IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received February 1, 2020, accepted February 12, 2020, date of publication February 18, 2020, date of current version February 28, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2974670

The Effects of Static Analysis for Dynamic Software Updating: An Exploratory Study

BABIKER HUSSIEN AHMED, SAI PECK LEE, AND MOON TING SU
Department of Software Engineering, Faculty of Computer Science and Information Technology, University of Malaya, Kuala Lumpur 50603, Malaysia
Corresponding author: Sai Peck Lee (saipeck@um.edu.my)

ABSTRACT Dynamic software updating (DSU) is the act of modifying software without stopping its execution. DSU is employed to preserve the high availability in the deployed software systems. Although significant investigations have been conducted on static analysis (SA) to determine DSU errors, no particular study exists that explores the effects of SA for dynamic software updating. The objective of the study presented in this paper is to explore the effects of static analysis for DSU. In this exploration, four evaluation metrics were declared including the number of update operations, the number of violations, the cyclomatic complexity, and the patch size. Also, a novel framework was introduced and 18 versions of baseline target programs were employed to explore the effects of static analysis for DSU. The results have explained that static analysis can detect violations of DSU. It may affect the complexity of the new versions of the target programs and having optimization can further reduce those violations. In addition, the results show that the SA may change the number of update operations and the patch size for DSU. Overall, the results have proved that SA for dynamic software updating could affect the complexity of the target programs, the number of update operations, and the patch size. Furthermore, this exploration has offered a novel framework and four evaluation metrics for measuring the effects of static analysis for DSU.

INDEX TERMS Dynamic software updating, program analysis, static analysis.

I. INTRODUCTION

Dynamic software updating (DSU) is the act of modifying software systems without having to terminate its execution [1], [2]. Experts have always considered DSU among the mandatory techniques for supporting the high availability of software systems [3], [4]. Recently, multiple solutions demonstrate DSU approaches [8], including the ability to analyze DSU statically before the trigger of the actual dynamic update procedures [6].

Static analysis (SA) is a program analysis approach to find software flaws without executing the program code [7]. Conspicuously, most early studies show that the SA helps to prove the correctness of dynamic software updating [8] and to ensure the safety of DSU [9], [10].

Although extensive research has been carried out on static analysis for DSU [6], [8]–[10], the effects of SA for dynamic software updating remain an open question. It is not obvious which parameters will be affected when applying static analysis for DSU. In this investigation, an exploratory study has been conducted to explore the effects of static analysis for dynamic software updating.

For this study, a novel framework was presented and four evaluation metrics were demonstrated to measure the effects of static analysis for DSU. The evaluation metrics includes the number of update operations [62], the number of violations [11]–[13], the cyclomatic complexity [14]–[16], and the patch size [17]–[19].

For this study, 18 versions of baseline target programs [6], [20]–[23] were employed to explore the effects of static analysis for DSU. The inspected target programs were extracted from 6 versions of benchmarks software including Apache SSHD 0.3, Apache SSHD 0.4, Zt-zip 1.4, Zt-zip 1.5, Apache Tomcat 8.5.37, and Apache Tomcat 8.5.38.

It is apparent from the outcome of this exploration is that the SA can help to detect violations to DSU. It may affect the complexity of the new versions of the target programs and having optimization can further reduce those violations. In addition, the results show that the static analysis may change the number of update operations and the patch size for dynamic software updating.

In overall, this study explored the effects of static analysis for DSU and has the following contributions:

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

- Demonstrate four evaluation metrics to measure the effects of static analysis for dynamic software updating.
- Introduce a novel framework for measuring the effects of static analysis for dynamic software updating.
- Measure the effects of static analysis for DSU.

This paper is organized as follows. Section II highlights the background and related works. Section III demonstrates the exploratory study. Section IV presents the conducted experiments. Section V presents the results and discussion. Section VI declares the principal findings. Section VII discusses threats to the validity of this study. Section VIII presents conclusions and future work.

II. BACKGROUND AND RELATED WORK

This section presents the background of static analysis, static analysis for dynamic software updating, and the related work.

A. STATIC ANALYSIS

Many works have recognized that program analysis plays an important role in finding software flaws [7], [26]–[28]. Program analysis can be implemented either statically, dynamically, or in a combination of both [24], [25]. Static analysis is (SA) is a program analysis approach useful in finding software errors without having to execute the program code [7], [24]. On the other hand, dynamic program analysis is performed during the program runtime [25].

Several studies have shown that the static analysis assist in runtime error detection [29], refactoring [30], code completion [31], and program understanding [31]. In addition, SA is employed for code conventions [31], code safety [10], code optimization [33], and measurements [32].

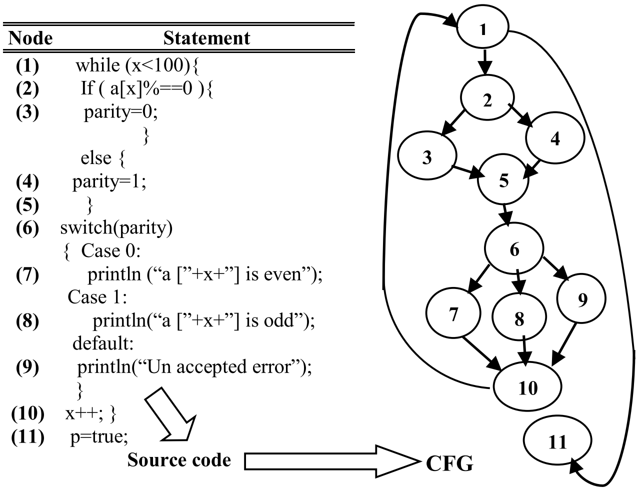
The static analysis could be implemented in three ways: intraprocedural for analyzing the code of one method [25], interprocedural for analyzing a program with multiple methods [25], and local static analysis for analyzing a basic block code in a method [25].

To perform SA, the target source code should be converted into a directed graph [34], though some studies also suggested an undirected graph [35]. In a directed graph, the connections between vertices are one-way and can be visualized as arrows linking pairs of vertices [35].

In the light of reported SA, the control flow graph (CFG) [36]–[38] is utilized for the static analysis. A control flow graph (CFG) is a directed graph shows program control flows during the execution of the program [36]–[38]. Fig.1 shows an example of CFG [38].

In addition, there is a wide choice of graphs demonstrated in earlier examinations including connection dependence graph [40], program dependence graph [37], flow dependence graph [37], control dependence graph [37], call graph [37], and data flow graph [39].

In brief, the static analysis includes control flow analysis (CFA) [41] and data flow analysis (DFA) [39]. Moreover, some studies suggested applying optimization in the target programs is significant to ensure the correctness of the target programs [10] and for achieving high performance [10].



Node	Statement
(1)	while (x<100){
(2)	If (a[x]%==0){
(3)	parity=0;
	}
	else {
(4)	parity=1;
(5)	}
(6)	switch(parity)
	{ Case 0:
(7)	println (“a [”+x+”] is even”);
	Case 1:
(8)	println(“a [”+x+”] is odd”);
	default:
(9)	println(“Un accepted error”);
	}
(10)	x++; }
(11)	p=true;

FIGURE 1. An example of a control flow graph (CFG) [38].

B. STATIC ANALYSIS FOR DYNAMIC SOFTWARE UPDATING

As shown in Fig.2, conceptually, dynamic software updating consists of two parts [42], the offline preparation part for preparing the code and the online updating part to handle the actual run time updating [42].

