一种基于动态插桩的JavaScript反事实执行方法

龚伟刚 游 伟 李 赞 石文昌 梁 彬
(数据工程与知识工程教育部重点实验室(中国人民大学) 北京 100872)
(中国人民大学信息学院 北京 100872)

摘 要 目前,静态分析技术已被广泛用于JavaScript程序的安全性分析。但是由于JavaScript支持通过eval等方法在运行时动态生成代码,仅靠静态分析难以取得动态生成代码。一种可行的解决方法是通过动态运行目标程序取得动态生成代码,再对其进行静态分析。然而,动态运行目标程序只能覆盖有限的执行路径,会遗漏其他执行路径中的动态生成代码。针对这一问题,基于动态插桩实现了一个反事实执行方法。该方法通过修改JavaScript引擎,在其语法解析阶段动态插入反事实执行体,使条件不成立的分支路径和当前执行路径均能够得到执行。通过该插桩方式,即使嵌套调用eval等方法,也能在其动态生成代码中完成插桩。同时,还实现了一种按需undo方法,以消除反事实执行体中赋值操作带来的影响,且能够避免冗余操作。实验结果表明,实现的方法能够有效地扩大动态分析中执行路径的覆盖面。

关键词 反事实执行,路径覆盖,动态分析,JavaScript
中图法分类号 TP309.2 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.11.004

JavaScript Counterfactual Execution Method Based on Dynamic Instrumentation

GONG Wei-gang YOU Wei LI Zan SHI Wen-chang LIANG Bin
(Key Laboratory of Data Engineering and Knowledge Engineering of Ministry of Education (Renmin University of China), Beijing 100872, China)
(School of Information, Renmin University of China, Beijing 100872, China)

Abstract The static analysis technique has been widely employed in the security analysis of JavaScript program. But the JavaScript program can leverage several functions such as eval to generate code at runtime, which is hard to obtain dana-mic generation code simply by static analysis. One feasible approach is to collect the code by running the target program dynamically and then make a static analysis on it. However, this approach can only explore a finite number of execution paths and will miss the dynamically generated code in other paths. This paper presented a counterfactual execution method based on dyna-mic instrumentation. In the method, the counterfactual execution structures are instrumented on-the-fly during the parse phase of JavaScript engine, to explore both the branch that would ordinarily be executed and the other branch that would not normally be run. In this way, even if the functions like eval are called nestedly, the dynami-cally generated code can also be instrumented. Besides, in order to undo the effect of any assignment in counterfactual execution structures, an on-demand undo method was implemented to avoid the redundant operations. The evaluation re-sults show that the method implemented in this paper can effectively expand the coverage of execution paths in dynamic analysis.

Keywords Counterfactual execution, Path coverage, Dynamic analysis, JavaScript

1 引言

静态分析作为一种重要的程序分析方法,常被用于对JavaScript程序进行安全性分析。如在Guarnieri及Livshits等人的研究工作[1-2]中,通过对JavaScript代码进行静态分析,构造函数调用图,以检测其中存在的安全漏洞。在Guha等人的工作[3]中,通过静态分析对客户端JavaScript程序中的行为进行建模,以防范攻击行为。但是,JavaScript作为一种高阶编程语言,具有高动态性的特点。实际中,JavaScript支持通过eval,document.write等方法在运行时动态生成代

到稿日期:2016-10-25 返修日期:2016-12-30 本文受国家自然科学基金(61170240,91418206,61472429),国家科技重大专项(2012ZX01039-004)资助。

龚伟刚(1992-),男,硕士生,主要研究方向为软件安全;游 伟(1988-),男,博士,主要研究方向为Android安全;李 赞(1993-),女,硕士生,主要研究方向为软件安全;石文昌(1964-),男,教授,博士生导师,主要研究方向为可信计算、数字取证等;梁 彬(1973-),男,教授,博士生导师,主要研究方向为软件安全性检测、系统软件安全机制等,E-mail:liangb@ruc.edu.cn(通信作者)。

