

VI. Related Work

Many existing tools [29], [30] can generate code patches for traditional stop-and-restart updating. For dynamic updating, many existing DSU systems [7], [8], [9], can only generate default transformers depending on syntactic changes. TOS [12] can generate more effective state transformers based on program synthesis. However, TOS depends on manually specified updates points (only a few in most cases) and thus its algorithm cannot scale to a large set of candidate update points. CURE uses feedback to guide TOS to improve its scalability.

Most DSU systems apply a timing restriction to select a proper update point. Hayden *etc.* [14] evaluates three timing restrictions using systematic testing: *i.e.*, (1) *activeness*

safety [1], [2], [7], [8], (2) *con-freeness safety* [9], and (3) manual identification, and find that manual identification is the most effective among the three. Tedsuto [31] is a another framework for testing dynamic patches. However, Tedsuto depends on human interventions in many aspects and also requires to custom the DSU system, which are inconvenient and time-consuming. CURE is a fully automated approach that can generate and test dynamic patches and also be non-intrusively applied in most DSU systems without any customization on the DSU system. As stated in [13], the correctness of a dynamic update cannot be decided automatically in general. Thus, many techniques [32], [33] that can improve the reliability of dynamic updating systems could be used in cooperation with an automated updating preparation tool such as CURE.

VII. Conclusion

This paper presents CURE, an automated approach to generating dynamic patches. By testing and profiling, CURE can generate a dynamic patch, including state transformers and their applicable update points. CURE makes a novel mixed testing method to exclude infeasible candidate update points. Our evaluation has shown that for most updates, CURE generates high quality dynamic patches than other state-of-the-art approaches. Specifically, CURE achieves 88.7% success rate, which is 14.4% higher than TOS and 27.5% higher than default patches.

Acknowledgments

This work was supported in part by National Basic Research 973 Program (Grant #2015CB352202), National Natural Science Foundation (Grants #61472177, #91318301, #61321491) of China. The authors would also like to thank the support of the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

References

[1] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz, "Opus: Online patches and updates for security," in *Usenix Security*, vol. 5, 2005, p. 18.

[2] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in *Proceedings of the ACM European conference on Computer systems*, 2009, pp. 187–198.

[3] D. Scott, "Assessing the costs of application downtime," *Gartner Group, May*, 1998.

[4] "Software horror stories," http://www.cs.tau.ac.il/ nachumd/horror.html, accessed: 30-June-2016.

[5] "Collection of software bugs," http://www5.in.tum.de/ huckle/bugse.html, accessed: 30-June-2016.

[6] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "Inside the slammer worm," *IEEE security and privacy*, vol. 1, no. 4, pp. 33–39, 2003.

[7] S. Subramanian, M. Hicks, and K. S. McKinley, "Dynamic software updates: a vm-centric approach," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 1–12.

[8] T. Gu, C. Cao, C. Xu, X. Ma, L. Zhang, and J. Lu, "Javelus: a low disruptive approach to dynamic software updates," in *Asia-Pacific Software Engineering Conference*, 2012, pp. 527–536.

[9] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for c," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006, pp. 72–83.

[10] C. M. Hayden, K. Saur, E. K. Smith, M. Hicks, and J. S. Foster, "Kitsune: Efficient, general-purpose dynamic software updating for c," pp. 249–264, 2014.

[11] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew, "Polus: A powerful live updating system," in *International Conference on Software Engineering*, 2007, pp. 271–281.

[12] S. Magill, M. Hicks, S. Subramanian, and K. S. McKinley, "Automating object transformations for dynamic software updating," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2012, pp. 265–280.

[13] D. Gupta, P. Jalote, and G. Barua, "A formal framework for on-line software version change," *IEEE Transactions on Software engineering*, vol. 22, no. 2, pp. 120–131, 1996.

[14] C. M. Hayden, E. K. Smith, E. A. Hardisty, M. Hicks, and J. S. Foster, "Evaluating dynamic software update safety using systematic testing," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1340–1354, 2012.

[15] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[16] L. White, "Insights into regression testing," *Proceedings of the Conference on Software Maintenance-1989, pages60-69. IEEE Computer Society Press, October1989*.

[17] "Junit test," http://junit.org/junit4/.

[18] A. Baumann, G. Heiser, J. Appavoo, D. Da Silva, O. Krieger, R. W. Wisniewski, and J. Kerr, "Providing dynamic update in an operating system." in *USENIX Annual Technical Conference, General Track*, 2005, pp. 279–291.

[19] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu, "Mutatis mutandis: Safe and predictable dynamic software updating," *ACM Transactions on Programming Languages and Systems*, vol. 29, no. 4, p. 22, 2007.

[20] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis, "Contextual effects for version-consistent dynamic software updating and safe concurrent programming," in *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2008, pp. 37–49.

[21] "The apache software foundation," https://github.com/apache.

[22] "Apache ftpserver commands," http://mina.apache.org/ftpserver-project/ftpserver_commands.html, accessed: 30-June-2016.

[23] Y. Jiang, T. Gu, C. Xu, X. Ma, and J. Lu, "Care: Cache guided deterministic replay for concurrent java programs," in *Proceedings of the International Conference on Software Engineering*, 2014, pp. 457–467.

[24] Y. Jiang, C. Xu, and X. Ma, "Dpac: an infrastructure for dynamic program analysis of concurrency java programs," in *Proceedings of the Middleware Doctoral Symposium*, 2013, p. 2.

[25] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang, "R2: An application-level kernel for record and replay," in *Proceedings of the USENIX conference on Operating systems design and implementation*, 2008, pp. 193–208.

[26] Y. Saito, "Jockey: a user-space library for record-replay debugging," in *Proceedings of the International Symposium on Automated Analysis-Driven Debugging*, 2005, pp. 69–76.

[27] B. Xin, W. N. Sumner, and X. Zhang, "Efficient program execution indexing," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 238–248.

[28] T. G. R. L. X. Ma and Z. Zhao, "Precise heap differentiating using access path and execution index," in *Proceedings of the 15th National Software Application Conference (NASAC)*, 2016, p. to appear.

[29] J. Andersen and J. L. Lawall, "Generic patch inference," *Automated Software engineering*, vol. 17, no. 2, pp. 119–148, 2010.

[30] J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S.-C. Khoo, "Semantic patch inference," in *Automated Software Engineering (ASE)*. IEEE, 2012, pp. 382–385.

[31] L. Pina and M. Hicks, "Tedsuto: A general framework for testing dynamic software updates," in *Proceedings of the International Conference on Software Testing*, 2016.

[32] T. Gu, C. Sun, X. Ma, J. Lu, and Z. Su, "Automatic runtime recovery via error handler synthesis," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 684–695.

[33] T. Gu, Z. Zhao, X. Ma, C. Xu, C. Cao, and J. Lü, "Improving reliability of dynamic software updating using runtime recovery," in *Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC)*, 2016, p. to appear.