Санкт-Петербургский государственный университет

Кафедра системного программирования

Костюков Юрий Олегович

Поддержка unsafe кода в символьной виртуальной машине .NET

Курсовая работа

Научный руководитель:
ст. преп. Я. А. Кириленко

Санкт-Петербург
2018

Оглавление

Введение 3

1. Постановка задачи 4

2. Обзор предметной области 5
   - 2.1. Символьное исполнение 5
   - 2.2. Horn / SMT решатели 5
   - 2.3. Unsafe код в .NET 6
     - 2.3.1. Арифметика указателей 6
     - 2.3.2. Фиксирование объекта в памяти 6
     - 2.3.3. Фиксированные буферы (fixed sized buffers) 6
     - 2.3.4. Выделение массива на стеке 7
     - 2.3.5. Упаковка структур 7
   - 2.4. Существующие решения 8
     - 2.4.1. Анализ .NET программ 8
     - 2.4.2. Анализ программ с указателями 9

3. Архитектура 11
   - 3.1. Подход к поддержке unsafe кода 13
   - 3.2. Арифметика указателей 13

4. Реализация 14
   - 4.1. Базовая реализация 14
   - 4.2. Поддержка конструкций языка C# 14
   - 4.3. Реализация арифметики указателей 14

Заключение 16

Список литературы 18

Введение

Символьное исполнение давно закрепилось в академии как одна из ключевых техник, используемых для решения задач верификации [11], синтеза [8], генерации тестовых покрытий [4] и других [16, 10, 14]. Символьное исполнение позволяет решать эти задачи полностью автоматически, без участия человека: не нужно снабжать каждую функцию её формальной спецификацией, поддерживать существующие спецификации в актуальном состоянии, искать в них ошибки и т.д.

За последние несколько лет было разработано множество новых подходов, позволяющих решить многие глобальные проблемы символьного исполнения, как то: взрыв числа путей исполнения (path explosion) [9], проблема символьных адресов в памяти [12], проблема выбора путей исполнения (path selection) [13] и др. Это позволило исследователям сконцентрироваться на более конкретных задачах. Одной из них является анализ кучи, что чаще всего подразумевает анализ программ с указателями. Глубокий анализ кучи позволил бы доказывать отсутствие утечек памяти, некорректных доступов к памяти (напр. выход за границы массива), неопределённого поведения.

Unsafe код в .NET [2, Глава 18] даёт возможность программисту обращаться к памяти через указатели, как это делается в C и C++. Это позволяет взаимодействовать с C/C++ кодом и получать ускорение за счёт снижения накладных расходов. Unsafe код много используется, например, в CoreCLR (.NET Core runtime), разработчики которого пытаются сделать код максимально эффективным через более низкоуровневое общение с памятью.

1. Постановка задачи

Целью данной работы является поддержка unsafe кода в символьной виртуальной машине .NET. Для её достижения были поставлены следующие задачи:

- исследовать классические и современные подходы к анализу программ с указателями;
- проанализировать платформу .NET на предмет сложности возможных программ с указателями;
- поддержать корректную символьную арифметику указателей;
- поддержать корректное разыменование указателей в простейших случаях;
- подготовить базовую платформу для более глубокого анализа программ с указателями.

```
1.  void foobar(int a, int b) {
2.     int x = 1, y = 0;
3.     if (a != 0) {
4.        y = 3+x;
5.        if (b == 0)
6.           x = 2*(a+b);
7.     }
8.     assert(x-y != 0);
9.  }
```

Рис. 1: Пример: какие значения `a` и `b` приведут к падению `assert`'а?

2. Обзор предметной области

В данной главе определены основные понятия и приведено описание техник, использованных при реализации поддержки unsafe кода в символьной виртуальной машине .NET.

