Feature-based Software Customization: Preliminary Analysis, Formalization, and Methods

Yufei Jiang, Can Zhang, Dinghao Wu, and Peng Liu
The Pennsylvania State University
University Park, PA 16802, USA
{yzj107,czhang,dwu,pliu}@ist.psu.edu

***Abstract*—Modern software engineering practice allows us to build more complex software than ever before. However, on the other hand, it causes some negative consequences such as bloatware and feature creep which have been observed in many software evolution and iteration lifecycle. In this paper, we propose an approach to customizing Java bytecode by applying static dataflow analysis and enhanced programming slicing technique. This approach allows developers to customize Java programs based on various users' requirements or remove unnecessary features from tangled code in legacy projects. We evaluate our approach by conducting case studies on removing cross cutting features from real world Java programs. The results show that our approach has the potential for practical use. Additionally, we find out that, by increasing the diversity of the software, our approach can help achieve moving target defense.**

I. Introduction

A typical modern software system delivers a set of features to its users in a bundled way. The requirements of removing and customizing one or some of those bundled features are raised from both developers and users, for both software engineering reasons and software security reasons.

From software engineering perspective, with the rapid development of modern software engineering, the complexity of modern software keeps increasing, which causes the problem known as bloatware [30]. A type of bloatware is caused by the feature creep phenomenon. Taking Microsoft Word as an example. Besides the function of text editing, users can also use MS Word to read and send emails. However, people seldom use MS Word in this way. Many other software products encounter similar situations. According to a survey, on average, more than 45% functions of a software product are never used by most users [22].

Apparently, feature-creep bloatware causes many negative consequences. It has larger size, higher code complexity, and potentially is less reliable and secure. Developers are difficult to change and maintain the code. It also implies longer testing time, more bug reports and releasing patches in a higher frequency.

From software security perspective, a feature-based software customization is required to respond to several different threats. In current trust model, users or developers can use analysis tools to scan the applications or libraries. Such scanning helps improve our confidence on the integrity of the applications or libraries but cannot give a guarantee. Once the decision of using a specific application or library is made, we *have to* 100% trust the behavior of those code as a whole. If a feature-based customization solution is available, we do not need to fully trust the application or the library even if we choose to use them. By removing some user-identified unnecessary features which might have sensitive behavior such as writing data to Internet or logging user' profile, we can achieving active defense. In addition, removing unnecessary features based on different requirements not only reduces the attack surface, but also achieves moving target defense by increasing software diversity.

Existing technologies and previous research cannot solve this problem very well. The effort of automatically removing bloat from bloatware has been made by some researchers. However, they did not touch the bloatware that is caused by feature creep. Some research focus on code local optimization [4], [15], [29]. For example, Xu [29] raises a methodology on addressing bloatware problem by identifying reusable data structures. Some other researchers try to eliminate bloat from the code size perspective. For example, Tip et al. [23] develop Jax to remove unused methods and fields from Java programs with the help of call graph and manual annotation. However, a feature is neither a class nor a component, which has different granularity compared with a program feature. Research on change impact analysis and code refactoring partially touches this problem. However, change analysis does not transform program itself. It identifies which part of test cases are affected by newly added functions and need to be replayed. Automatic code refactoring transforms program to enhance the performance, readability, and reliability of the code without changing its behavior. However, in our research question, we want to customize and transform the program to *change* the behavior of the program.

In this paper, we propose a novel approach to conducting feature-based program customization via multistep static analysis. One of the steps of our approach is based on an enhanced program slicing method called *solo slicing*. Based on a set of seed methods defining a feature, our approach investigates its call sites, return value, and parameters. Then starting from the return values and parameters at call sites, we remove any code that depends on return values, and any code on which *only* the parameters depend. Next, we remove call site itself. At last, if possible, we remove method definition by checking a set of rules.

More specifically, we use return values in the seed method call sites as our forward slicing criteria to find out all statements that depend on the return values on both data flow and control flow. By removing these statements, we guarantee the program is still runnable after seed method call sites removal.

We use parameters in the seed method call sites as our solo slicing criteria to find out all statements on which *only* the parameters depend via data flow or control flow. By removing these statements, we trim off the redundancy caused by the absence of the seed methods call sites.

We evaluate our methodology by conducting case studies on several real-world Java applications. We aim to remove the network connection feature, database connection feature, and logging feature from those applications respectively. The results of the case studies show that our approach is correct and effective. In summary, we make the following contributions:

- We define the feature and the problem of features-based customization.
- We propose a multistep static analysis which is based on enhanced program slicing technology to perform feature-based customization.
- We identify several features that are prone to be interwoven with other code and contains security-sensitive behavior.
- We conduct a preliminary case study to evaluate the feasibility, correctness, and effectiveness of our approach.

The rest of the paper is organized as follows. Section II provides the motivation of our research. We define and formalize the research problem of feature-based software customization in section III. We present the general approach in Section IV. The evaluation and case study are reported in Section V. Discussion is presented in Section VI. We introduce the related work in Section VII and conclude in Section VIII.

II. Motivation

In this section, we elaborate the motivations of our research from both software engineering and software security perspectives. We give a formal definition of our research problem and the terms that we are going to use in the following sections.

A. Software Engineering Pragmatic Issues

Rinard [18] lists several reasons causing functionality bloat in modern software. Based on his work, we summarize some software engineering pragmatic issues related to feature creep problem. First, feature creep happens in most software development projects. When a software product becomes mature and stable, the developers still update it by adding more functions into original design. Second, software is designed and delivered as general purpose software which contains all functions required by all potential users. However, for a specific user, his or her requirements on the software are special. Only a small part of the functions of the software are useful. Other functions, to this specific user, become redundant features. Third, software reusing is also an important source of functionality bloat. The design of libraries tends to be generalized. The design of legacy projects is specialized for the purpose of legacy requirements. None of them are built for current projects and requirements. Building applications upon them inevitably brings redundant features into the software. Fourth, development "errors" also import new features. Developers are not always aware of the all effects potentially

Listing 1: A simplified example showing how the transaction integrity feature cross cuts with the moneyTransfer business logic

```java
public void moneyTransfer(int amount, User sender,
        User receiver){
  logger.info("transaction␣starts");
  //do money transfer business logic;
  logger.info("money␣was␣deducted␣from␣sender's␣balance");
  //do money transfer business logic;
  logger.info("money␣was␣added␣to␣receiver's␣balance");
  logger.info("transaction␣completes");}
```

caused by the code they are writing. Holzmann [7] calls this phenomenon "dark code" which means "the application somehow can do things nobody programmed it to do".