In the offline preprocessing part, where SA is utilized, a dynamic patch needs to be prepared to cover the transformation of the code and data between the old and the new versions of the target program. In the online updating part, patch processing will be implemented through data updating, code updating, and safety checks.

It was reported in the literature that the static analysis for dynamic software updating is anticipated to achieve the following objectives but not limited to:

- Predict the exact requirements of an update by knowing what exactly has been changed in the new version of the target program [6].
- Ensure utilizing adequate update points [9], [43].
- Guarantee the safety of applying dynamic software updating [9], [10].
- Prove the correctness of dynamic software updating [8].
- Predict which dynamic software updating solution might work better than others [6].

C. RELATED WORK

In recent years, various approaches have been proposed to utilize static analysis for dynamic software updating [6], [8], [10], [26]. In this section, we highlight several studies that are extremely relevant to this study.

Šelajev *et al.* [6] formalized an approach for SA for dynamic software updating. The study [6] defined an approach for producing lists of the exact changes between versions of Java programs and examined if a given DSU system supports that update.

Also, Zhenxing *et al.* [8] proposed a static analysis method to guarantee the correctness of dynamic software updating and to make DSU easier and more flexible. In addition,

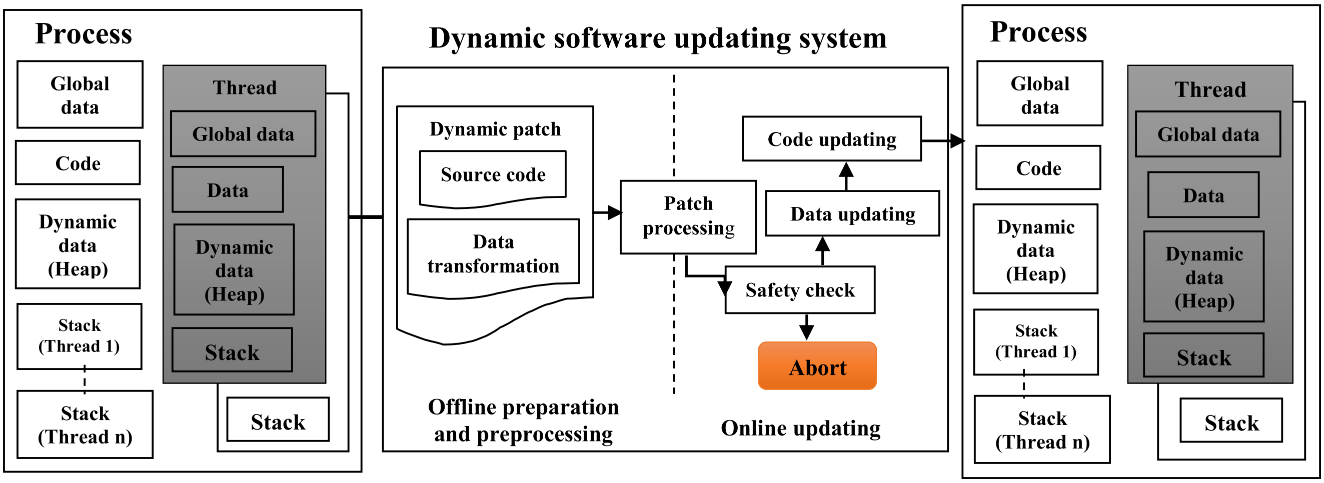


FIGURE 2. Conceptual dynamic software updating [42].

TABLE 1. The research questions and objectives of this study.

#	Research question	Objective
RQ 1	What evaluation metrics can be utilized to measure the effects of static analysis for dynamic software updating?	1. To propose evaluation metrics to measure the effects of static analysis for dynamic software updating.
RQ 2	What are the effects of static analysis for dynamic software updating?	2. To measure the effects of static analysis for dynamic software updating.

Li and Ogawa [26] presented and demonstrated an on-the-fly interprocedural program analysis algorithm called the sliding-window algorithm for efficiency and scalability in whole-program analysis on runtime [26].

Moreover, Sreedhar *et al.* [10] presented a novel framework called the extant analysis framework for interprocedural optimization. The study [10] described properties for DSU safety tests and provided algorithms for their generation and placement [10].

As presented in the related studies [6], [8], [10], [26], most studies in the static analysis for DSU are in the business of proposing a new approach [6], a method [8], an algorithm [26], and a framework [10]. To the best of our knowledge, no study to date has demonstrated evaluation metrics to measure the effect of static analysis for dynamic software updating. Respectfully, to highlight the effects of static analysis for DSU, the research domain requires exploration (such as this study) to understand more the effects of static analysis for dynamic software updating.

III. EXPLORATORY STUDY

This section declares the purpose of this exploration and presents measuring the effects of static analysis for dynamic software updating.

A. PURPOSE OF THE STUDY

This study aims to explore the effects of static analysis for dynamic software updating. The research questions (RQ) and the objective of each RQ are illustrated in Table 1.

B. MEASURING EFFECTS OF STATIC ANALYSIS FOR DYNAMIC SOFTWARE UPDATING

This sub-section highlights the proposed evaluation metrics to answer RQ 1 and demonstrates a suggested framework to respond to RQ 2.

1) EVALUATION METRICS

For this exploration, we borrow the idea of using the computed metrics for two versions of a system to assess the evolution [44] to measure the effects of static analysis for dynamic software updating. In addition, we propose to utilize four evaluation metrics, includes the number of update operations [62], the number of violations [11]–[13], Cyclomatic Complexity (CC) [14]–[16], and the patch size [17]–[19].

a: THE NUMBER OF UPDATE OPERATIONS

In this analysis, we propose to compare 2 versions of one target program for code similarities using the available Diff algorithms [62] and count the number of update operations (NOUO) as follows:

$$\mathbf{NOUO} = \sum \text{Update operations} \tag{1}$$

Throughout this paper, we use the acronym NOUO to refer to the number of update operations.

b: THE METRIC OF THE NUMBER OF VIOLATIONS

In this analysis, we propose to utilize the metrics of the number of violations [11] due to the following:

- Measuring the number of violations is a common practice in assessing coding standards that are strongly related to latent faults [11]–[13].
- Previous research has emphasized violation measures for DSU such as evaluating type safety violations 8].

Suppose we analyze the code of the target program which resulted from the update of old and the new code. The total number of violations (TNOV) can be calculated as

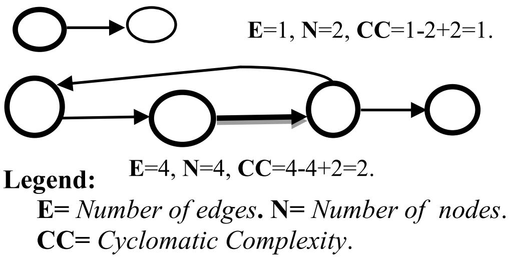

E=1, N=2, CC=1-2+2=1.
E=4, N=4, CC=4-4+2=2.
Legend:
E= Number of edges. N= Number of nodes.
CC= Cyclomatic Complexity.

FIGURE 3. Cyclomatic complexity metric examples.

follows:

$$\textbf{TNOV} = \sum \text{VAL} \tag{2}$$

where

VAL *is a violation of a static analysis rule.*

c: THE METRIC OF PATCH SIZE

The patch as a set of changes to a computer program or its supporting data designed to update, fix, or improve the program [45], [46]. The patch size is the sum of several contains including the size of the code being inserted [17], supplementary inserted codes, and annotations [17], [18].