2.1. Символьное исполнение

Символьное исполнение — техника, позволяющая исследовать различные пути исполнения программы одновременно, поддерживая т.н. символьную память и текущие ограничения на исследуемый путь (path condition). Если символьно исполнять функцию `foobar` на рис. 1 [15], то, например, после исполнения строки 6 символьная память будет в состоянии $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 2(\alpha_a + \alpha_b), y \mapsto 4\}$, а path condition будет $\pi = \alpha_a \neq 0 \wedge \alpha_b = 0$. Здесь видно, что по символьному адресу x лежит символьный терм $2(\alpha_a + \alpha_b)$, а не константа, как было бы в случае конкретного исполнения. В результате анализ программ сводится к анализу символьных термов (например, доказать, что при текущих ограничениях нет доступа по индексу за границами массива). Достаточно хороший обзор современных техник и проблем символьного исполнения может быть найден в статье Baldoni et al [15].

2.2. Horn / SMT решатели

Horn / SMT решатели — решатели логики первого порядка, часто используемые для обнаружения недостижимых веток при символьном исполнении. Обычно принимают на вход формулу логики первого по-

рядка с символами из заранее заданных теорий. Например, формула, описывающая отсортированный массив:

$$(\forall i, j)(i \leq j \leq n \Rightarrow a[i] \leq a[j])$$

Если формула выполнима, решатель вернёт SAT и модель с конкретизированными свободными переменными. Иначе решатель возвращает UNSAT. Так как некоторые теории или их комбинации неразрешимы, решатель может иногда возвращать UNKNOWN или зависать.

Таким образом, используя логические решатели, можно отбрасывать недостижимые ветки программы путём кодирования её в формулу логики первого порядка.

2.3. Unsafe код в .NET

2.3.1. Арифметика указателей

В .NET доступны все классические операции с указателями: можно складывать (и вычитать) указатели и числа, вычитать указатели друг из друга, сравнивать указатели друг с другом, преобразовывать их типы.

2.3.2. Фиксирование объекта в памяти

Ключевое слово `fixed` позволяет *фиксировать* перемещаемые объекты в памяти. Как известно, в .NET ссылочные типы и массивы размещаются в куче, где по решению сборщика мусора могут перемещаться. Чтобы гарантировать, что по указателю на объект мы будем считывать данные действительно из этого объекта в произвольный момент времени, объект необходимо зафиксировать.

2.3.3. Фиксированные буферы (fixed sized buffers)

В .NET если поместить массив в структуру, там на самом деле будет лежать ссылка на него, а не сам массив. Однако можно его в структуру вставить, если объявить его с модификатором `fixed`. Такие массивы

могут быть только линейными, индексироваться только с 0 и тип их элемента может быть только примитивным (`int`, `double` и т.д.).

2.3.4. Выделение массива на стеке

Как было сказано выше, массивы в .NET выделяются в куче. Однако ключевое слово `stackalloc` позволяет выделить массив на стеке. Ограничения те же, что и у фиксированных буферов.

2.3.5. Упаковка структур

Хотя непосредственно возможность различной упаковки структур в .NET не относится к unsafe коду, но в особых случаях она может приводить к реинтерпретациям. Пример такого поведения можно получить, исполнив следующий фрагмент кода:

```
[StructLayout(LayoutKind.Explicit)]
struct StrangeStruct
{
    [FieldOffset(0)]
    public uint a;
    [FieldOffset(2)]
    public uint b;
    [FieldOffset(3)]
    public byte c;
}

static void ReinterpretationViaStructAccess()
{
    StrangeStruct s = new StrangeStruct();
    s.a = 10240;
    s.b = 20480;
    s.c = 255;
}
```

Размещение полей внутри структуры можно схематично представить следующим образом:

<table>
<tr><td>Байты</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr>
<tr><td></td><td colspan="4"><code>uint a</code></td><td colspan="2"></td></tr>
<tr><td></td><td colspan="2"></td><td colspan="4"><code>uint b</code></td></tr>
<tr><td></td><td colspan="3"></td><td><code>byte c</code></td><td colspan="2"></td></tr>
</table>

2.4. Существующие решения

В данном разделе приведен обзор существующих символьных моделей памяти для программ с указателями и инструментов для их интерпретации.