码进行执行，而且程序编写者还能够利用 eval 等代码生成方法对 JavaScript 代码进行混淆处理[4]。Ratanaworabhan 及 Richards 等人的研究工作[5-7]表明，仅依靠静态分析难以应对 JavaScript 的动态特性，从而无法获得其动态生成代码并进行分析。图 1 给出了一个动态生成代码示例，其中当 a 等于 0 时，进入 if 语句的 else 分支。该分支中的 document[v]即为 document. write 方法，其参数 temp 通过字符串拼接而成，其内容为“〈script〉eval(‘location. replace(“http: // www. evil. com”);’);〈/script〉”。该示例通过嵌套调用动态代码生成方法，实现向恶意网页的自动跳转。面对这种复杂的动态代码生成逻辑，静态分析方法无法取得其中的动态生成代码。

```
if(a !=0) {
  ……
} else {
  v="ri"+"te";
  v="w". concat(v);
  var str=["60","115","99","114","105","112","116","62","101",
           "118","97","108","40","39","108","111","99","97",
           "116","105","111","110","46","114","101","112","108",
           "97","99","101","40","34","104","116","116","112",
           "58","47","47","119","119","119","46","101","118",
           "105","108","46","99","111","109","34","41","59","39",
           "41","59","60","47","115","99","114","105","112",
           "116","62"];
  var temp='';
  var gg='';
  for (i=0; i < str. length; i ++) {
    gg=str[i];
    temp=temp+String. fromCharCode(gg);
  }
  document[v](temp);
}
```

图 1　动态生成代码示例

为了解决动态生成代码给静态分析带来的问题，可以在静态分析前先动态运行目标分析程序，取得运行过程中的动态生成代码，然后再对其进行静态分析。如，Wei 及 Chugh 等人[8-10]均采用动态分析和静态分析相结合的方法对 JavaScript程序进行分析。但是动态分析在一次执行过程中只能覆盖当前执行路径，而程序中一般存在多条执行路径，其他执行路径中也可能存在动态生成代码。如图 1 中的动态生成代码即位于 if 语句的 else 分支中，若当前执行路径下 a 不等于 0，则无法得到该动态生成代码。

为解决该问题，Schäfer 和 Sridharan 等人采用静态插桩的方法，针对 JavaScript 中的 if 语句，提出了一种可行的反事实执行方法[11]。通过该方法，使动态分析能够在一次执行过程中覆盖 if 语句中的多条分支路径，有效扩大了测试路径的覆盖面。但是，该方法仍有以下 3 点不足：1）该工作采用静态插桩方法，在 JavaScript 源代码层面识别 if 语句并插入反事实执行体。但如图 1 所示，JavaScript 支持嵌套调用 eval 等动态代码生成方法，静态插桩的方法无法对动态生成代码中的 if 语句进行插桩。2）由于反事实执行体中的语句在正常情况下并不会被执行，因此为了消除其中的赋值操作对程序后续正常操作的影响，该工作通过静态插桩，在退出反事实执行体时，将其中所有经过赋值操作的元素都恢复成操作前的原始值（undo）。但实际上，如果元素在程序后续操作中未被再次使用，则没有必要对其进行 undo 操作。该工作的 undo 方法中存在冗余操作，会影响效率。3）该工作只考虑了 if 语句中的分支路径，未实现对 switch 语句的反事实执行方法，而与 if 语句相比，switch 语句往往存在更多的分支路径。