In this analysis, we suggest to use the metric of the patch size [17]–[19] due to the following reasons:

- A large number of existing studies in the broader literature have examined patches in to request and guide the dynamic updating [42], [47].
- Research has provided evidence for loading time in DSU is proportional to the patch size [18].

For the current assessment, the metric of the patch size (per byte) as is calculated as follows:

$$\textbf{SP} = \text{Size of the patch (per byte)} \tag{3}$$

where

SP *is the patch size.*

d: CYCLOMATIC COMPLEXITY METRIC

For this examination, we suggest utilizing the cyclomatic or McCabe complexity (CC) [14]–[16] metric to the following reasons:

- Cyclomatic complexity is considered to predict software components that likely have a high defect rate or that might be difficult to test and maintain [14], [44].
- The majority of prior research has applied complexity measures to help establish risk and stability estimations on an item of code [14].
- An increasing number of investigations have explained that CC is accomplished by measuring the control flow structure in a directed graph [8], [14], [48], [49].

Previous studies have shown that CC measures the number of linearly independent paths in a code [8], [14], [48], [49]. CC is defined as the number of edges minus the number of nodes plus 2 [17] as follows:

$$\textbf{CC} = \text{e} - \text{n} + 2 \tag{4}$$

where **e** *is the number of edges*, **n** *is the number of nodes*. Furthermore, Fig.3 shows examples of CC.

2) FESAD: A FRAMEWORK FOR MEASURING EFFECTS OF STATIC ANALYSIS FOR DYNAMIC SOFTWARE UPDATING

For this study, we searched for literature to observe how SA for dynamic software updating is implemented. Our first set of analyses have examined the related studies [6], [8], [10], [26], and the second batch of investigations have inspected additional studies [42], [43], [50], [51] for more understanding of SA for dynamic software updating. As an outcome, a novel framework was proposed for utilizing SA for measuring the effects of static analysis for DSU and illustrated in Fig. 4.

As displayed in Fig. 4, the proposed framework is containing three phases as follows:

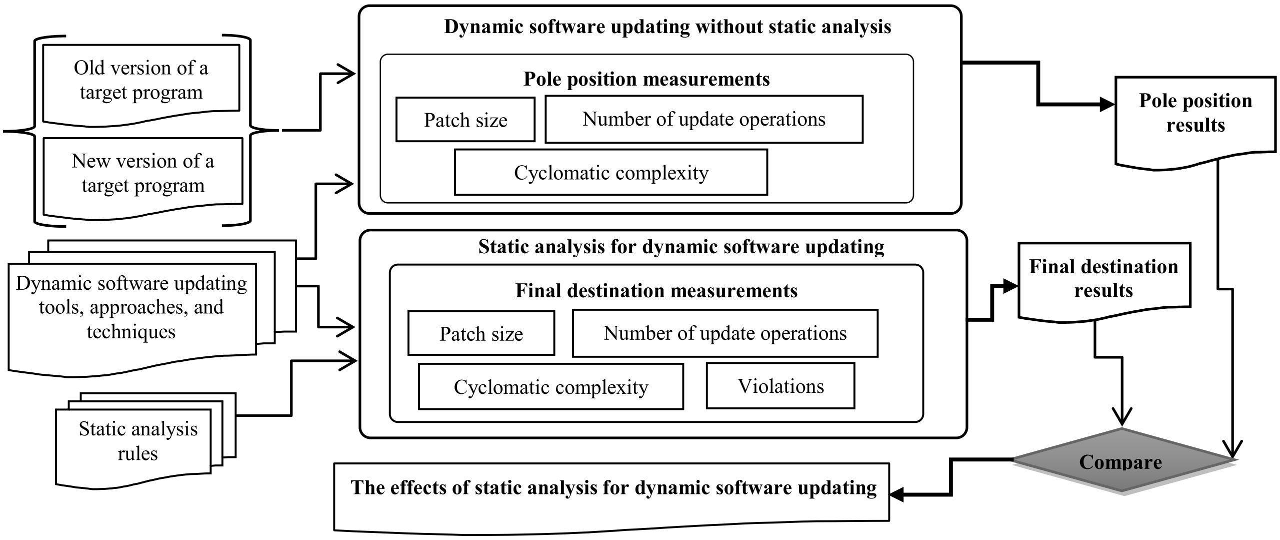

Old version of a target program
New version of a target program
Dynamic software updating tools, approaches, and techniques
Static analysis rules
Dynamic software updating without static analysis
Pole position measurements
Patch size
Number of update operations
Cyclomatic complexity
Static analysis for dynamic software updating
Final destination measurements
Patch size
Number of update operations
Cyclomatic complexity
Violations
Pole position results
Final destination results
Compare
The effects of static analysis for dynamic software updating

FIGURE 4. The High-level architecture of FESAD (Framework for measuring Effects of Static Analysis for DSU).

TABLE 2. Examples of static analysis rules available in PMD [52], [52]–[54], [56].

Type	Objective	Examples of static analysis rules
Error-prone	To avoid runtime errors.	Avoid multiple unary operators; bad comparison; close resource; dataflow anomaly; do not call garbage collection explicitly; equals null; unconditional if statement; use proper class Loader.
Multithreading	To apply and flag issues when dealing with multiple threads of execution.	Avoid synchronized at the method level; double-checked locking, not a thread-safe singleton; unsynchronized static date format; use concurrent HashMap.

a) **Phase 1:** *Dynamic software updating without SA*

In this phase, DSU will be applied without static analysis. Also, pole position (The start point) measurements [62] will be utilized to measure the following:

1) The patch size.
2) The number of update operations (NOUO).
3) The cyclomatic complexity for the target program.

b) **Phase 2:** *Static analysis for dynamic software updating*

In this phase, the static analysis will be applied for DSU. Also, the final destination (The endpoint) measurements will be utilized to measure the following:

A. The number of violations.
B. The patch size.
C. The number of update operations (NOUO).
D. The cyclomatic complexity for the target program.

Currently, a large number of existing studies have utilized the static analysis rules in static program analyzer tools such as the programmer mistake detection tool (PMD) [52]–[57]. For example, Table 2 highlights some of the available static analysis rules in PMD.

A number of authors have recognized static analysis including control flow analysis can be utilized for dynamic software updating in error prevention, detection, and removal [6], [8], [10], [26]. For example, recent research suggests that static analysis rules for DSU can be utilized for the following:

- Detection and annotating of unsafe update points [59].
- Guarantees about which types will be updatable at runtime [5].
- Predicts which types are modifiable at each update point [5].
- Ensure when an item (method, identifiers) will be removed from a class, any access to the removed item through the old code after the update is forbidden because it will lead to a runtime error [59].
- Check the signature of a class constructor which has parameters to be changed, if there is no implicit type conversion between the type of the parameters of the old and new constructor, the update is said to be unsafe [59].

In addition, some authors have driven the further improvement through possible optimization in the target programs can be included to code smells [59] and to assist in errors prevention, detection, and removal for dynamic software updating, such as dead code elimination [58], Inlining [58], copy propagation [58], constant propagation [58], strength reduction-induction variable [58], tail recursion elimination [58], and software pipelining [58].

c) **Phase 3:** *Compare the results of Phase (a) and Phase (b)*

To measure the effects of static analysis for dynamic software updating, in this investigation, we have suggested the following assessments:

A. Compare the difference between the patch size in DSU without static analysis (*Phase 1*) against patch size in SA for dynamic software updating (*Phase 2*).
B. Compare the difference between NOUO in DSU without static analysis (*Phase 1*) against NOUO in SA for dynamic software updating (*Phase 2*).
C. Compare the difference between the CC of the target program in DSU without static analysis (*Phase 1*) against the CC of the same target program in SA for dynamic software updating (*Phase 2*).