1) Why customizing a feature is difficult: If a property, feature, or component is well abstracted, then it is easy to be changed, extended, or removed. The challenges of features removal actually are caused by the challenges of features abstraction. Modern programming languages (e.g. OO languages), code organization (e.g. package domain, name space) and other software engineering toolkit give developers a way to model a real-world work flow and split them into smaller and smaller units. However, a system could be modeled into different abstractions and concerns. Programmers can only design the software according to the *primary abstractions*. The secondary and third important abstractions might be cross cut with the primary abstractions. Kiczales et al. [11] discuss several cross-cutting features in typical real world applications. For example, to design an online banking system, the concepts and entities that are primarily abstracted would be "balance", "account", "user", and etc. Extending and changing those entities are relatively easy because of well abstraction and encapsulation.

Feature "transaction integrity" also needs to be abstracted and implemented in an online banking system. However, it is not a primary abstraction of an online banking system. Code listing 1 shows an example. In that code snippet, feature "transaction integrity" which is enforced by logging cross cuts with money transfer and any other transaction business logic. If developers want to change or remove the implementation of money transfer business logic, they just need to change the business logic code inside method moneyTransfer. However, if developers want to enhance the "transaction integrity" by changing logging policy. They have no way to change the code in one place. They have to change all methods that the transaction integrity cross cuts with such as "moneyTransfer", "directDeposit", "checkDeposit", and many others. Similar examples could be found in the way how people implement network connection and database connection. Code listing 2 shows how network connection feature cross cuts with other business logic. Code listing 3 shows a similar example on database connection feature cross cutting with other business logic. These examples just demonstrate some common cases shared by many projects. In each specific project, it has more specialized features that are tangled with other code.

Correctly customizing a well modularized component in a program is already a challenge. From the examples above we can see that the pervasive features that cross cut with

Listing 2: A simplified example showing how the network connection feature cross cuts with the ingestContent business logic

```
public void ingestContent() throws Exception {
  URL oracle = new URL("http://www.example.com/");
  URLConnection yc = oracle.openConnection();
  BufferedReader in = new BufferedReader(
  new InputStreamReader(yc.getInputStream()));
  String inputLine;
  while ((inputLine = in.readLine()) != null){
    //do business logic;
  }
  in.close();}
```

Listing 3: A simplified example showing how the database connection feature cross cuts with the userAuthentication business logic

```
public void userAuthentication(){
  Class.forName("org.postgresql.Driver");
  Connection connection = null;
  connection = DriverManager.getConnection(
        "jdbc:oracle:thin:@localhost:1521:fakename",
                "username","password");
  //do business logic;
  connection.close();}
```

other business logic additionally increase the difficulty of customizing. That probably is one of the reasons that developers keep them there even after recognizing the negative effects that might be caused by some redundant features.

B. Security Concerns

We also have strong motivations to customize some features from the software for security reasons. Redundant features play a role in at least three security threat models.

First, malicious software vendors might threat users' privacy. There are many cases that the software companies insert backdoors to collect users' or their competitor software's behavior. If network connection feature is not used by users at all, then the users can just trim the network connection feature or at least writing-to-network feature off from the software. For example, many text editors have network connection feature. Trimming off such a feature will not affect the functions of the text editors. If the users of software hold highly sensitive data or they work in a settings with some hard constraints (e.g. military offices), they should remove those features that are not useful to users' business but with sensitive behaviors.

Second, malicious libraries provider might repackage original authenticated libraries to insert code for their own interests which threat the integrity of software that includes such a library. For example, many mobile application developers include adware library (adware here refers to the software that allows third party distributes and displays advertisements on your own apps, web pages, or software) to earn extra revenue. Some adware might secretly collect both developers' data and users' information. Besides existing scanning and malicious behaviors detecting technologies, developers still have requirements and motivation to customize third party libraries they include in their applications to achieve active defense.

Third, the software systems that lack diversity might be compromised all together at one time by outside attackers. The approach of moving target defense aims to create asymmetric uncertainty for cyber threats [10]. Feature-based software customization according to users' requirements offers a natural way to increase software diversity and achieve moving target defense.

III. Problem Definition

Before we discuss the approach to conducting feature-based program customization, we first define, formalize, and set the scope for the research problem in this section. Based on the previous analysis in last section, we have found that many features cross cut with other business logic. So feature customization cannot be done by modifying one or several methods' definition. Features are implemented as many spread and repeated method calls. So we use methods call sites as feature definition basis.

We formally define feature based on interprocedural control flow graph (ICFG) $G = (V, E)$ which regards method invocation as a special kind of control flow [17]. Besides the normal control flow edges E_n, three special kinds of control flow edges are imported to handle procedure invocation process: *call-to-return-site* edge $E_{\text{call-to-return-site}}$, *call-to-start* edge $E_{\text{call-to-start}}$, and *exit-to-return* edge $E_{\text{call-to-return}}$. *Call-to-return* edge connects the call site node and the node following the call site. *Call-to-start* edge connects the node that invokes the method and the entry node of the callee. *Exit-to-return* connects the exit node (usually return node) to the node immediate after the call site. Formally, all-edges set E is the unions of those five subsets of edges, i.e. $E = E_n \cup E_{\text{call-to-return}} \cup E_{\text{call-to-start}} \cup E_{\text{exit-to-return-site}}$.

Definition 3.1. We define *seeds* as a set of methods denoted by $\mathit{SEEDS} = \{m_1, m_2, m_3, ..., m_k\}$. Seeds could be the methods in Java standard libraries or part of the application which are specified by users or developers. Seeds usually are the methods conducting sensitive operations or other functions of interests. Simply, we call a set of methods of interest "seed methods".