基于以上背景，本文通过修改 JavaScript 引擎，基于动态插桩，提出了一种 JavaScript 反事实执行方法。与 Schäfer 等人的工作[11]相比，该方法采用了动态插桩方式，在 JavaScript引擎进行语法解析生成抽象语法树的过程中，动态地插入反事实执行体（On-the-Fly）。由于 JavaScript 代码在执行前必须先经过语法分析，因此基于此方法能够覆盖所有 JavaScript 执行代码，包括嵌套生成的动态生成代码。在实现 undo 方法时，本文通过在 JavaScript 引擎的编译阶段进行动态插桩，实现了一种特殊的按需 undo 方法，称为 Undo_on_Demand。该方法只在对相应元素作取值操作时才恢复其原始值，避免了 Schäfer 等人工作中的冗余操作。同时，本文针对 JavaScript 中的 if 语句和 switch 语句，分别设计了相应的反事实执行体结构，能够覆盖程序中更多的分支路径。最后，在谷歌的 V8 JavaScript 引擎[12]中实现了本文的反事实执行方法，并利用该方法对基于 PhoneGap 框架[13]开发的 HTML5 新型安卓应用进行分析，验证了其能够有效扩大动态分析过程中测试路径的覆盖面。

本文第 2 节对反事实执行的设计方法进行描述；第 3 节介绍如何通过动态插桩实现本文的反事实执行方法；第 4 节以具体实例对实验结果进行说明；最后总结全文。

2　反事实执行系统的设计

反事实执行系统的基本架构如图 2 所示。

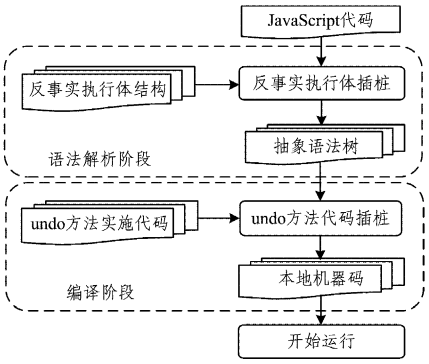

图 2　反事实执行架构图

在设计过程中主要面临两个方面的问题，分别是如何设计满足需求的反事实执行体结构，以及如何在 V8 JavaScript 引擎中设计按需的 undo 方法。下面分别对这两个设计过程进行描述。

2.1　反事实执行体结构

本文通过设计特殊的反事实执行体结构，并在 V8 引擎的语法解析阶段以抽象语法树（Abstract Syntax Tree，AST）节点的形式将其动态插入，使在对 JavaScript 程序进行动态分析时，能够在一次执行过程中遍历多条分支路径。反事实执行体主要由经过特殊构造的 if 语句构成，为了使原 if 和

switch 语句中所有分支的代码均能得到执行，将反事实执行体的判断条件设定为原语句中分支条件的相反值，再将原语句中的代码复制到反事实执行体中。同时，在进入反事实执行体前，需要先对原 if 语句和 switch 语句条件的确定性进行判断，若其是一个确定值，即在实际执行过程中总是只有一个分支被执行，则不必进入反事实执行体执行其他分支中的代码。本文通过使用已有的 JavaScript 动态污点分析工具，在 JavaScript 程序运行过程中对元素的确定性进行判定。下面分别对 if 语句和 switch 语句的反事实执行体结构进行描述。

2.1.1 if 反事实执行体结构

if 语句的反事实执行体结构以 JavaScript 伪代码的形式在图 3 中给出，并用粗斜体标识。第 1－9 行是反事实执行体结构，第 10－16 行则是原始的 if 语句。原始 if 语句中包含两条分支路径，当 condition 为 true 时，执行 then 分支中的语句 statement1，否则执行 else 分支中的语句 statement2。在反事实执行体中，先调用事先添加到 V8 引擎中的方法 is_Undeterminate()对 condition 的确定性进行判断，若是不确定的则继续执行。由于在反事实执行体中将条件取反为!condition，因此与原始 if 语句正好相反，当 condition 为 true 时，执行 else 分支中的语句 statement2，否则执行 then 分支中的语句 statement1。通过插入反事实执行体，最终实现了无论 condition 取值如何，if 语句两条分支路径中的代码都能够得到执行，有效扩大了动态分析过程对 if 语句中分支路径的覆盖面。

```
1. if (is_Undeterminate (condition)) {
2.   if (!condition) {
3.     statement 1;
4.     ……
5.   } else {
6.     statement 2;
7.     ……
8.   }
9. }
10. if(condition) {
11.   statement1;
12.   ……
13. } else {
14.   statement2;
15.   ……
16. }
```