2.4.1. Анализ .NET программ

Pex Pex [18] — инструмент для генерации тестовых покрытий для программ под .NET. Входит в состав Visual Studio 2015 Enterprise под названием IntelliTest. Pex поддерживает работу с unsafe кодом; разработчики утверждают, что он был успешно использован для поиска ошибок в mscorlib (ядре .NET). Однако его недостатком является то, что он базируется на динамическом символьном исполнении [7], а значит фактически анализирует конкретные трассы программ, а не символьные термы. Таким образом, может быть не исследовано значительное число путей исполнения.

SharpChecker SharpChecker [20] — недавно опубликованный инструмент для статического анализа программ под .NET. Несмотря на то что в диссертации [20] нет явной информации о наличии поддержки unsafe кода, в ней используются различные современные техники символьного исполнения, как, например, слияние состояний. Однако так как это статический анализатор, он базируется на переаппроксимации, а значит в его работе могут происходить ложные срабатывания, т.е. он может показывать ошибки там, где их нет.

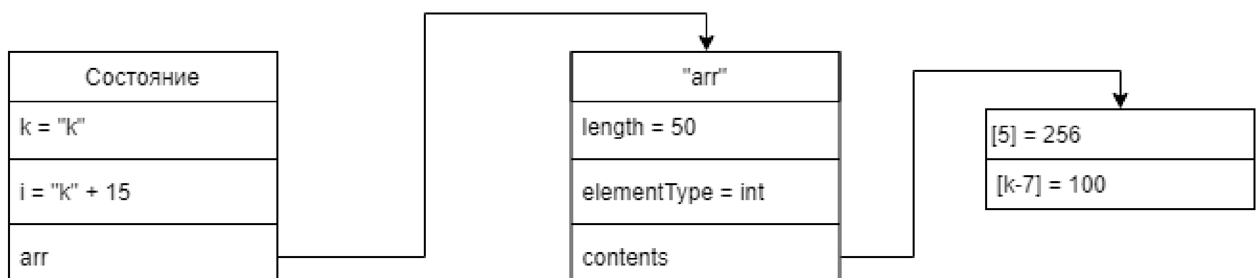
V# Отличительными особенностями символьной виртуальной машины V# являются:

- интерпретация неограниченной рекурсии;
- поддержка символьной типизации;
- композициональность [1].

Символьная память устроена следующим образом: она состоит из вложенных куч. Ключи — локации (могут быть произвольными термами: как константами, так и символьными значениями), значения — конкретные, примитивные символьные константы (числа, булевские значения и т.д.), дочерние кучи (массивы, строки, структуры, классы,...).

Ниже представлены функция и состояние символьной памяти после её исполнения:

```
static void MemoryChange(int k)
{
    int i = k + 15;
    int[] arr = new int[50];
    arr[5] = 256;
    arr[k - 7] = 100;
}
```



2.4.2. Анализ программ с указателями

Несмотря на десятилетия исследований, задача анализа программ с указателями всё ещё не разрешена, и потому интерес к ней по прежнему не спадает [6, 3, 19, 5, 17]. На данный момент не выделилось конкретных подходов, позволяющих проводить качественный анализ в рамках произвольной модели памяти с указателями, однако с каждым годом

эффективность алгоритмов, моделей и подходов только увеличивается. Ключевыми для данной работы являются следующие статьи.

- В статье [6] описывается контекстно-зависимая модель памяти для верификации C/C++ программ. Идея заключается в приближении памяти с указателями её меньшей графовой симуляцией (graph simulation). Сжатие графа значительно ускоряет анализ, однако оно приводит к ощутимым потерям информации, что делает анализ неполным.
- Подход MemSight [3] позволил снизить число необходимых конкретизаций адресов указателей, что привело к увеличению числа исследуемых состояний. Идея подхода состоит в том, что локациями в символьных состояниях должны быть не только константы и символьные термы, но и символьные выражения (например, память с отступом по указателю будет выглядеть так: $\sigma = \{ptr + shift \mapsto data\}$).
- Модель памяти Bugst [19] предоставляет следующие интересные возможности: можно работать с памятью символьного размера, а также производить «мультизаписи» по некоторому адресу (аналог записи в `union` языка C).