For this study, the suggested framework is utilized in the subsequent sections. Throughout this paper, we will use the acronym FESAD to refer to the proposed framework.

IV. EXPERIMENTS

This section presents the conducted experiments for measuring the effects of static analysis for DSU.

For this study, the experimental setup was prepared as follows:

- The operating system is 64-bit Windows 10, with 8 GB RAM, Intel Core 3.40 GHz, and 8-core CPU.
- Java Development Kit (JDK) 1.8.

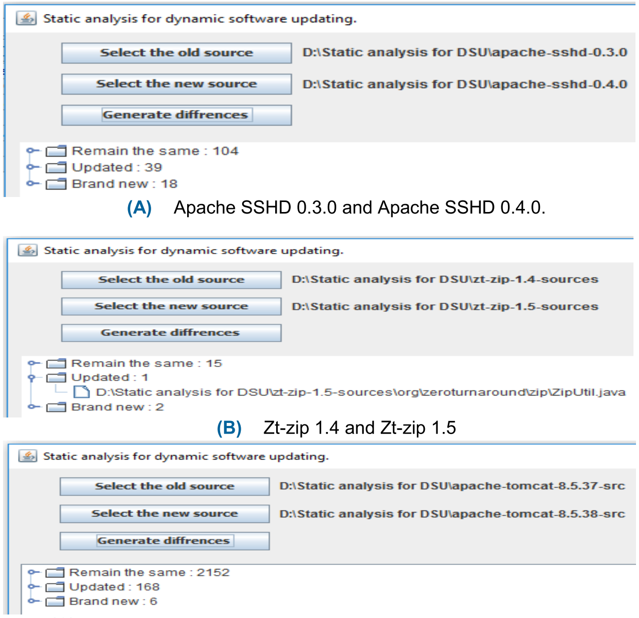
For the selections of the experimental subject, the literature review shows the majority of prior research in DSU utilizing a group of baseline target programs as a benchmark for the research in the domain [6], [20]–[23]. Among them, the following subjects were selected for this study:

1) **Apache SSHD [20], [21]:** is a Java software supports SSH protocols on both client and server-side. For this investigation, 2 versions were selected including Apache SSHD version 0.3.0 and Apache SSHD version 0.4.0.
2) **Zt-zip [6]:** is a Java software for file compression. For this examination, 2 versions were selected including Zt-zip version1.4 and Zt-zip version 1.5.
3) **Apache Tomcat [21], [23]:** is an open-source Java HTTP web server. For this exploration, 2 versions were selected including Apache Tomcat version 8.5.37 and Apache Tomcat version 8.5.38.

For this exploration, official software releases from the package repositories of the selected software were copied. Also, a list of updated programs was identified from each of the selected subjects utilizing Algorithm 1. The results of applying Algorithm1 is illustrated in Fig.5.

Algorithm 1 Extraction of the Differences Between 2 Software Versions

```
Procedure Differences between 2 software versions.
Input: Directory of old code D1, Directory of new
       code D2, Source extensions Ext;
Output: A list of the updated programs;
String [ ] extensions = Ext // examples {"java", "c"}
Boolean recursive = true;
Collection files = list Files in (D1, Ext, recursive);
For  (Iterator iter = files. iter; iter. has Next () )
    File f1 = (File) iter. Next
    If (f1 exists in D1) and (f1 exists in D2)
        File f2 =new File (D2, f1. Name)
        If (f1 <> f2)
            Save f1 in the list of updated programs;
            else
            Save f1 in the list of duplicated
            programs;
        end if
        else
        Save f1 in the list of new programs;
    end if
end for
end procedure
```



Static analysis for dynamic software updating.

Select the old source: D:\Static analysis for DSU\apache-sshd-0.3.0

Select the new source: D:\Static analysis for DSU\apache-sshd-0.4.0

Generate diffrences

Remain the same : 104
Updated : 39
Brand new : 18

(A) Apache SSHD 0.3.0 and Apache SSHD 0.4.0.

Static analysis for dynamic software updating.

Select the old source: D:\Static analysis for DSU\zt-zip-1.4-sources

Select the new source: D:\Static analysis for DSU\zt-zip-1.5-sources

Generate diffrences

Remain the same : 15
Updated : 1
D:\Static analysis for DSU\zt-zip-1.5-sources\org\zeroturnaround\zip\ZipUtil.java
Brand new : 2

(B) Zt-zip 1.4 and Zt-zip 1.5

Static analysis for dynamic software updating.

Select the old source: D:\Static analysis for DSU\apache-tomcat-8.5.37-src

Select the new source: D:\Static analysis for DSU\apache-tomcat-8.5.38-src

Generate diffrences

Remain the same : 2152
Updated : 168
Brand new : 6

(C) Apache Tomcat 8.5.37 and Apache Tomcat 8.5.38

FIGURE 5. The extraction of the differences between the versions of the selected subjects.

In addition, the details of the updated target programs as shown in Fig. 5 are as follows:

- **For Apache SSHD version 0.3.0 and Apache SSHD 0.4.0:** A total of 104 programs were not updated (Remains the same), 39 programs were updated, and 18 new programs.
- **For Zt-zip version 1.4 and Zt-zip 1.5:** An aggregate of 15 programs were not updated, one program was updated, and 2 new programs.
- **For Tomcat version 8.5.37 and Apache Tomcat 8.5.38:** A collection of 2152 programs was not updated, 168 programs were updated, and 6 new programs.

TABLE 3. Selected target programs.

Subject	Target program	The number of lines of codes (LOC)	
		The old version	The new version
Apache SSHD	ClientSessionImpl	212	303
	ChannelSession	317	352
	ServerSession	318	342
	SshServer	260	268
Zt-zip	ZipUtil	542	605
Apache Tomcat	BasicAuthenticator	105	114
	JNDIRealm	838	887
	WsHttpUpgradeHandler	103	103
	WsServerContainer	157	174

TABLE 4. Results of pole position results when dynamic software updating without static analysis.

Subject	Target program	Pole position measurements	
		NOUO	Patch size (Per byte)
Apache SSHD	ClientSessionImpl	32	1854
	ChannelSession	20	1051
	ServerSession	18	905
	SshServer	28	1382
Zt-zip	ZipUtil	25	1425
Apache Tomcat	BasicAuthenticator	10	550
	JNDIRealm	10	734
	WsHttpUpgradeHandler	10	619
	WsServerContainer	19	952

For this study, 18 target programs were selected from the nominated subjects and presented in Table 3. Also, the FESAD framework was utilized and the results are presented and discussed in Section V.

V. RESULTS AND DISCUSSION

This section presents and discusses the experimental results of this exploration including pole position results, final destination results, data analysis, and comparison of the collected results.

A. POLE POSITION RESULTS

As illustrated in the FESAD framework, the initial set of analyses explorers applying DSU without static analysis in the given experimental subjects. Table 4 presents pole position results including NOUO (Number of update operations) and patch size when DSU was implemented in the selected target programs without static analysis.

Also, Table 5 presents the Cyclomatic Complexity (CC) for the selected target programs when DSU was implemented in the selected target programs without SA.