图 3 if 反事实执行体结构

在设计反事实执行体时，为了与原程序的执行语义保持一致，必须保证在执行完反事实执行体后能够继续执行后续的原始 if 语句。考虑到 JavaScript 编程语言中只有 break，continue 和 return 语句会导致函数中的后续操作不被执行，因此在将 if 中的语句复制到反事实执行体前，需对其中的这 3 类语句进行特殊处理(在 switch 反事实执行体中需要做相同操作，不再重复说明)。针对 break 和 continue 语句，若其在反事实执行体中单独出现，即其作用的循环结构在反事实执行体外，则将其删除。对于 return 语句，由于其返回的表达式中可能包含目标分析代码，因此不能简单地将 return 语句直接删除，而需要保留其中的表达式部分。本文将 return 语句从“return expression;”形式转换成“expression;”形式，由此保证在反事实执行体中函数不会直接返回，可以继续执行后续的原始 if 语句。

2.1.2 switch 反事实执行体结构

与 if 语句相比，由于 switch 语句中一般存在多个 case，而每一个 case 均对应一条分支执行路径，因此为了能够遍历所有 case 分支路径中的代码，switch 语句的反事实执行体结构比 if 语句更加复杂。图 4 以 JavaScript 伪代码形式给出了 switch 语句的反事实执行体结构，其中以一个含有两个 case 分支和一个 default 分支的原始 switch 语句进行说明。

```
1. if (is_Undeterminate (tag)) {
2.   if (tag == condition1) {
3.     statement 2;
4.     statement 3;
5.     ……
6.   }
7.   if (tag == condition2) {
8.     statement 1;
9.     statement 3;
10.    ……
11.  }
12.  if (tag != condition1 && tag != condition2){
13.    statement 1;
14.    statement 2;
15.    ……
16.  }
17. }
18. switch(tag) {
19.   case condition1:
20.     statement1;
21.     ……
22.     break;
23.   case condition2:
24.     statement2;
25.     ……
26.     break;
27.   default:
28.     statement3;
29.     ……
30. }
```

图 4 switch 反事实执行体结构

在图 4 中，switch 语句的反事实执行体结构用粗斜体标识。第 1－17 行是反事实执行体结构，第 18－30 行则是原始的 switch 语句。原始 switch 语句中包含 3 条分支，依次为 case condition1，case condition2 和 default，每个分支中均包含若干条 JavaScript 语句。在 switch 反事实执行体中，先对 tag 标签的确定性进行判断，当其是不确定时才继续执行，然后针对 3 条分支分别构造了一条 if 语句，并将另两条分支中的 JavaScript 语句复制到该 if 语句中。例如，若当前 tag 等于 condition1，则反事实执行体 condition1 对应的 if 语句中包含有其余分支中的语句 statement2 和 statement3。最终，实现在插入反事实执行体后，无论 tag 属于哪一种 case，所有分支中的语句均能够得到执行，有效扩大了动态分析过程对 switch 语句中分支路径的覆盖面。

2.2 按需 undo 方法

为了实现本文的按需 undo 方法，即在对元素做取值操作

时恢复其在反事实执行体中做赋值操作之前的原始值，需要对 JavaScript 程序中元素的写操作进行插桩，若其发生在反事实执行体中，则在将新的值写入元素之前，需先记录下该元素的原始值以待恢复。同时，还需要对 JavaScript 程序中元素的读操作进行插桩，若该元素在反事实执行体中经过赋值操作，且当前读操作已离开反事实执行体，则需返回之前记录下的原始值。