Definition 3.2. We define *call sites* of a method f a set of node on this graph G, $C_m = \{c_1^m, c_2^m, \ldots, c_n^m\}$, such that $\forall c_i^m \in C^m, \exists e_i \in E_{\text{call-to-start}}$ connects c_i^m and the entry node of method m. Simply, we call the all statements that invoke a method the "call sites" of that method.

Definition 3.3. We define a *feature* a set of call sites $F = \{C_{m_1}, C_{m_2}, C_{m_2}, ..., C_{m_n}\}$ such that $\forall C_{m_i} \in F, m_i \in \mathit{SEEDS}$, Simply, a feature consists of all call sites of SEEDS.

Example. In our example program snippet shown in code listing 4, if we specify methods f and g as the seed methods, then $\mathit{SEEDS} = \{f, g\}$. The call site of method f is C_1. The call site of method g is C_2. In this case, the feature is a set consisting of two call sites: $\{C_1, C_2\}$. Taking a network feature for example, the seed set is defined to be the set of network-related APIs and the feature set is the set of call sites to these APIs. In some scenarios, we need to remove the network feature from an application. In this paper, we discuss an approach of feature removing.

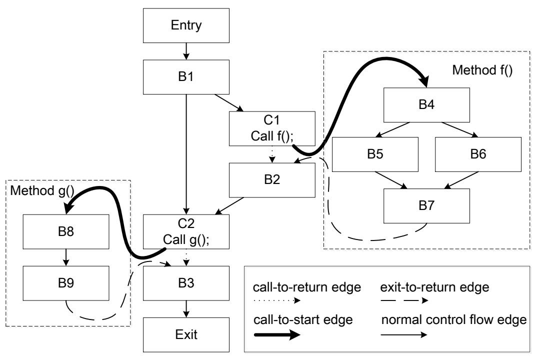


Fig. 1: Interprocedural Control Flow Graph of code listing 4

Listing 4: A code listing example

```java
public class CodeListing {
  public static void main(String[] args) {
    int argument=Integer.parseInt(args[0]);
    int ret=0;
    CodeListing instance=new CodeListing();
    if(argument<=42){
          ret=instance.f(argument);
    }
      instance.g();
  }
  public int f(int x){
    int y;
    if(x>0){
      y=1;
    }else{
      y=-1;
    }
    return y;
  }
  public void g(){
    System.out.println("Hello");
    System.out.println("World");
  }
}
```

IV. Approach

A. Overview

Based on the definition given by last section, we can clearly define our task is to remove all call sites of the seed methods *safely* and clear all the redundancy caused by this removal. To remove all methods invocation in the program, we potentially need to remove 4 parts of code in order: the code that depends on the return value or side effects (objects and array references that redefined in the callee) of the call site, call site itself, the code that is *only* depended by the parameters of the call site, and the method definition. To better demonstrate our idea, we will use a Java program example shown in code listing 5 through the whole section to show how our solution deletes those tangled code step by step. The Java code in listing 5 shows a simplified SMTP client that interacts with network. We omit the exception handling and invalid data checking in this example. This program opens a network connection after necessary preparation. It writes string *message* to the network (line 14). It reads data from the network and stores the data into array *b* via calling method *read* of class *DataInputStream* which is in JRE (line 16). We also show a simplified implementation of method *DataInputStream.read* in code listing 6 to ease the elaboration of our approach. In the

Listing 5: An example that a client reads data from and writes data to the network

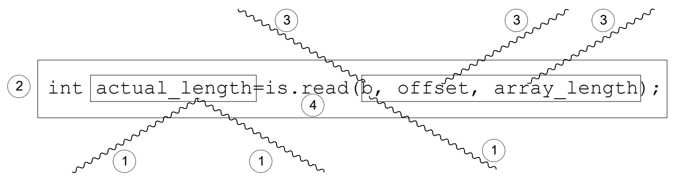
```java
1  public class SocketInAndOut{
2    public static void main(String[] args) {
3      Socket smtpSocket = null;
4      DataOutputStream os = null;
5      DataInputStream is = null;
6      String message_body=args[1];
7      String message="message_example";
8      int offset=0;
9      byte[] b=new byte[100];
10      smtpSocket = new Socket("hostname", 25);
11     os = new DataOutputStream
        (smtpSocket.getOutputStream());
12     is = new DataInputStream
        (smtpSocket.getInputStream());
13     int array_length=b.length;
14     os.writeBytes(message);
15     String responseLine;
16     int actual_length=is.read(b, offset, array_length);
17     if(actual_length<array_length){
18       System.out.println(actual_length);
19         }
20     System.out.println(b[0]);
21     System.out.println(offset);
22     os.close();
23     is.close();
24     smtpSocket.close();
25   }
31}
```

end, the program prints the actual length of the data that is read from the network, the first byte of the message, and the value of offset.

Our goal is to customize this program into a data-reading-free program which only writes data to but never receives any data from the network. The seed method in our example is DataInputStream.read(byte[] b, int off, int len). The only call site in our case is the statement in line 16 of listing 5. Fig. 2 highlights this call site and denotes 4 customization steps. In the first step, all code that depends on the return value `actual_length` and the value of the cells of array *b* which is changed in the callee would be removed. In the second step, call site itself would be removed. In the third step, we are going to remove the code on which *only* the parameters depend. In the last step, we are going to check if it is possible to delete the method definition of DataInputStream.read(byte[] b, int off, int len). It is not always the case that we need to perform all four steps every time. In our example, the deleting in step 4 will not happen because the seed method we choose is part of JRE. In other cases, we may skip step 2 or step 3 if the method does not have return value and side effects or the number of its parameters is 0. We unified all 4 steps of program customization as Program Dependency Graph (PDG) and System Dependency Graph (SDG) updating and reachability problems solving process. We need to build call graph and SDG as analysis preparation procedure. The details of each step are given by the following subsections respectively.