B. FINAL DESTINATION RESULTS

As represented in the FESAD framework, the second batch of the analyses explores applying SA for dynamic software

TABLE 5. Results of pole position results when dynamic software updating without static analysis.

Subject	Target program	Cyclomatic Complexity (CC)	
		The old version	The new version
Apache	ClientSessionImpl	44	93
SSHD	ChannelSession	83	66
	ServerSession	56	59
	SshServer	66	80
Zt-zip	ZipUtil	148	163
Apache	BasicAuthenticator	29	25
Tomcat	JNDIRealm	237	277
	WsHttpUpgradeHandler	14	14
	WsServerContainer	40	43

TABLE 6. Number of violations during a static analysis for DSU.

Subject	Target program	Using PMD		Related to DSU
		The old version	The new version	
Apache	ClientSessionImpl	10	10	2
SSHD	ChannelSession	11	14	2
	ServerSession	12	12	2
	SshServer	5	6	1
Zt-zip	ZipUtil	20	17	3
Apache	BasicAuthenticator	2	5	1
Tomcat	JNDIRealm	6	6	1
	WsHttpUpgradeHandler	4	4	0
	WsServerContainer	6	7	0

TABLE 7. Number of violations after utilizing optimization IN static analysis for DSU.

Subject	Target program	Using PMD	Related to DSU
		The new version	
Apache	ClientSessionImpl	2	0
SSHD	ChannelSession	1	0
	ServerSession	5	0
	SshServer	1	0
Zt-zip	ZipUtil	6	0
Apache	BasicAuthenticator	1	0
Tomcat	JNDIRealm	2	0
	WsHttpUpgradeHandler	2	0
	WsServerContainer	2	0

updating. Therefore, we applied static analysis for DSU utilizing the PMD tool and the static analysis rules in Section III. As a result, Table 6 shows the number of violations when static analysis for DSU was implemented in the selected target programs.

In addition, we have implemented optimizations to fix the violations of dynamic software updating in the new versions of the selected target programs. Table 7 shows the number of violations after optimizations in the new versions of the selected target programs.

Moreover, Table 8 declares the results of the CC metrics after SA is utilized for dynamic software updating in the new versions of the selected target programs.

Furthermore, Table 9 shows NOUO (Number of update operations) and the patch size when the static analysis for DSU was implemented in the selected target programs.

C. DATA ANALYSIS

Before proceeding to compare the pole position results against the final destination results, it will be necessary to analyze the data. The analysis includes the obtained data of NOUO, patches size, the number of violations, and CC.

TABLE 8. Results of final destination results after static analysis for DSU was implemented.

Subject	Target program	Cyclomatic Complexity (CC) of the new version
Apache	ClientSessionImpl	67
SSHD	ChannelSession	92
	ServerSession	59
	SshServer	80
Zt-zip	ZipUtil	161
Apache	BasicAuthenticator	25
Tomcat	JNDIRealm	275
	WsHttpUpgradeHandler	14
	WsServerContainer	43

TABLE 9. Results of final destination results when static analysis for DSU was implemented.

Subject	Target program	Final destination measurements	
		NOUO	Patch size (Per byte)
Apache	ClientSessionImpl	35	2017
SSHD	ChannelSession	21	1097
	ServerSession	23	1175
	SshServer	23	1564
Zt-zip	ZipUtil	68	4285
	BasicAuthenticator	14	841
Apache	JNDIRealm	16	1016
Tomcat	WsHttpUpgradeHandler	10	619
	WsServerContainer	19	952

1) THE NUMBER OF UPDATE OPERATIONS

As illustrated in Table 4, the average of the number of update operations (NOUO) is approximately 19 in the pole position results. On the other hand, as presented in Table 9, the average of NOUO is relatively increased to 25 in the final destination results. In addition, as shown in Fig.6, we exhibited the results in a Clustered Column Chart.

As displayed in Fig.6, X-axis presents the selected target programs. Y-axis displays the NOUO. The clustered columns show the pole position and the final destination results. The results show that the NOUO numbers are changed in most of the selected target programs in the final destination results.

2) THE PATCH SIZE

As depicted in Table 4, the max patch size is 1845 bytes and the minimum patch size is 550 bytes in the pole position results. On the other hand, as displayed in Table 9, the max patch size is 4285 bytes and the minimum patch size is 619 bytes in the final destination results. Moreover, as illustrated in Fig.7, we presented the results in a Clustered Bar Chart.

As demonstrated in Fig.7, Y-axis presents the selected target programs. X-axis presents the patch size (per byte). The clustered columns show the pole position and the final destination results. The results show that the patch size is

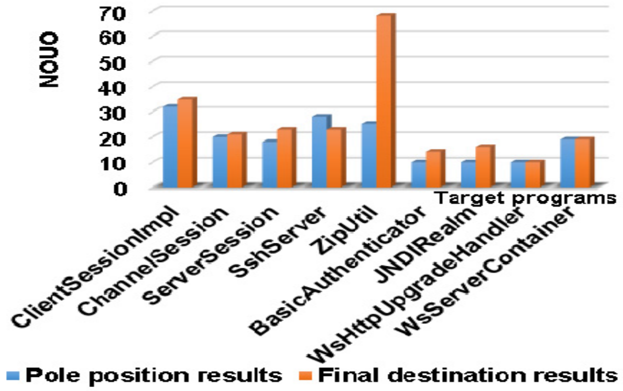


FIGURE 6. The number of update operations (NOUO) in the selected target programs.

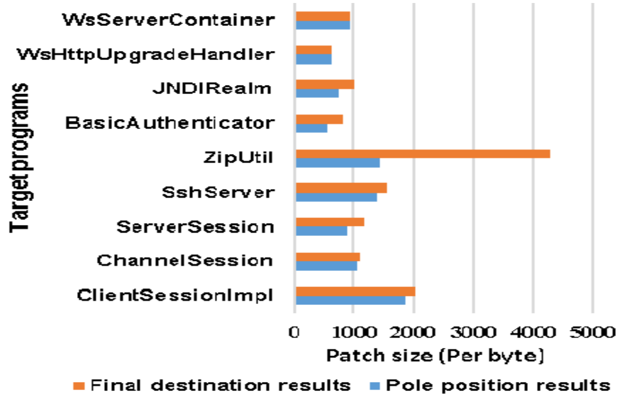


FIGURE 7. Patch size (Per byte) in the selected target programs.

amended in most of the selected target programs in the final destination results.

3) THE NUMBER OF VIOLATIONS

As illustrated in Table 6, the average of the number of violations is 9 when applying SA before optimizations. On the other hand, as presented in Table 7, the average is relatively decreased to 2 when applying static analysis with optimizations. In addition, as presented in Fig.8, we demonstrated the results in a Clustered Column Chart.

As displayed in Fig.8, X-axis presents the selected target programs. Y-axis displays the number of violations. The clustered columns show the number of violations before and after SA and optimizations. The results show that the number of violations is decreased in all the selected new target programs after optimizations.