在按需 undo 方法的设计过程中，一个关键问题是如何对程序元素的原始值进行存储，并在 undo 时取回指定元素的原始值。JavaScript 是一种高动态性的编程语言，其变量类型在运行过程中可能发生改变，同时还支持动态添加或删除变量的属性；而且，V8 在运行过程中可能改变 JavaScript 对象的存储模式，使对象属性的存储位置发生变化。本文参考现有的相关工作[14-15]，采用类似方法，通过包裹对象实现对原始值的存储。该方法针对在反事实执行体中经过赋值操作的元素，将元素当前的值与它的原始值一起存储在一个新的包裹对象中，然后把该元素的值替换成这个包裹对象。为了与该方案配合，需要修改 JavaScript 操作的执行逻辑。在读取变量的值进行操作前，需要对其类型进行判定，如果是包裹对象，则需根据当前是否在反事实执行体中，决定返回包裹对象中的当前值还是原始值，然后再进行后续操作。

本文设计的包裹对象是在 V8 中添加的一个新的 JavaScript对象 BoxObj。图 5 以高层次的 JavaScript 伪代码形式对 BoxObj 对象及其相关方法中的关键操作进行描述。BoxObj 对象中有 2 个属性，属性 cur_val 存储元素当前的值，属性 ori_val 存储原始值。由于程序中存在 if 和 switch 语句嵌套的情况，会导致反事实执行体也发生嵌套，因此为了存储元素在每一层反事实执行体中对应的原始值，将 ori_val 定义成数组的形式，并调用 Set()方法将原始值存储到该数组中的对应位置。

```
1. function BoxObj(original_val, current_val) {
2.     this. cur_val=current_val;
3.     this. ori_val=[];
4.     this. ori_val=Set(this. ori_val, original_val);
5. }
6.
7. function ReadObj(obj) {
8.     if (obj instanceof BoxObj) {
9.       if(in_CounterFactual()) return obj. cur_val;
10.        else return Get(obj. ori_val);
11.    }
12.    return obj;
13. }
14.
15. function WriteObj(obj, value) {
16.    if(in_CounterFactual())
17.      return new BoxObj(obj, value);
18.    return value;
19. }
```

图 5　undo 方法包裹对象

为了保证包裹对象不被篡改，将其实现成 V8 的一个内置 JavaScript 对象，使得目标分析程序无法读取和修改包裹对象。只有在 V8 编译过程中动态插入的 undo 插桩代码能够通过 2 个内部方法对包裹对象进行操作。

(1)ReadObj()：在元素的读操作中进行插桩，调用该方法。若该元素不是包裹对象，则说明其没有在反事实执行体中经历赋值操作，直接返回；否则需要继续判断当前是否还未离开反事实执行体，若是，则直接返回包裹对象中的当前值 cur_val，否则需要进行 undo，调用 Get()方法从 ori_val 数组中读取对应的原始值返回。

(2)WriteObj()：在元素的写操作中进行插桩，调用该方法。若当前不在反事实执行体中，则不需保存元素的原始值，将待赋给元素 obj 的值 value 直接返回；否则，需要保留元素 obj 的原始值，将其与 value 传入新生成的包裹对象中，并返回该包裹对象。

3　反事实执行系统的实现

本文的反事实执行方法首先在 V8 JavaScript 引擎的语法解析阶段以抽象语法树节点的形式将反事实执行体动态插入，以执行更多的分支路径；然后在 V8 引擎的编译阶段动态插入 undo 插桩代码，以实现按需 undo。下面分别对这两个插桩过程进行描述。