B. First Step: Forward Slicing

We now present the approach of forward slicing to identify the variables that rely on the return value and the side effect of the call site. We use the two-pass SDG-searching algorithm introduced by Horwitz, Reps, and Binkley [9].



```
int actual_length=is.read(b, offset, array_length);
```

Fig. 2: Delete Overview

Listing 6: A simplified java.io.DataInputStream.read implementation

```java
public final int read(byte b[], int offset, int len){
  int c = read();
  b[offset] = (byte)c;
  int i=1;
  for(;i<len;i++){
    c=read();
    if(c==-1){
      break;
    }
    b[offset+i]=(byte);
  }
  return i;
}
```

First, we need to identify which variables in the program are slicing criteria. The selected variables should be the ones that are defined or redefined in the method and caused the effects out of the scope of that method. Apparently, the return value of the method is such a variable. Besides the return value, a method may have other side effects. The side effects are caused by the change to the value pointed by array or object reference or the change to the array or object reference itself. In SDG, both explicit return value and implicit side effects data flow can be handled uniformly. We show a simplified DataInputStream.read(byte b[], int off, int len) implementation in code listing 6 which is called in line 16 of code listing 5. Method *read* has one return value and one side effect. The return value is that it returns an integer to the variable *actual_length* in line 16 of code listing 5. The side effect is caused by the change to the value of the cells of array b in code listing 6. The cells of array b is used as right value after the call site of *read*. We present a partial SDG in Fig. 3 to show how *read* method interprocedurally depends with the code in the listing 5. In Fig. 3, the nodes in grey are special nodes in SDG. They map the data flow between actual parameters and formal parameters. Because SDG captures all *interprocedural* dataflow dependency, in the figure we can see that the return value and side effect are modeled in the same way. Thus, from all actual out nodes of the call sites, a graph reachability analysis would be performed to slice out all statements that depend on the "output" of the call sites. The sliced out statements would be deleted. In our example of code listing 5, line 17 depends on *actual_length* via data flow. Line 18 depends on *actual_legnth* via control flow. Line 20 depends on the value stored in array b via data flow. However, line 21 would not be removed. Method *read* does not mutate the value of offset, so the *offset* in line 21 does not depend on the call site of *read* in line 16. In summary, line 17 to line 20 would be deleted. After this step, there are no statements depending on the seed methods.

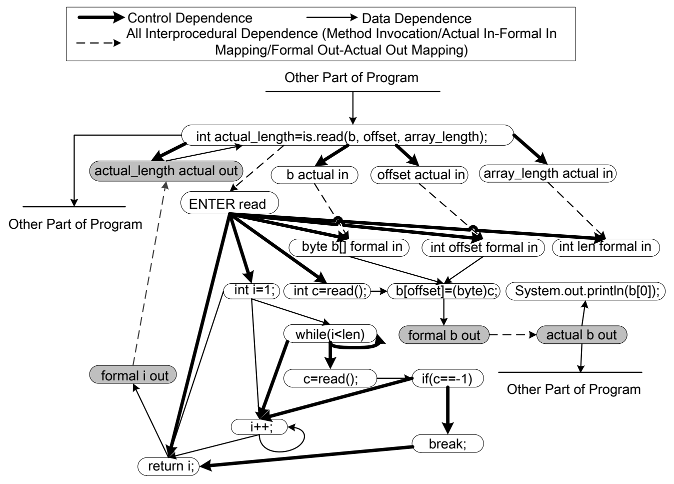
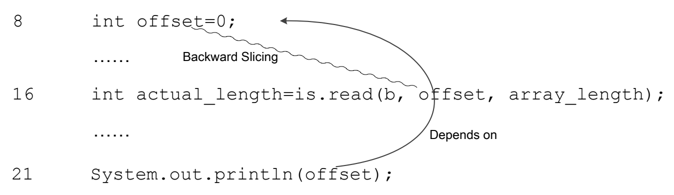


Fig. 3: Forward Slicing on Return Value and Side Effect



```
8       int offset=0;
        ......
16      int actual_length=is.read(b, offset, array_length);
        ......
21      System.out.println(offset);
```

Fig. 4: Traditional Slicing Fails to Identify the Redundancy Caused by Call Site Removal

C. Second and Third Step: Call Site Delete and Solo-slicing

In these two steps, our target is to remove the call sites and the statements on which *only* the parameters of seed methods call sites depend. Program slicing cannot help us solve this problem. According to the definition given by Weiser [28], program slicing could be denoted by a slicing criterion. Formally, it is a tuple defined as $C =< i, V >$, where i is a statement of program P and V is a subset of variables of program P. Slicing technology helps identify the statements that may affect V in i via data flow or control flow. Theoretically, the slicing result is still an executable program. However, our research question in this step could be abstracted as, which statements *only* affect V. This question could also be asked in the other way equivalently: after removal of the call sites, which statements before call sites could be removed safely. Here we do not require the sliced-out result still a runnable program. But we require the program left is still runnable.

We still use code listing 5 as our example. If we set the backward slicing criteria as the variable *offset* call site of *read* at line 15, then the statement in line 8 would be sliced out. However, we cannot delete the statement in line 8 because line 21 still depends on it. Fig. 4 highlights the relationship between these statements. This example demonstrates why slicing cannot solve this problem. Some other various versions of slicing technologies such as thin slicing [21] improved the slicing results based on an evolved slicing definition. They also cannot solve the problem we raised here.

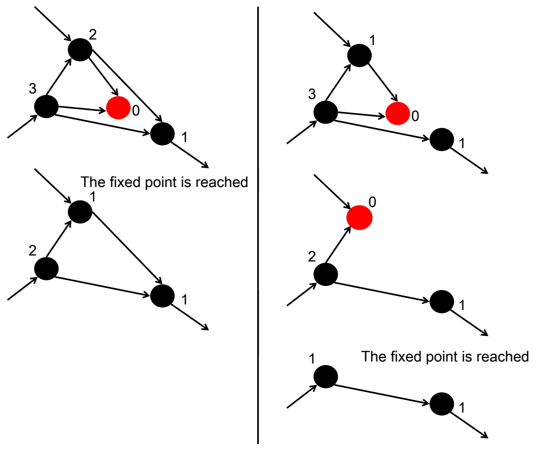


Fig. 5: Solo-slicing Algorithm Illustration

To this end, we find out that traditional slicing and its existing variation versions cannot solve the problem we encounter in this step. We define an enhanced program slicing methodology called solo-slicing to solve our problem. Like other slicing technology, we define solo-slicing and develop our algorithm based on System Dependency Graph (SDG).