4) CYCLOMATIC COMPLEXITY

As depicted in Table 5, the max number of cyclomatic complexity (CC) in the new versions of the target programs is 277 and the minimum value is 14 in the pole position results. On the other hand, as shown in Table 8, the max number of CC in the new versions of the target programs is 275 and the

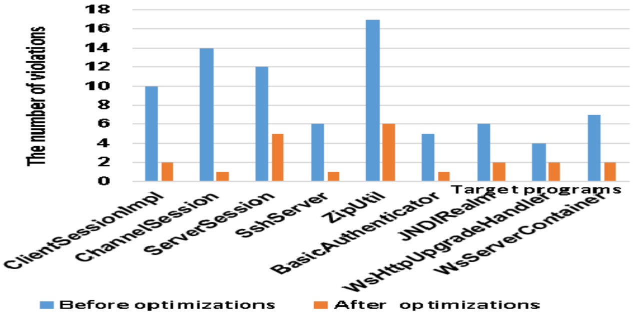


FIGURE 8. The number of violations in the selected target programs.

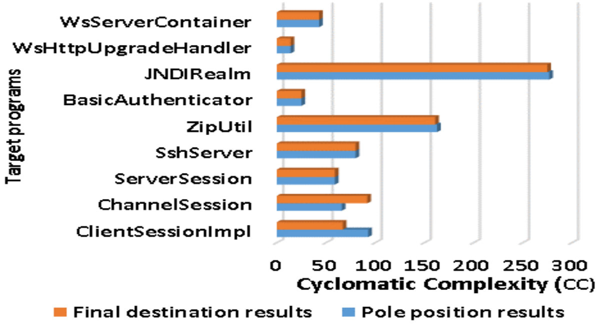


FIGURE 9. Cyclomatic Complexity (CC) of the new versions of selected target programs.

minimum value is 14 in the final destination results. Furthermore, as demonstrated in Fig.9, we presented the results in a Clustered Bar Chart.

As displayed in Fig.9, X-axis presents the selected target programs. Y-axis displays the CC of the new versions of the target programs. The clustered columns show the pole position and the final destination results. The results show that the values of CC are changed in the final destination results for 4 target programs including the new versions of ClientSessionImpl, ChannelSession, JNDIRealm, and ZipUtil.

D. COMPARE THE POLE POSITION RESULTS AND THE FINAL DESTINATION RESULTS

As demonstrated in the FESAD framework, the third batch of the analyses compares the results of pole position results against the final destination results. As an outcome, we observed the following:

- As shown in Table 4, Table 9 and Fig.7, the sizes of patches are changed after SA for most of the selected target programs except for WsServerContainer and WsHttpUpgradeHandler. Overall, out of 9 patches, only 2 remains in the same size. Thus, the data supports the premise that the static analysis may change the patch size for dynamic software updating.
- As presented in Table 6, Table 7, and Fig.8, the number of violations for the same target programs were changed which determines that the static analysis is identifying violations (Table 6) to DSU and

having optimization (Table 7) can further reduce those violations.

- As presented in Table 4, Table 9 and Fig.6, the NOUO are changed after SA was implemented for most of the selected target programs except for WsServerContainer and WsHttpUpgradeHandler. From the results, it is clear that the static analysis may change the number of update operations for dynamic software updating.
- As displayed in Table 5, Table 8, and Fig.9, the CC values not modified after static analysis for DSU was implemented for 4 new versions of target programs including, ZipUtil, ClientSessionImpl, ChannelSession, and, JNDIRealm. The most striking result to emerge from the data is that the SA for dynamic software updating may change the complexity of the new versions of the target programs.

VI. PRINCIPAL FINDINGS

Our results cast a new light on static analysis for dynamic software updating through an exploratory study where four evaluation metrics were declared, a novel framework was introduced, and 18 versions of baseline target programs were employed to explore the effects of static analysis for DSU. Based on our results, we declare the main findings of our exploration as follows:

- The results verified that static analysis for dynamic software updating determines violations of DSU. This key conclusion validates the usefulness of investigating and fixing errors early to reduce violations when applying DSU. The present finding seems to be consistent with other research that found that static analysis will help to prove the correctness of DSU [8] and to ensure the safety of DSU [9], [10].
- The most remarkable finding was that SA for dynamic software updating may change the patch size. As a consequence, this finding is in accordance with findings that measure the update loading time based on the size of the patch [18].
- The most interesting finding was that SA for dynamic software updating may affect the complexity of the target programs. The present findings are directly in line with previous findings that found the changes in the complexity of the program will affect the maintainability of the program [16], and will directly affect the eligibility and the reliability of the software [16]. Also, this finding can be compared with an argument made by Bierman *et al.* [60] that updatable programs must be reliable, yet updating itself introduces further complexity, also, Bierman *et al.* [60] reveals that to prevent total confusion, techniques are required for ensuring that the dynamic updates are in some sense safe [60].

VII. THREATS TO VALIDITY

Although this study shows the core of our exploration of the effects of static analysis for dynamic software updating, there are threats to the validity of the results that readers should take into account when interpreting the outcome. The threats include the conclusion, internal and external validity. This section clarifies threats to the validity of this study.

A. INTERNAL VALIDITY

Internal validity takes place when it implies that we proposed a framework based on our experience in dynamic software updating. To mitigate this threat, we introduced our framework (FESAD) based on several investigations [6], [8], [10], [26] , [42], [43], [50], [51] to identify how static analysis for DSU was demonstrated in those studies.

B. EXTERNAL VALIDITY

External validity takes place when it implies the validity of our target programs, the way the target programs were updated, and the code patches utilized, whereby the results might be dissimilar if different parameters are used and limit the generalization of our results.

We cannot claim that the results presented in Java programs are valid for other target programs than Java. To minimize those threats, in our exploratory study, we used 18 real target programs with a real update that was used before in other dynamic software updating studies [6], [20]–[23].

C. CONCLUSION VALIDITY

For conclusion validity, we are not aware of biases we may have had when interpreting the results. The reader should be aware of the impact of our interests on the study. To mitigate this threat, four evaluation metrics were demonstrated to draw the conclusion utilizing well-known metrics including the number of violations [11]–[13], the cyclomatic complexity [14]–[16], and the patch size [17]–[19].

VIII. CONCLUSION AND FUTURE WORK

In recent years, there has been increasing interest in modifying software systems without termination, i.e. dynamic software updating (DSU) [1], [2]. One of the respected examinations in this domain is the static analysis for DSU [5], [6], [8], [10], [26]. Experts have always considered DSU among the mandatory techniques for supporting the high availability of software systems [3], [4].

Although significant investigations have been conducted on static analysis (SA) to determine DSU errors, no particular study exists that explores the effects of SA for dynamic software updating. Respectfully, this study explored the effects of static analysis for dynamic software updating.

Static analysis (SA) is a program analysis approach to find software flaws without executing the program code [7]. Conspicuously, most early studies show that the SA helps to prove the correctness of dynamic software updating [8] and to ensure the safety of DSU [9], [10].

In this study, a novel framework was presented and four evaluation metrics were demonstrated to measure the effects of static analysis for DSU. The evaluation metrics includes the number of update operations [62], the number of violations [11]–[13], the cyclomatic complexity [14]–[16], and the patch size [17]–[19].

In this study, 18 versions of baseline target programs [6], [20], [21]–[23] were employed to explore the effects of static analysis for DSU. The inspected target programs are extracted from 6 versions of benchmarks software including Apache SSHD 0.3, Apache SSHD 0.4, Zt-zip 1.4, Zt-zip 1.5, Apache Tomcat 8.5.37, and Apache Tomcat 8.5.38.

It is apparent from the outcome is that the SA can help to detect violations to DSU. It may affect the complexity of the new versions of the target programs and having optimization can further reduce those violations. In addition, the results show that the static analysis may change the number of update operations and the patch size for dynamic software updating.