Несмотря на заметные успехи этих подходов, их всё же приходится модифицировать, потому что они либо аппроксимируют реальную память (что приводит к неточному, хоть и быстрому анализу), либо предлагают модель, проигрывающую по сравнению с текущей моделью проекта V#.

3. Архитектура

На рисунке 2 представлена схема работы V#.

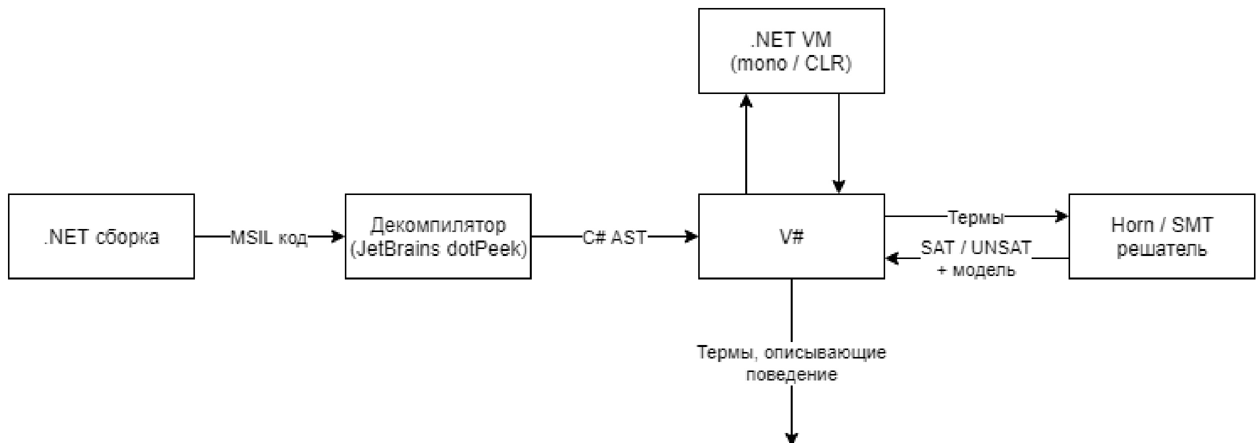


Рис. 2: Схема работы V#

Символьное исполнение происходит итеративно: C# AST кодируется в символьные термы, которые вместе с ограничениями на пути исполнения передаются решателю. Решатель возвращает нам SAT / UNSAT / UNKNOWN и модель, исходя из которых мы отбрасываем какие-то ветки исполнения. В результате работы символьной виртуальной машины получаются термы, описывающие поведение программы, и некоторое символьное состояние памяти.

На рисунке 3 представлена структура проекта V#.

Модуль SILI кодирует дерево C# кода в термы и передаёт их в ядро SILI.Core. Ядро же непосредственно реализует всю нетривиальную логику работы символьной виртуальной машины.

Часть архитектуры ядра, связанная с поддержкой unsafe кода, представлена на рисунке 4.