1) Program Dependence Graphs and Solo-slicing: Our algorithm is based on a worklist and graph search which is shown in Algorithm 1. The main idea is to delete one vertex first from SDG and the edges pointing to it. The first step has removed the statements depending on the return value and side effects of seed method call sites. After removing call site vertex, we update the SDG and check the out degree of vertices in the updated graph. Specifically, if there are vertices whose out degree is zero, we can remove those vertices and the in-degree edges pointing to those vertices. We repeat this process until there are no vertices having zero out degree. When no vertices have zero out degree, we say the fixed point is reached and the algorithm stops. By this algorithm, we can calculate the solo-slicing result and delete the solo-sliced-out statements along side. Fig. 5 illustrates two examples of solo-slicing. The number on each vertex denotes the out degree of that vertex. The vertex that has zero out degree is in red. Two examples have the same vertices number and the different initial dependency relationship settings. The figure shows how they reach the fixed point via different number of steps.

Back to the example in code listing 5, call site of *read* in line 16 is the slicing criterion of solo-slicing which will be removed first as the trigger of the graph updating. It has three parameters, array b, integer *offset*, and integer *array_length*. Among them, array b depends on line 9 where array b is declared. Variable *array_length* depends on line 13 and line 9 via data flow. No other statements depend on line 9 and line 13. Thus these two statements are removed. Variable *offset* depends on line 8. However, variable *offset* in line 21 also depends on line 8. According to solo-slicing algorithm, line 8 would not be removed. In summary, in step 2 and 3, call site of *read* in line 16, redundancy in line 9 and line 13 are identified and removed.

Algorithm 1 Solo-slicing Algorithm

```
function SOLOSLICING($G$,$S$)
    $PDG \leftarrow G$
    $CallSiteVerticesSet \leftarrow S$
    $WorkList \leftarrow CallSiteVerticesSet$
    while $WorkList \neq \emptyset$ do
        $Vertex \leftarrow WorkList.getOneVertex()$
        $PDG.removeGivenVertex(Vertex)$
        $PDG.updateVerticesOutDegree()$
        for each vertex $v$ changes its out degree do
            if $v.OutDegree = 0$ then
                $WorkList.add(v)$
            end if
        end for
    end while
end function
```

Listing 7: After Customization

```
1  public class SocketInAndOut{
2    public static void main(String[] args) {
3      Socket smtpSocket = null;
4      DataOutputStream os = null;
5      DataInputStream is = null;
6      String message_body=args[1];
7      String message="message_example";
8      int offset=0;
10      smtpSocket = new Socket("hostname", 25);
11     os = new DataOutputStream
        (smtpSocket.getOutputStream());
12     is = new DataInputStream
        (smtpSocket.getInputStream());
14     os.writeBytes(message);
15     String responseLine;
21     System.out.println(offset);
22     os.close();
23     is.close();
24     smtpSocket.close();
25   }
31}
```

D. Fourth Step: Method Definition Delete

After all the processes above, in this step, we check if it is possible to remove the seed methods definition. If the a seed method resides in an application class or a third-party library class, then we remove it. If this method is in Java Runtime Environment (JRE), then we do not remove it.

After all four steps, the result of our customization on code listing 5 is shown in code listing 7. After customization, this code snippet is runnable, does not contain the *read* feature and does not have redundancy code caused by the feature customization.

V. Evaluation and Case Studies

A. The Pervasiveness of Cross Cutting Features in Real World Java Program

In this subsection, we present the results to the research question "how pervasive the cross cutting features are in the real world Java program". We select three features. They are network connection, database connection, and logging.

We conduct experiments on DaCapo 9.12-bach benchmarks, which contain 10 programs. These 10 benchmarks are typical desktop standalone applications that are designed by following the principle of "small is beautiful". For most of them, network connection and data persistence are not their proposed functions. If one benchmark's main function happens to be network connection and data persistence, we will skip that application.

1) Presence of Network Connection Call Sites: The number of network connection call sites of each benchmark is shown in the first row of Table I. Among the benchmarks, tomcat is a web server whose main business logic includes network connection. In this case, we do not consider network connection feature as a cross cutting feature for benchmark tomcat. So we do not calculate this number for the benchmark tomcat. From the table, we can see that 7 out of 9 benchmarks have cross cutting network connection API call sites. The benchmark batik has the highest call sites number 83. The benchmark avrora and sunflow do not network connection call sites. On average, each benchmark has 15 network connection call sites.

2) Presence of Database Connection: The number of database connection call sites of each benchmark is shown in the second row of Table I. Among the benchmarks, h2 itself is a database. So we do not calculate database connection call sites number for benchmark h2. From the table, we can see that 5 out of 9 benchmarks have cross cutting database connection API call sites. The benchmark tomcat has the highest call sites number 8. On average, each benchmark has 2 database connection call sites.

3) Presence of Logging: The number of logging call sites of each benchmark is shown in the third row of Table I. From the table, we can see that 5 of out of 10 benchmarks have cross cutting logging API call sites. The benchmark lucene has the highest call sites number 962. On average, each benchmark has 125 logging call sites.

The data above provides the evidence that the features that have nothing to do with applications' main business logic are pervasive in real world Java applications. This fact indicates that, first, it is feasible to remove undesired features; second, it is important to remove those undesired features.

B. Case Studies

In this subsection, we evaluate the correctness of our approach. Further impacts (e.g. performance and security) of feature removal are out of the scope of current stage, which is our future work. The fact that if a feature is correctly removed or not is hard to be measured quantitatively, therefore we choose case studies as our evaluation methodology. Each customized application is validated by selected tests running and manual inspecting.