Looking forward, further attempts could prove quite beneficial to search to reduce the potential effects of static analysis for dynamic software updating. Overall, the results of this study will help the researchers and practitioners in carrying out static analysis for dynamic software updating with more understanding of its effects.

REFERENCES

[1] C. M. Hayden, K. Saur, M. Hicks, and J. S. Foster, "A study of dynamic software update quiescence for multithreaded programs," in *Proc. 4th Int. Workshop Hot Topics Softw. Upgrades (HotSWUp)*, Jun. 2012, pp. 6–10.

[2] H. Seifzadeh, H. Abolhassani, and M. S. Moshkenani, "A survey of dynamic software updating," *J. Softw., Evol. Process*, vol. 25, no. 5, pp. 535–568, May 2013.

[3] M. Weisbach, N. Taing, M. Wutzler, T. Springer, A. Schill, and S. Clarke, "Decentralized coordination of dynamic software updates in the Internet of things," in *Proc. IEEE 3rd World Forum Internet Things (WF-IoT)*, Dec. 2016, pp. 171–176.

[4] M. Neumann, C. Bach, A. Miclaus, T. Riedel, and M. Beigl, "AlwaysOn Web of things infrastructure using dynamic software updating," in *Proc. 7th Int. Workshop Web Things (WoT)*, Stuttgart, Germany, 2016, pp. 1–6.

[5] G. Stoyle, "A theory of dynamic software updates," Tech. Rep., 2007.

[6] O. Šelajev, R. Raudjärv, and J. Kabanov, "Static analysis for dynamic updates," in *Proc. 9th Central Eastern Eur. Softw. Eng. Conf. Russia (CEE-SECR)*, Oct. 2013, p. 7.

[7] N. Truong, P. Roe, P. Bancroft, "Static analysis of students' Java programs," in *Proc. 6th Australas. Conf. Comput. Educ.*, vol. 30, Jan. 2004, pp. 317–325.

[8] Y. Zhenxing, Z. Zhixiang, and B. Kerong, "An approach to dynamic software updating for java," in *Proc. IEEE Pacific-Asia Workshop Comput. Intell. Ind. Appl.*, Dec. 2008, pp. 930–934.

[9] M. Hicks and S. Nettles, "Dynamic software updating," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 6, pp. 1049–1096, Nov. 2005.

[10] V. C. Sreedhar, M. Burke, and J. D. Choi, "A framework for interprocedural optimization in the presence of dynamic class loading," *ACM SIGPLAN Notices*, Vol. 35, no. 5, pp. 196–207, Aug. 2000.

[11] Y. Takai, T. Kobayashi, and K. Agusa, "Software metrics based on coding standards violations," in *Proc. Joint Conf. 21st Int. Workshop Softw. Meas. 6th Int. Conf. Softw. Process Product Meas.*, Nov. 2011, pp. 273–278.

[12] Z. Lin, D. Marinov, H. Zhong, Y. Chen, and J. Zhao, "JaConTeBe: A benchmark suite of real-world java concurrency bugs (T)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Nov. 2015, pp. 178–189.

[13] B. Lucia, J. Devietti, L. Ceze, and K. Strauss, "Atom-aid: Detecting and surviving atomicity violations," *IEEE Micro*, vol. 29, no. 1, pp. 73–83, Jan. 2009.

[14] C. Ebert and J. Cain, "Cyclomatic complexity," *IEEE Softw.*, vol. 33, no. 6, pp. 27–29, Nov. 2016.

[15] D. Landman, A. Serebrenik, and J. Vinju, "Empirical analysis of the relationship between CC and SLOC in a large corpus of java methods," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, Sep. 2014, pp. 221–230.

[16] T. Honglei, S. Wei, and Z. Yanan, "The research on software metrics and software complexity metrics," in *Proc. Int. Forum Comput. Sci.-Technol. Appl.*, Dec. 2009, pp. 131–136.

[17] A. Tamches and B. P. Miller, "Fine-grained dynamic instrumentation of commodity operating system kernels," in *Proc. 3rd Symp. Oper. Syst. Design Implement. (OSDI)*, Berkeley, CA, USA, Feb. 1999, pp. 117–130.

[18] I. Neamtiu and M. Hicks, "Safe and timely updates to multi-threaded programs," *ACM SIGPLAN Notices*, vol. 44, no. 6, p. 13, May 2009.

[19] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for c," *ACM SIGPLAN Notices*, vol. 41, no. 6, p. 72, Jun. 2006.

[20] H. Chen, J. Yu, C. Hang, B. Zang, and P.-C. Yew, "Dynamic software updating using a relaxed consistency model," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 679–694, Sep. 2011.

[21] T. Gu, C. Cao, C. Xu, X. Ma, L. Zhang, and J. Lü, "Low-disruptive dynamic updating of java applications," *Inf. Softw. Technol.*, vol. 56, no. 9, pp. 1086–1098, Sep. 2014.

[22] J. Shen and R. A. Bazzi, "A formal study of backward compatible dynamic software updates," in *Software Engineering and Formal Methods*. Cham, Switzerland: Springer, 2015, pp. 231–248.

[23] T. Gu, C. Cao, C. Xu, X. Ma, L. Zhang, and J. Lu, "Javelus: A low disruptive approach to dynamic software updates," in *Proc. 19th Asia–Pacific Softw. Eng. Conf.*, Dec. 2012, pp. 527–536.

[24] C. Wang, J. Hill, J. Knight, and J. Davidson, "Software tamper resistance: Obstructing static analysis of programs," Univ. Virginia, Charlottesville, VA, USA, Tech. Rep. CS-2000-12, 2000, pp. 1–18.

[25] C. Artho and A. Biere, "Combined static and dynamic analysis," *Electron. Notes Theor. Comput. Sci.*, vol. 131, pp. 3–14, May 2005.

[26] X. Li and M. Ogawa, "A sliding-window algorithm for on-the-fly interprocedural program analysis," in *Proc. Int. Conf. Formal Eng. Methods*. Cham, Switzerland: Springer, 2017, pp. 281–297.

[27] A. Tomb, G. Brat, and W. Visser, "Variably interprocedural program analysis for runtime error detection," in *Proc. Int. Symp. Softw. Test. Anal. (ISSTA)*, Jul. 2007, pp. 97–107.

[28] R. Lounas, M. Mezghiche, and J.-L. Lanet, "Towards a general framework for formal reasoning about java bytecode transformation," 2013, *arXiv:1307.8212*. [Online]. Available: http://arxiv.org/abs/1307.8212

[29] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel, "Context-sensitive program analysis as database queries," in *Proc. 24th ACM SIGMOD-SIGACT-SIGART Symp. Princ. Database Syst. (PODS)*, Jun. 2005, pp. 1–12.

[30] F. Wedyan, D. Alrmuny, and J. M. Bieman, "The effectiveness of automated static analysis tools for fault detection and refactoring prediction," in *Proc. Int. Conf. Softw. Test. Verification Validation*, Apr. 2009, pp. 141–150.

[31] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 419–428, Jun. 2014.

[32] G. Altekar, I. B. P. Bagrak, A. Schultz, "OPUS: Online patches and updates for security," in *Proc. USENIX Secur. Symp.*, Aug. 2005, pp. 287–302.

[33] R. Heckmann and C. Ferdinand, "Worst-case execution time prediction by static program analysis," in *Proc. 18th Int. Parallel Distrib. Process. Symp. (IPDPS)*, Apr. 2004, pp. 26–30.