3.1　反事实执行体插桩的实现

V8 引擎中 if 语句和 switch 语句的语法解析函数分别为 ParseIfStatement()和 ParseSwitchStatement ()。以 if 语句为例对反事实执行体的插桩过程进行说明，图 6 为插桩后的 ParseIfStatement()函数。

```
1. ParseIfStatement(…) {
2.     ……
3.     Expression * condition=ParseExpression(…);
4.     Statement * then_statement=ParseStatement(…);
5.     Statement * else_statement=NULL;
6.     if (peek()==Token::ELSE) {
7.       else_statement=ParseStatement(…);
8.     } else {
9.       else_statement=NewEmptyStatement(…);
10.    }
11.    IfStatement *  IfStatement=NewIfStatement(
12.        condition, then_statement, else_statement, …);
13.    Block * block=NewBlock(…);
14.    IfStatement * NotIfStatement=NewIfStatement(
15.      !condition, then_statement, else_statement, …);
16.    IfStatement * CEIfStatement=NewIfStatement(
17.      isUndeterminateCall, NotIfStatement, …);
18.    block->AddStatement(CEIfStatement);
19.    block->AddStatement(IfStatement);
20.    return block;
21.}
```

图 6　if 语句插桩方法

图 6 中，插桩代码位于第 13－20 行，首先将条件取反生成 NotIfStatement，该 if 语句中的 then 分支和 else 分支与原 if 语句相同；然后将 NotIfStatement 放入反事实执行体 CEIfStatement 的 then 分支中，在 CEIfStatement 的条件中对原 if 语句条件进行确定性判定；最后再将反事实执行体 CEIfStatement 和原始 if 语句 IfStatement 放入一个块中返回。

3.2 按需 undo 方法插桩的实现

本节以语句 t=v1 op v2 为例，介绍如何在 V8 编译过程中进行插桩从而实现按需 undo 方法。该语句包含二元操作和赋值操作，在 V8 引擎中对应的编译函数分别为 VisitArithmeticExpression()和 VisitAssignment()，插桩后如图 7 所示，其中的插桩代码用粗斜体标识。

```
1. VisitArithmeticExpression(expr) {
2.     ……
3.     VisitForStackValue(left);
4.     __ pop(r0);
5.     __ InvokeBuiltinReadObj(r0);
6.     __ push(r0);
7.     VisitForAccumulatorValue(right);
8.     __ InvokeBuiltinReadObj(r0);
9.     ……
10. }
11.
12. VisitAssignment(expr) {
13.     ……
14.     VisitForAccumulatorValue(expr->value());
15.     EmitVariableLoad(expr->target(), r1, …);
16.     __ InvokeBuiltinWriteObj(r1, r0);
17.     EmitVariableAssignment(expr->target(), …);
18.     ……
19. }
```

图 7 undo 插桩方法

VisitArithmeticExpression()函数中，在读取左右操作数时，根据按需 undo 方法，分别在第 5 行和第 8 行插入插桩代码，调用方法 ReadObj()，对操作数做 undo 处理。二元操作结束后，进入赋值操作的编译函数 VisitAssignment()，插桩代码为第 15 行和第 16 行。第 14 行先读取赋值语句等号右边的值，并将其存入 r0 寄存器，然后在第 15 行读取赋值语句等号左边的原始值，将其存入 r1 寄存器。接着在第 16 行调用方法 WriteObj()，若当前位于反事实执行体中，则保存原始值到一个包裹对象中，并返回该包裹对象用于最终的赋值操作。

4 实验评估

使用本文实现在 V8 JavaScript 引擎中的反事实执行方法，在摩托罗拉 MOTO G(Android-5.0)真实手机设备上对 HTML5 混合型安卓应用中的 JavaScript 代码进行动态分析。在实验过程中观察能否执行条件不成立的非当前分支路径，并截获其中参数为固定字符串的 eval，document.write 等动态代码生成方法，以评估其有效性。同时，针对应用运行过程中的时间开销和空间开销分别进行测量，以评估本文反事实执行方法的性能开销。

4.1 功能评估

在目前收集到的 HTML5 混合型安卓应用中，存在 3 个应用在当前条件不成立的分支路径下，调用了参数为固定字符串的 eval，document.write 等动态代码生成方法。实验中，反事实执行方法成功检测出了这 3 个应用中在非当前路径下的确定动态生成代码信息，有效扩大了测试路径的覆盖面。