1) DrJava: Network Connection: DrJava is a lightweight Java programming environment for pedagogic purpose [1]. The core functionality of DrJava has nothing to do with network connection. However, it does have network connections in its code for checking updates. DrJava has 687 classes. The total number of lines of code are 163,566. We conduct a case study on removing network based feature from Dr.Java. Network related features are defined by methods *openConnection* and *openStream* in class *java.net.URL*. Specifically, if the developers want to have network connection in their program, they must call those APIs. Thus, in this case, *SEEDS* consists of methods *openConnection* and *openStream*. Table II shows the specific call sites of the seed methods. We use them as seed methods to conduct feature-based customization based on the approach we proposed.

Listing 8: DrJava openConnection Callsite Forward Slicing Results

```
321  URLConnection uc = fileURL.openConnection();
322  final int length = uc.getContentLength();
323  InputStream in = uc.getInputStream();
324  ProgressMonitorInputStream pin =
       new ProgressMonitorInputStream
      (_mainFrame, "Downloading␣"+fileName+"␣...", in);
325  ProgressMonitor pm = pin.getProgressMonitor();
326  pm.setMaximum(length);
327  pm.setMillisToDecideToPopup(0);
328  pm.setMillisToPopup(0);
330  { public void run() { closeAction(); } });
331  BufferedInputStream bin = new BufferedInputStream(pin);
334  edu.rice.cs.plt.io.IOUtil.copyInputStream(bin,bout);
335  bin.close();
337  if ((!destFile.exists())
       || (destFile.length() != length)) {
338    abortUpdate("Could␣not␣download␣update."); return;
339  }
```

Among these call sites, we use method *updateAction* in class *NewVersionPopup* as an example. Code listing 8 shows the forward slicing results of *openConnection* inside method *updateAction*. The call site is at line 321. The return value of this call site is *uc* whose type is *URLConnection*. This call site does not have side effect. So the only slicing criteria is *uc* in the statement of line 321. The statements shown in code listing 8 would be deleted. The method *openConnection* does not have parameters. So the backward solo-slicing would not be performed. In the end, we find out that the method *openConnection* is in JRE. So the method definition will not be removed. After customization, we test Dr.Java by the test cases designed by us. First, DrJava could start up successfully after feature customization. Second, DrJava cannot access Internet to check and download update. Third, rest of functions could work normally.

2) Hadoop: Database Connection: Apache Hadoop is an open source software for scalable distributive computing. In this case study, we want to remove the database connection feature from Apache Hadoop project. Database connection related features are defined by method *getConnection* in class *java.sql.DriverManager*. Specifically, to connect with database, developers must write a sequence of routine code to perform a series of operations which starts with *DriverManager.getConnection*. So in this case, *SEEDS* contains only one method *getConnection*. It has two call sites. Both of them are in method *getConnection* of class *DBConfiguration*. Code listing 9 shows two call sites of our seed method in *DBConfiguration.getConnection*. It is notable and interesting that *DBConfiguration.getConnection* actually is a wrapper method of Java standard API *DriverManager.getConnection*. They even have the same method name. *DBConfiguration.getConnection* directly uses the return value of seed method as its own return value (line 151 and line 153). This fact causes that all call sites of *DBConfiguration.getConnection* are also removed from the program in the forward slicing stage.

TABLE I: Network, Database, and Logging Features

Benchmarks	avrora	batik	fop	h2	jython	lucene	pmd	sunflow	tomcat	xalan
Number of Network Feature Call Sites	0	83	28	1	1	8	3	0	N/A	9
Number of Database Feature Call Sites	0	0	0	N/A	2	6	3	0	8	1
Number of Logging Feature Call Sites	0	0	87	1	5	962	0	0	195	0

TABLE II: Call Sites of method *openConnection* and *openStream* in DrJava

SEEDS: java.net.URL.openConnection/openStream
edu.rice.cs.drjava.ui.NewVersionPopup$6.updateAction
edu.rice.cs.drjava.ui.NewVersionPopup.getManualDownloadURL
edu.rice.cs.drjava.ui.MainFrame._generateJavaAPISet
edu.rice.cs.drjava.ui.NewVersionPopup.getBuildTime

Listing 9: The Code of Call Sites of *DriverManager.getConnection*

```
150  if(conf.get(DBConfiguration.USERNAME_PROPERTY) == null) {
151    return DriverManager.getConnection(
         conf.get(DBConfiguration.URL_PROPERTY));
152  } else {
153    return DriverManager.getConnection(
           conf.get(DBConfiguration.URL_PROPERTY),
           conf.get(DBConfiguration.USERNAME_PROPERTY),
           conf.get(DBConfiguration.PASSWORD_PROPERTY));
155  }
```

The backward solo-slicing starts from the parameters of *DriverManger.getConnection*. We use the call site at line 151 as an example. That seed method call site uses the *anonymous* return value of *conf.get(DBConfiguration.URL_PROPERTY))* as its parameter. Apparently, an anonymous return value is impossible to be used somewhere else but its call site. So the call site *conf.get* is also removed. After this removal, the number of statements that depend on constant value *DBConfiguration.URL_PROPERTY* is one less. But the SDG shows that its out degree is still greater than zero at this moment. So the solo-slicing stops here and the static constant field *DBConfiguration.URL_PROPERTY* will not be removed. The backward solo-slicing from parameters of call site in line 153 follows the same manner. But the results are different. The static constant fields *DBConfiguration.USERNAME_PROPERTY* and *DBConfiguration.PASSWORD_PROPERTY* are removed in the end because line 153 is the *only* statements that depend on these two fields. At last step, we check if it is possible to delete the method definition. Our seed method is part of JRE. So the seed method definition will not be deleted. But its wrapper method *DBConfiguration.getConnection*, whose call sites are removed as well, is in application space. So the method definition of *DBConfiguration.getConnection* will be deleted.

After customization, we test *Hadoop-mapreduce-client* by the test cases designed by us. We find out that the database connection is disabled. The project's rest of functions could work normally.