[34] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *Proc. Eur. Conf. Object-Oriented Program.* Berlin, Germany: Springer, 1995, pp. 77–101.

[35] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, "Graph structure in the Web," *Comput. Netw.*, vol. 33, nos. 1–6, pp. 309–320, Jun. 2000.

[36] E. Hosnieh and H. Haga, "A novel approach to program comprehension process using slicing techniques," *J. Comput.*, vol. 11, no. 5, pp. 353–365, 2016.

[37] C. Sandberg, A. Ermedahl, J. Gustafsson, and B. Lisper, "Faster WCET flow analysis by program slicing," *ACM SIGPLAN Notices*, vol. 41, no. 7, pp. 103–112, Jul. 2006.

[38] J. Cardoso, "How to measure the control-flow complexity of Web processes and workflows," in *Workflow Handbook*, L. Fischer, Ed. Lighthouse Point, FL, USA: Future Strategies, 2005, pp. 199–212.

[39] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proc. 22nd ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, Jan. 1995, pp. 49–61.

[40] R. A. Ballance and A. B. Maccabe, "Program dependence graphs for the rest of us," Ph.D. dissertation, Dept. Comput. Sci., College Eng., Univ. New Mexico, Albuquerque, NM, USA, 1992.

[41] F. Nielson and H. R. Nielson, "Interprocedural control flow analysis," in *Proc. Eur. Symp. Program.* Berlin, Germany: Springer, 2002, pp. 20–39.

[42] V. Ilvonen, P. Ihantola, and T. Mikkonen, "Dynamic software updating techniques in practice and Educator's guides: A review," in *Proc. IEEE 29th Int. Conf. Softw. Eng. Edu. Training (CSEET)*, Apr. 2016, pp. 86–90.

[43] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu, "Mutatis mutandis: Safe and predictable dynamic software updating," *ACM SIGPLAN Notices*, vol. 40, no. 1, pp. 183–194.

[44] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *J. Softw. Maintenance Evol., Res. Pract.*, vol. 19, no. 2, pp. 77–131, Mar. 2007.

[45] Patch (Computing). (Nov. 19, 2019). *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Patch_(computing)

[46] *What is a Bug Fix?—Definition from Techopedia*. Accessed: Jul. 29, 2015. [Online]. Available: https://www.techopedia.com/definition/18105/bug-fix

[47] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[48] R. Ghiya and L. J. Hendren, "Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in," in *Proc. 23rd ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, Jan. 1996, pp. 1–15,

[49] M. W. Hall and K. Kennedy, "Efficient call graph analysis," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 3, pp. 227–242, Sep. 1992.

[50] E. K. Smith, M. Hicks, and J. S. Foster, "Towards standardized benchmarks for dynamic software updating systems," in *Proc. 4th Int. Workshop Hot Topics Softw. Upgrades (HotSWUp)*, Jun. 2012, pp. 11–15.

[51] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci, "StaDynA: Addressing the problem of dynamic code updates in the security analysis of android applications," in *Proc. the 5th ACM Conf. Data Appl. Secur. Privacy*, Mar. 2015, pp. 37–48.

[52] E. Söderberg, T. Ekman, G. Hedin, and E. Magnusson, "Extensible intraprocedural flow analysis at the abstract syntax tree level," *Sci. Comput. Program.*, vol. 78, no. 10, pp. 1809–1827, Oct. 2013.

[53] P. Louridas, "Static code analysis," *IEEE Softw.*, vol. 23, no. 4, pp. 58–61, Jul. 2006.

[54] S. Herbold, J. Grabowski, and S. Waack, "Calculation and optimization of thresholds for sets of software metrics," *Empirical Softw. Eng.*, vol. 16, no. 6, pp. 812–841, Dec. 2011.

[55] H. Prahofer, F. Angerer, R. Ramler, H. Lacheiner, and F. Grillenberger, "Opportunities and challenges of static code analysis of IEC 61131-3 programs," in *Proc. IEEE 17th Int. Conf. Emerg. Technol. Factory Autom. (ETFA)*, Sep. 2012, pp. 1–8.

[56] *PMD Source Code Analyzer Project*. Accessed: Jan. 2, 2019. [Online]. Available: https://pmd.github.io/latest/pmd_userdocs_making_rulesets.html

[57] V. Balachandran,"Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, May 2013, pp. 931–940.

[58] J. S. Seng and D. M. Tullsen, "The effect of compiler optimizations on Pentium 4 power consumption," in *Proc. 7th Workshop Interact. Between Compilers Comput. Archit.*, Feb. 2003, pp. 51–56.

[59] M. J. Kordkandi, "Towards change validation in dynamic system updating frameworks," Ph.d. dissertation, Università degli Studi di Milano, Milan, Italy, 2018.

[60] S. Zhang and L. Huang, "Formalizing class dynamic software updating," in *Proc. Sixth Int. Conf. Qual. Softw. (QSIC)*, Oct. 2006, pp. 1–17.

[61] J. C. Munson and T. M. Khoshgoftaar, "Measuring dynamic program complexity," *IEEE Softw.*, vol. 9, no. 6, pp. 48–55, Nov. 1992.

[62] G. Canfora, L. Cerulo, and M. D. Penta, "Identifying changed source code lines from version repositories," in *Proc. 4th Int. Workshop Mining Softw. Repositories*, May 2007, p. 14.

[63] M. Pukall, C. Kästner, W. Cazzola, S. Götz, A. Grebhahn, R. Schröter, and G. Saake, "JavAdaptor-flexible runtime updates of java applications," *Softw., Pract. Exper.*, vol. 43, no. 2, pp. 153–185, Feb. 2013.

BABIKER HUSSIEN AHMED received the Bachelor of Computer Science degree (Hons.) and the Master of Science degree in software engineering from the Sudan University of Science and Technology (SUST), Sudan, in 2004 and 2009, respectively. He is currently pursuing the Ph.D. degree with the Department of Software Engineering, Faculty of Computer Science and Information Technology, University of Malaya (UM). His research interests include dynamic software updating, graph theory, program analysis, software containers, and cloud computing.

SAI PECK LEE received the master's degree in computer science from the University of Malaya, the Diplôme d'études approfondies (D.E.A.) degree in computer science from the Université Pierre et Marie Curie (Paris VI), and the Ph.D. degree in computer science from Université PanthéonSorbonne (Paris I). She is currently a Professor with the Faculty of Computer Science and Information Technology, University of Malaya. She has authored or coauthored an academic book, a few book chapters, and over 100 articles in various local and international conferences and journals. Her current research interests in software engineering include object-oriented techniques and CASE tools, software reuse, software fault localization, requirements engineering, application and persistence frameworks, and software traceability and clustering. She has been an Active Member in the reviewer committees and programme committees of several local and international conferences. She is also in several Experts Referee Panels, both locally and internationally.

MOON TING SU received the Bachelor of Computer Science degree (Hons.) and the Master of Science degree in computer science from University Putra Malaysia, and the Ph.D. degree in computer science from The University of Auckland. She is currently a Senior Lecturer with the Department of Software Engineering, Faculty of Computer Science and Information Technology (FCSIT), University of Malaya (UM), Malaysia. Her research interests are architecture knowledge, e-learning, web services, microservices, virtual reality for the Internet, computer aided software engineering (CASE) tools (syntax-directed programming editor/programming environment), end-user programming, reusable requirements, object-oriented software systems, and cloud computing.