下面以其中一个应用 individuell.biz.app.eisenkappel 的检测过程为例进行说明。实验中在非当前路径下检测出的确定动态生成代码信息位于图 8 的 if 语句中，该 if 语句的条件是对当前是否位于 ios 平台进行判定，并在其中通过 document.write 方法将常量字符串转换为代码进行执行。

```
if ((/iphone|ipod|ipad/gi).test(navigator.appVersion) && ("standalone"
in navigator) && !navigator.standalone) {
    document.write('<link rel="stylesheet" href="assets\/add-bubble\/
    style\/add2home.css">');
    document.write('<script type="application\/javascript" src="assets\/
    add-bubble\/src\/add2home.js" charset="utf-8"><\/s'+'cript>');
}
```

图 8 确定动态生成代码

由于在 Android 平台中进行实验，因此原 if 语句条件为 false，正常情况下动态分析不会进入该 if 语句执行，因而无法得到其中的确定动态生成代码信息。但在我们的反事实执行方法中，由于平台信息是不确定的，且当前条件为 false，满足该 if 语句对应的反事实执行体条件，因此会执行已复制到反事实执行体中的 if 语句代码，并检测出其中包含的确定动态生成代码信息，如图 9 所示。

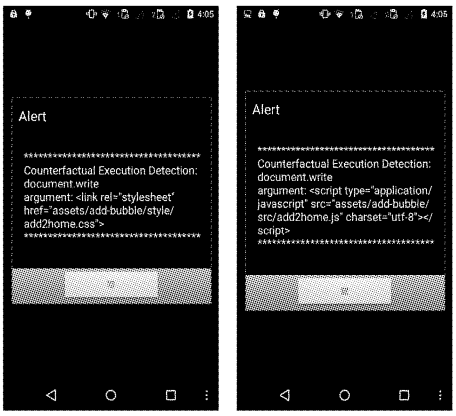

图 9 反事实执行的实验结果

4.2 性能评估

在性能评估实验中，基于应用 individuell.biz.app.eisenkappel，首先对应用中首页的加载时间进行测量，以评估本文的反事实执行方法在实际分析过程中的时间开销；然后使用安卓调试工具 adb 对应用运行时占用的内存大小进行测量，以评估本文方法的空间开销。实验中分别在插桩前和插桩后重复运行该测试应用 100 次，结果表明，插桩后的平均额外时间开销约为 1520ms，平均额外空间开销约为 9396kB，这在分析过程中完全可以接受。

结束语 本文提出了一种基于动态插桩的 JavaScript 反事实执行方法，通过在 JavaScript 引擎的语法解析阶段插入 if 语句和 switch 语句的反事实执行体，使动态分析能够遍历这两种语句中的所有分支路径；同时还设计了一种与之匹配的按需 undo 方法，以消除反事实执行体中赋值操作对程序后续正常操作的影响。本文的反事实执行方法已在 V8 JavaScript 引擎上进行了实现，实验结果表明，该方法能够有效地扩大动态分析中执行路径的覆盖面，且性能开销在分析过程中可以被接受。

(下转第 49 页)

dings of the 26th IEEE International Symposium on Software Reliability Engineering. Washington DC, USA: IEEE, 2015: 553-564.

[16] SHAR L K, TAN H B K, BRIAND L. Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis [C]// Proceedings of the 35th International Conference on Software Engineering. Washington DC, USA: IEEE, 2013, 8104: 642-651.

[17] LV W M, LIU J. The classification and analysis of the security bugs in C/C++ programs[J]. Computer Engineering and Applications, 2005, 41(5): 123-125. (in Chinese)
吕维梅,刘坚. C/C++程序安全漏洞的分类与分析[J]. 计算机工程与应用,2005,41(5):123-125.

[18] MA H T. The principles and defense methods of security bug in computer software[J]. Science & Technology Association Forum, 2009(6): 49. (in Chinese)
马海涛. 计算机软件安全漏洞原理及防范方法[J]. 科协论坛,2009(6):49.