3) Maven: Logging: Apache Maven is a software project management tool written in Java which can facilitate building automation, documents generation, and dependency resolving. In this case study, we want to remove debugging information logging feature from Maven. In Java, there are multiple logging frameworks. But their design are quite similar. By calling different methods of logger, logger can log the information and label these information with different importance-level tags. Thus administrators can handle or retrieve these logs according to their importance levels for different purposes. Maven uses the logging framework from *Plexus* project. In this framework, the logging importance levels, from least important to most important, are ranked as *LEVEL_DEBUG, LEVEL_INFO, LEVEL_WARN, LEVEL_ERROR,* and *LEVEL_FATAL*. To log debug information, developers need to call *Logger.debug* method. So *SEEDS* in this case study, contains one method *org.codehaus.plexus.logging.Logger.debug*. By removing this seed method, we can remove debug information logging feature.

The seed method has 60 call sites in 17 classes as shown in Table III. Some package names are omitted due to its length exceeding the page limit. We use one of these call sites in class *DefaultProjectDependenciesResolver* as an example. The relevant code is highlighted in listing 10. The code from line 253 to line 277 is omitted due to the page limit. This code listing displays a quite typical scenario that logging feature cross cuts with other business logic. The business logic of method *visitEnter* is about node dependency resolving. Logging code is interwoven with the main business logic of the method here. The call site of *Logger.debug* is at line 283. It does not have return value. So we do not need to perform forward slicing. It takes *buffer.toString()* as its parameter. The backward solo-slicing will start from *buffer.toString()*. After removing call site, *buffer* in line 281 will lose its only dependent. Thus line 281 would be deleted which causes *buffer* in line 280 lose its only dependent. Such a solo-slicing chain will go all the way back to line 250. The solo-slicing will not stop until line 250 is deleted. In other 59 call sites, the operation and removing process is similar. After we remove all call sites of the seed method. We can remove the seed method definition because this seed method is in a third party library.

To this end, we have removed the debugging information feature from Apache Maven project. We test Maven by the test cases designed by us. We find that Maven cannot do debugging information logging any more. Other functions of Maven work normally.

VI. Discussion

A. Solo-slicing

To solve the problem of redundancy removal, we propose a new slicing concept and technique, solo-slicing. Solo-slicing could slice out the statements that are *only* affected by or *only* affect the slicing criteria. As a side product of our research on feature-based software customization, solo-slicing potentially could be applied to many other research areas independently. Solo-slicing might improve the slicing efficiency on software debugging, transformation, and binary difference comparison

TABLE III: *Logger.log* Call Sites in Apache Maven project

SEEDS: java.util.logging.Logger.log	
The Classes that have SEEDS call sites	**Call Sites Number**
org.apache.maven.bridge.MavenRepositorySystem	1
org.apache.maven.classrealm.DefaultClassRealmManager	8
org.apache.maven.DefaultMaven	3
org.apache.maven.lifecycle.internal.LifecycleDebugLogger	20
org.apache.maven.· · · .DefaultLifecyclePluginAnalyzer	1
org.apache.maven.lifecycle.internal.MojoDescriptorCreator	1
org.apache.maven.· · · .MultiThreadedBuilder	2
org.apache.lifecycle.DefaultLifecycles	1
org.apache.maven.LoggingRepositoryListener	2
org.apache.maven.plugin.internal.DefaultMavenPluginManager	5
org.apache.maven.· · · .DefaultPluginPrefixResolver	3
org.apache.maven.· · · .DefaultPluginVersionResolver	5
org.apache.maven.plugin.DebugConfigurationListener	2
org.apache.maven.project.DefaultProjectBuildingHelper	2
org.apache.maven.project.artifact.MavenMetadataSource	1
org.apache.maven.project.DefaultProjectDependenciesResolver	1
org.apache.maven.toolchain.DefaultToolchainsBuilder	1

Listing 10: The Code Around One Call Site of Seed Method in Apache Maven Project

```
public boolean visitEnter( DependencyNode node ){
  StringBuilder buffer = new StringBuilder( 128 );
  buffer.append( indent );
  org.eclipse.aether.graph.Dependency dep
    = node.getDependency();
  if ( dep != null ){
    ...//omit due to page limit.
  }else{
    buffer.append( project.getGroupId() );
    buffer.append(':').append(project.getArtifactId());
    buffer.append(':').append(project.getPackaging());
    buffer.append(':').append(project.getVersion());
  }
  logger.debug( buffer.toString() );
  indent += "␣␣␣";
  return true;
}
```

tasks in certain scenarios. The possible impact there is worth further investigation.

B. Limitations and Future Work

This paper focuses on a novel research question, the formalization and analysis of this research question, and potential techniques for solving the problem. We evaluate our approach from several perspectives. However, we have not done large-scale experiments yet in this preliminary feasibility study. In the future, we will conduct more empirical studies to evaluate our approach in a more comprehensive way.

VII. Related Work

A. Bloatware

With the fast development of modern software engineering, "bloatware" problem is gaining more and more attention from both academia and industry. From static perspective, bloatware increases the cost of dependency management, building, changing, and storing. Morgenthaler et al. try to lower the difficulty of dependency management and target building caused by huge monolithic code base [14]. By removing the build files associated with dead code, identifying "unbuildable targets" and unnecessary command line flags, developers could ease the process of target building and pay down so-called "technical debt". They try to mitigate the problem without changing the code base. Wang et al. follow the similar approach and additionally include code base itself into consideration [26]. They implement a tool to find out intra- and inter-module dependencies on both symbol level and module level. Developers can use those information to conduct large-scale refactoring on their huge code base. Vakilian et al. propose an approach to decomposing large build targets into smaller ones to avoid frequently triggering build and test tasks [25]. Ryder and Tip propose change impact analysis to precisely identify affected regression testing cases due to the change to the large code base [19]. Holzmann gives an overview about the problems of bloatware and code inflation [7]. From the performance perspective, Xu [29] proposes a strategy of reusing those redundant objects by utilizing a tool called Coco, which can replace ineffective Java collections into effective ones to get rid of bloat from Java software soundly and adaptively.