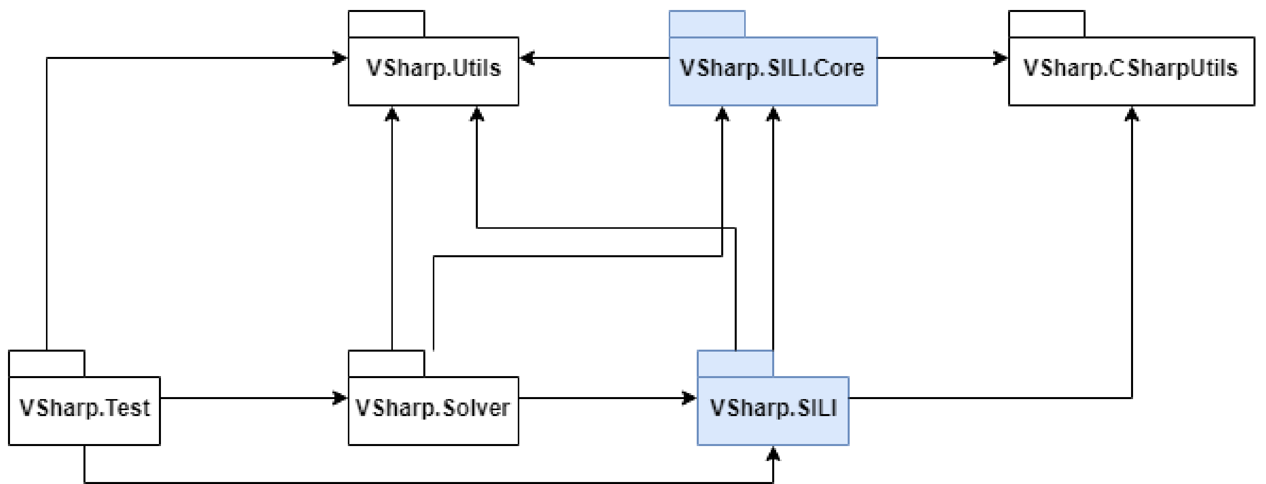


Рис. 3: Структура проекта V# [1]

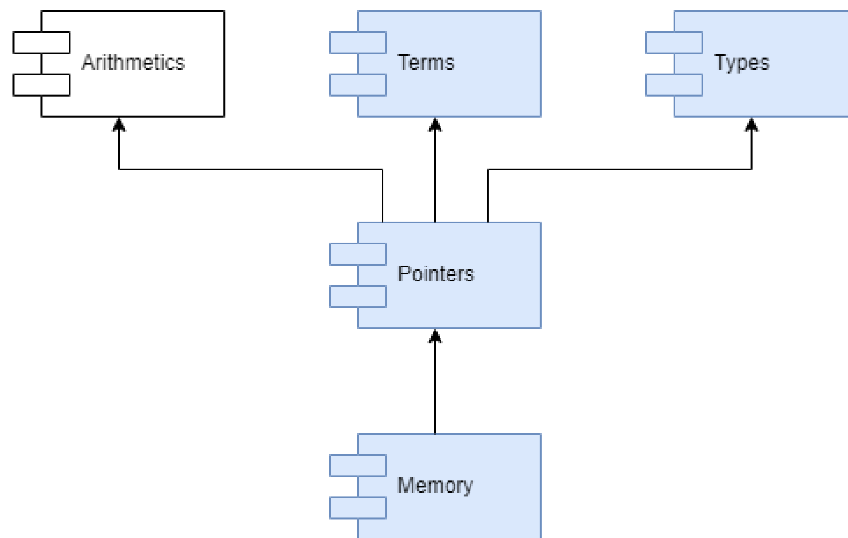


Рис. 4: Модуль указателей в иерархии модулей

Модули на рисунке 4 выполняют следующие функции:

- `Arithmetics` — модуль сокращения числовой арифметики;
- `Terms` — модуль для конструирования символьных термов;
- `Types` — модуль для конструирования символьных типов;
- `Memory` — модуль для работы с символьной памятью.

[1]Здесь и далее голубым цветом помечены модули, затронутые этой работой

3.1. Подход к поддержке unsafe кода

Поддержка unsafe кода в .NET сводится к поддержке символьной арифметики указателей и символьной реинтерпретации: любые сдвиги указателей и доступы к элементам массива сводятся к арифметике указателей, а приведения типов и разыменования (после сдвигов) — к реинтерпретации.