[19] NGUYEN P H, YSKOUT K, HEYMAN T, et al. SoSPa: A system of Security design Patterns for systematically engineering secure systems[C]// Proceedings of the 18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. Washington DC, USA: IEEE, 2015.

[20] YSKOUT K, SCANDARIATO R, JOOSEN W. Do Security Patterns Really Help Designers? [C] // Proceedings of the 37th IEEE/ACM International Conference on Software Engineering. Washington DC, USA: IEEE, 2015: 292-302.

[21] FELDERER M, ZEZH P, BREU R, et al. Model-based security testing: a taxonomy and systematic classification[J]. Software Testing Verification & Reliability, 2016, 26(2): 119-148.

[22] FELDERER M, BÜCHLER M, JOHNS M, et al. Security Testing: A Survey[M]// Advances in Computers. 2016: 1-51.

[23] XIA X, LO D, SHIHAB E, et al. Automatic, high accuracy prediction of reopened bugs[J]. Automated Software Engineering, 2015, 22(1): 75-109.

(上接第 26 页)

参 考 文 献

[1] GUARNIERI S, LIVSHITS V B. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code[C]// Proceedings of the 18th Conference on USENIX Security Symposium. New York, USA: ACM, 2009: 78-85.

[2] GUARNIERI S, PISTOIA M, TRIPP O, et al. Saving the world wide web from vulnerable JavaScript[C]// Proceedings of the 2011 International Symposium on Software Testing and Analysis. New York, USA: ACM, 2011: 177-187.

[3] GUHA A, KRISHAMURTHI S, JIM T. Using static analysis for Ajax intrusion detection[C]// Proceedings of the 18th International Conference on World Wide Web. New York, USA: ACM, 2009: 561-570.

[4] XU W, ZHANG F F, ZHU S C. The power of obfuscation techniques in malicious JavaScript code: A measurement study[C]// Proceedings of the 2012 7th International Conference on Malicious and Unwanted Software. Washington DC, USA: IEEE, 2012: 9-16.

[5] RATANAWORABHAN P, LIVSHITS B, ZORN B G. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications[C]// Usenix Conference on Web Application Development. 2010.

[6] RICHARDS G, HAMMER C, BURG B, et al. The eval that men do[M]// ECOOP 2011-Object-Oriented Programming. Springer Berlin Heidelberg, 2011: 52-78.

[7] RICHARDS G, LEBRESNE S, BURG B, et al. An analysis of the dynamic behavior of JavaScript programs[J]. ACM SIGPLAN Notices, 2010, 45(6): 1-12.

[8] WEI S, RYDER B G. Practical blended taint analysis for JavaScript[C]// Proceedings of the 2013 International Symposium on Software Testing and Analysis. New York, USA: ACM, 2013: 336-346.

[9] CHUGH R, MEISTER J A, JHALA R, et al. Staged information flow for JavaScript[C]// Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, USA: ACM, 2009: 50-62.

[10] VOGT P, NENTWICH F, JOVANOVIC N, et al. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis[C]// The 14th Annual Network & Distributed System Security Symposium. Reston, USA: ISOC, 2007: 12.

[11] SCHÄFER M, SRIDHARAN M, DOLBY J, et al. Dynamic determinacy analysis[C]// Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, USA: ACM, 2013: 165-174.

[12] Google. ChromeV8[EB/OL]. [2016-07-07]. https://developers.google.com/v8.

[13] Adobe. Adobe PhoneGap[EB/OL]. [2016-07-07]. http://phonegap.com.

[14] CHUDNOV A, NAUMANN D A. Inlined information flow monitoring for JavaScript[C]// Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. New York, USA: ACM, 2015: 629-643.

[15] JANG D, JHALA R, LERNER S, et al. An empirical study of privacy-violating information flows in JavaScript Web applications[C]// Proceedings of the 17th ACM Conference on Computer and Communication Security. New York, USA: ACM, 2010: 270-283.