B. Program Slicing

There has been a substantial amount of research on program slicing. Mark Weiser first raises the idea of program slicing, which could be applied to regression testing, program parallelization and automatic debugging [28]. Along with this idea, he also presents a static program-slicing algorithm. However, this algorithm could only be applied to a program that has a monolithic procedure. Arvind and Shankar presented a methodology to use program-slicing technology to facilitate regression testing [2]. Specifically, their methodology is based on alias analysis and an interprocedural program slicing algorithm proposed by Horwitz et al. [8]. Ottenstein and Ottenstein developed the program dependence graph (PDG) [16]. This data structure provides an infrastructure to develop a new more effeicent program slicing algorightm. But PDG only facilitates intraprocedural program slicing analysis. Horwitz et al. solved the problem of interprocedural program slicing. They introduce a new form, system dependence graph (SDG), to represent the program [8]. The challenge of conducting interprocedural slicing is analyzing calling context of procedures. Compared with PDG, this approach overcomes the difficulty by importing transitive dependences relationship. Some other researchers make efforts on dynamic slicing techniques. Wang and Roychoudhury implement a Java dynamic tool called JSlice [27]. The strength of this tool is that huge bytecode traces could be represented in an very effective manner. Hammacher implements a tool called JavaSlicer [6], which is easy to set up and use. To make this tool Java virtual machine implementation independent, the author takes advantage of Java agent technology. Treffer and Uflacker also implement Java dynamic slicing on soot framework [24].

C. Software Diversity

Software diversity enhances the software security from multiple aspects. Software diversity offers a probabilistic protection mechanism [12]. Additionally, it is a kind of active defense which can defend a wide range of types of attack, including unknown attack methods. Software could be diversified in different levels by different approaches. It offers a large design space to software diversity researchers. Our research offers one way to diversify software via feature-based customization. Other research in this area diversifies software by different granularities. Snow et al. diversify the software on instruction level [20]. Their approaches include equivalent instructions and equivalent instruction sequences substitution. Some other works diversify the software on basic block level [5], [13]. Their technologies include opaque predicate insertion and branch function insertion. On the program level, approach instruction set randomization [3] and virtualization-based obfuscation are proposed. They are efficient on defending code injection attacks.

VIII. Conclusion

In this paper, we discuss and formally define a novel research problem of feature-based software customization. Based on that, we present a multistep static program slicing based approach to conducting feature-based software customization. Additionally, as a part of of the multistep approach, we propose a new concept, solo-slicing, which can slice out the statements that are *only* affected by or *only* affect the slicing criteria. Our approach can help remove a feature from the software safely and clean all redundancy caused by this removal, and can potentially help legacy code retrofitting and maintenance.

IX. Acknowledgments

This research was supported in part by the Office of Naval Research (ONR) grants N00014-13-1-0175 and N00014-16-1-2265.

References

[1] E. Allen, R. Cartwright, and B. Stoler, "Drjava: A lightweight pedagogic environment for java," in *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '02. ACM, 2002, pp. 137–141.

[2] D. Arvind and P. Shankar, "Slicing of Java programs using the Soot framework," Master's thesis, Indian Institute of Science, 2006.

[3] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović, "Randomized instruction set emulation," *ACM Transactions on Information and System Security (TISSEC)*, vol. 8, no. 1, pp. 3–40, 2005.

[4] Y. Bu, V. Borkar, G. Xu, and M. J. Carey, "A bloat-aware design for big data applications," in *Proceedings of the 2013 International Symposium on Memory Management*, ser. ISMM '13. ACM, 2013.

[5] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '98. ACM, 1998, pp. 184–196.

[6] C. Hammacher, "Design and implementation of an efficient dynamic slicer for Java," Bachelor's Thesis, Saarland University, 2008.

[7] G. J. Holzmann, "Code inflation," *Software, IEEE*, vol. 32, no. 2, 2015.

[8] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, ser. PLDI '88. New York, NY, USA: ACM, 1988, pp. 35–46.

[9] ——, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 1, pp. 26–60, 1990.

[10] S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, Eds., *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*, ser. Advances in Information Security. Springer, 2011, vol. 54.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *European Conference on Object-Oriented Programming (ECOOP '97)*, 1997.

[12] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 276–291.

[13] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS '03. ACM, 2003, pp. 290–299.

[14] J. D. Morgenthaler, M. Gridnev, R. Sauciuc, and S. Bhansali, "Searching for build debt: Experiences managing technical debt at Google," in *Proceedings. of the Third International Workshop on Managing Technical Debt*, 2012.

[15] K. Nguyen and G. Xu, "Cachetor: Detecting cacheable data to remove bloat," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (FSE'13)*. ACM, 2013, pp. 268–278.

[16] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," in *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 1984.

[17] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. ACM, 1995, pp. 49–61.

[18] M. Rinard, "Manipulating program functionality to eliminate security vulnerabilities," in *Moving Target Defense*. Springer, 2011.

[19] B. G. Ryder and F. Tip, "Change impact analysis for object-oriented programs," in *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2001.

[20] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 574–588.

[21] M. Sridharan, S. J. Fink, and R. Bodik, "Thin slicing," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. ACM, 2007, pp. 112–122.

[22] Standish Group, "CHAOS report 2009," Tech. Rep., 2009.

[23] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter, "Practical experience with an application extractor for Java," in *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '99. ACM, 1999, pp. 292–305.

[24] A. Treffer and M. Uflacker, "Dynamic slicing with Soot," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis (SOAP '14)*, 2014.

[25] M. Vakilian, R. Sauciuc, J. D. Morgenthaler, and V. Mirrokni, "Automated decomposition of build targets," in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE '15. IEEE Press, 2015, pp. 123–133.

[26] P. Wang, J. Yang, L. Tan, R. Kroeger, and J. D. Morgenthaler, "Generating precise dependencies for large software," in *Proceedings of the Forth International Workshop on Managing Technical Debt*, 2013.

[27] T. Wang and A. Roychoudhury, "Dynamic slicing on Java bytecode traces," *ACM Transaction on Programming Languages and Systems (TOPLAS)*, vol. 30, no. 2, Mar. 2008.

[28] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE '81. IEEE Press, 1981, pp. 439–449.

[29] G. Xu, "Finding reusable data structures," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12. ACM, 2012, pp. 1017–1034.

[30] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky, "Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER '10)*. ACM, 2010, pp. 421–426.