3.2. Арифметика указателей

Основная сложность поддержки арифметики указателей заключалась в сохранении информации при вычитании указателей. Рассмотрим следующий фрагмент кода:

```
static int* Func(int* p, int* q, int* r)
{
    long d1 = p - q;
    long d2 = q - r;
    long d3 = d1 + d2; // (p - q) + (q - r) = p - r
    return r + d3;     //        r + (p - r) = p
}
```

Здесь нужно вернуть указатель `p`. Но перед этим во всех `d`i должны быть сохранены разности указателей. Основная идея поддержки корректного сокращения таких разностей заключается в их представлении в виде мультимножеств. Так, например, `d1` будет выглядеть следующим образом: $\{\, p : 1, q : -1 \,\}$, а выражение `p + 2 * d1`: $\{\, p : 3, q : -2 \,\}$.

На основании такого представления можно судить о корректности разыменуемых адресов. Например, разыменование (в общем случае) `p + 2 * d1` нельзя провести из-за того что `d1` может различаться от запуска к запуску (т.к. `p` и `q` могут находиться в куче, например указывать на два разных массива).

4. Реализация

4.1. Базовая реализация

Для добавления поддержки unsafe кода в символьную виртуальную машину .NET были проведены следующие работы:

- система типов расширена новым типом `Pointer`;
- к каждому конструктору ссылочных типов (с которыми возможна реинтепретация) было добавлено поле с опциональным типом (которым данные будут реинтерпретироваться);
- добавлен новый конструктор термов для сдвигов, содержащий локацию и символьный сдвиг в байтах;
- функции, осуществляющие приведение типов и доступ к памяти, расширены на термы-указатели.

4.2. Поддержка конструкций языка C#

В модуле обработки синтаксических деревьев SILI были поддержаны следующие конструкции языка C#:

- `stackalloc` — выделение символьного линейного массива на стеке;
- `fixed` — простая развёртка (т.к. эта конструкция влияет лишь на поведение сборщика мусора, чьё поведение неспецифицировано, а значит не нуждается в поддержке);
- `sizeof` — реализован через рефлексию.

4.3. Реализация арифметики указателей

Также был написан модуль сокращения символьной арифметики указателей. Он устроен следующим образом:

- разности указателей помещаются в мультимножества, обёрнутые в символьные константы;
- упрощение вычитания сводится к упрощению сложения или разбору случаев: если мы имеем дело с термом с отступом, следует

отдельно рассмотреть его отступ (это может быть мультимножество с указателями, тогда возможно сокращение);

- упрощение сложения также разбирает термы с отступами и сводится к операциям над мультимножествами;
- после всех упрощений происходит проверка на вырожденность: если в мультимножестве всего один указатель, то его можно «вынуть»;
- чтобы сократить сложное выражение с числами и указателями, мы пробегаем по дереву этого выражения и собираем все мультимножества в одно, не затрагивая другие числа (о них заботится модуль упрощения арифметики).

Заключение

В ходе работы получены следующие результаты:

- исследованы классические и современные подходы к анализу программ с указателями;
- проанализирована платформа .NET на предмет сложности возможных программ с указателями;
- поддержана корректная символьная арифметика указателей;
- поддержано корректное разыменование указателей в простейших случаях;
- подготовлена базовая платформа для более глубокого анализа программ с указателями;
- результаты работы представлены на конференции «Современные технологии в теории и практике программирования 2018».

Дальнейшее направление работы

План дальнейшей работы на текущий момент указан ниже.

1. Архитектура структур:
   (a) проанализировать возможные архитектуры структур, в т.ч. текущую (кучи) и наиболее перспективную (интервальные деревья);
   (b) реализовать наиболее оптимальную архитектуру структур.
2. Стек и архитектура указателей:
   (a) проанализировать спецификацию C# на предмет наличия неспецифицированного поведения (в частности при отступах на стеке);
   (b) проанализировать текущую архитектуру указателей, которая в дальнейшем будет использоваться при реинтерпретации;
   (c) реализовать наиболее приемлемую архитектуру указателей.
3. Реализация поддержки реинтерпретации на стеке и в куче:
   (a) независимые реализации для стека и кучи, если стек специфицирован;

(b) одинаковые реализации в ином случае.

Это планируется сделать в рамках дипломной работы.

Список литературы

[1] Anand Saswat, Godefroid Patrice, Tillmann Nikolai. Demand-driven compositional symbolic execution // International Conference on Tools and Algorithms for the Construction and Analysis of Systems / Springer. — 2008. — P. 367–381.

[2] C# language specification. Version 6. — 2016. — Режим доступа: https://github.com/ljw1004/csharpspec/blob/gh-pages/CSharp%20Language%20Specification.pdf?raw=true (дата обращения: 16.05.2018).

[3] Coppa Emilio, D'Elia Daniele Cono, Demetrescu Camil. Rethinking pointer reasoning in symbolic execution // Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering / IEEE Press. — 2017. — P. 613–618.

[4] Efficient observability-based test generation by dynamic symbolic execution / Dongjiang You, Sanjai Rayadurgam, Michael Whalen et al. ; IEEE. — 2015. — P. 228–238.

[5] From low-level pointers to high-level containers / Kamil Dudka, Lukáš Holík, Petr Peringer et al. // International Conference on Verification, Model Checking, and Abstract Interpretation / Springer. — 2016. — P. 431–452.

[6] Gurfinkel Arie, Navas Jorge A. A Context-Sensitive Memory Model for Verification of C/C++ Programs // International Static Analysis Symposium / Springer. — 2017. — P. 148–168.

[7] Kolawa Adam K, Salvador Roman. Method and system for generating a computer program test suite using dynamic symbolic execution of Java programs. — 1998. — US Patent 5,784,553.

[8] Mechtaev Sergey, Yi Jooyong, Roychoudhury Abhik. Angelix: Scalable multiline program patch synthesis via symbolic analysis / IEEE. — 2016. — P. 691–701.

[9] Mitigating Path Explosion in Symbolic Execution via Branches Merging / Yunfei Su, Mengjun Li, Chaojing Tang, Rongjun Shen // International Journal of Applied Engineering Research. — 2016. — Vol. 11, no. 17. — P. 9159–9165.

[10] Pasareanu Corina S, Phan Quoc-Sang, Malacaria Pasquale. Multi-run side-channel analysis using Symbolic Execution and Max-SMT / IEEE. — 2016. — P. 387–400.

[11] Păsăreanu Corina S, Visser Willem. Verification of Java programs using symbolic execution and invariant generation / Springer. — 2004. — P. 164–181.

[12] Schwartz Edward J, Avgerinos Thanassis, Brumley David. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask) // Security and privacy (SP), 2010 IEEE symposium on / IEEE. — 2010. — P. 317–331.

[13] Selective Symbolic Execution / Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, George Candea // Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep). — 2009.

[14] Sethu Ramesh. Systems and methods to reverse engineer code to models using program analysis and symbolic execution. — Google Patents, 2018. — US Patent App. 15/268,011.

[15] A Survey of Symbolic Execution Techniques / Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia et al. // ACM Comput. Surv. — 2018. — Vol. 51, no. 3.

[16] Symnet: Scalable symbolic execution for modern networks / Radu Stoenescu, Matei Popovici, Lorina Negreanu, Costin Raiciu // Proceedings of the 2016 ACM SIGCOMM Conference / ACM. — 2016. — P. 314–327.

[17] Tan Tian, Li Yue, Xue Jingling. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata // Proceedings

of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation / ACM. — 2017. — P. 278–291.

[18] Tillmann Nikolai, De Halleux Jonathan. Pex–white box test generation for. net // International conference on tests and proofs / Springer. — 2008. — P. 134–153.

[19] Trtık Marek, Strejcek Jan. Symbolic Memory with Pointers⋆ // Automated Technology for Verification and Analysis: 12th International Symposium, ATVA 2014, Sydney, Australia, November 3-7, 2014, Proceedings / Springer. — Vol. 8837. — 2014. — P. 380.

[20] Кошелев Владимир Константинович. Межпроцедурный статический анализ для поиска ошибок в исходном коде программ на языке C#. — 2017. — Режим доступа: http://www.ispras.ru/dcouncil/docs/diss/2017/koshelev/koshelev.php (дата обращения: 16.05.2018).