DOI: 10.15514/ISPRAS-2022-34(3)-1

Исследование свойств алгоритма слайсинга предиката пути

А.В. Вишняков, ORCID: 0000-0003-1819-220X <vishnya@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. Безопасный цикл разработки ПО (SDL) применяется для повышения надежности и защищенности программного обеспечения. В жизненный цикл программы добавляются этапы для проверки свойств ее безопасности. Среди прочего повсеместно применяется фаззинг-тестирование, которое позволяет обнаруживать аварийные завершения и зависания анализируемого кода. Гибридный подход, совмещающий в себе фаззинг и динамическую символьную интерпретацию, показал еще большую эффективность, чем классический фаззинг. Более того, символьная интерпретация позволяет добавлять дополнительные проверки, называемые предикатами безопасности, которые ищут ошибки работы с памятью и неопределенное поведение. Данная статья исследует свойства и характеристики алгоритма слайсинга предиката пути, который позволяет устранять избыточные ограничения из предиката пути без потери точности. В статье доказывается, что алгоритм конечен и не теряет решений. Более того, производится оценка асимптотической сложности алгоритма.

Keywords: динамическая символьная интерпретация; DSE; предикат пути; слайсинг; устранение избыточных ограничений; анализ бинарного кода

Для цитирования: Вишняков А.В. Исследование свойств алгоритма слайсинга предиката пути. Труды ИСП РАН, том 34, вып. 3, 2022 г., стр. 7-12. DOI: 10.15514/ISPRAS–2022–34(3)–1

Благодарности: Работа поддержана грантом РФФИ № 20-07-00921 А

Analyzing properties of path predicate slicing algorithm

A.V. Vishnyakov, ORCID: 0000-0003-1819-220X <vishnya@ispras.ru>
Ivannikov Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia

Abstract. Security development lifecycle (SDL) is applied to improve software reliability and security. It extends program lifecycle with additional testing of security properties. Among other things, fuzz testing is widely used, which allows one to detect crashes and hangs of the analyzed code. The hybrid approach that combines fuzzing and dynamic symbolic execution showed even greater efficiency than classical fuzzing. Moreover, symbolic execution empowers one to add additional runtime checks called security predicates that detect memory errors and undefined behavior. This article explores the properties of the path predicate slicing algorithm that eliminates redundant constraints from a path predicate without accuracy loss. The article proves that the algorithm is finite and does not lose solutions. Moreover, the algorithm asymptotic complexity is estimated.

Keywords: dynamic symbolic execution; DSE; path predicate; slicing; irrelevant constraints elimination; binary analysis

For citation: Vishnyakov A.V. Analyzing properties of path predicate slicing algorithm. Trudy ISP RAN/Proc. ISP RAS, vol. 34, issue 3, 2022, pp. 7-12 (in Russian). DOI: 10.15514/ISPRAS–2022–34(3)–1

Acknowledgments: This work was supported by RFBR grant 20-07-00921 A

1. Введение

Безопасный цикл разработки ПО (SDL) [1-3] применяется для повышения надежности и защищенности программного обеспечения. В жизненный цикл программы добавляются этапы для проверки свойств ее безопасности. Среди прочего повсеместно применяется фаззинг-тестирование [4, 5], которое позволяет обнаруживать аварийные завершения и зависания анализируемого кода. Гибридный подход [6-9], совмещающий в себе фаззинг и динамическую символьную интерпретацию [10, 11], показал еще большую эффективность, чем классический фаззинг. Более того, символьная интерпретация позволяет добавлять дополнительные проверки, называемые предикатами безопасности [12], которые ищут ошибки работы с памятью и неопределенное поведение.

Динамическая символьная интерпретация осуществляется следующим образом. Сначала каждому входному байту программы (например, байту входного файла) ставится в соответствие свободная символьная переменная. Далее производится интерпретация конкретного пути выполнения программы, заданного некоторыми фиксированными входными данными. Инструкции, которые оперируют над символьными значениями (зависящими от входных данных) порождают символьные выражения (формулы) в соответствии с их операционной семантикой. Изменения памяти и регистров обновляют символьное состояние, которое содержит отображение из регистров и байтов памяти в символьные выражения. Каждое условное ветвление добавляется в предикат пути. Конъюнкция ограничений из предиката пути имеет модель — значения символьные переменных (входных байтов программы), которые проведут программу по тому же пути выполнения.

Обычно выделяют две задачи, которые позволяет решать динамическая символьная интерпретация: увеличение покрытия кода (открытие новых путей выполнения) и поиск ошибок.

Открытие новых путей осуществляется путем инвертирования условия переходов. Для этого составляется конъюнкция ограничений из предиката пути до инвертируемого перехода с его отрицанием. Таким образом, модель для полученного предиката проведет программу по другому пути выполнения.

Поиск ошибок [12] производится за счет решения предикатов безопасности, описывающих условие возникновение ошибки (например, равенства делителя нулю). Если аналогично составить конъюнкцию предиката пути и предиката безопасности, то в результате решения могут получиться входные данные, активирующие ошибку. Таким образом, предикат пути отвечает за достижение программой точки проявления ошибки, а предикат безопасности — за ее реализацию.

2. Алгоритм слайсинга предиката пути

Алгоритм 1 слайсинга предиката пути [13] (вдохновлен KLEE [14]) позволяет устранять избыточные ограничения из предиката пути без потери точности. Алгоритм принимает на вход целевое ограничение *cond* (условие для инвертирования перехода или предикат безопасности) и конъюнкцию ограничений из предиката пути П до точки проверки целевого ограничения. Предварительно вычисляются используемые символьные переменные для каждого ограничения в предикате пути и целевого условия. Их вычисление производится путем обхода абстрактных синтаксических деревьев для символьных выражений. Изначально множество переменных слайсинга *vars* содержит символьные переменные, от которых зависит *cond*. Далее происходят проходы по всем ограничениям *c* из предиката пути П. Если переменные *vars* пересекаются с используемыми переменными в *c*, то множество *vars* пополняется символьными переменными из *c*. Проходы происходят до тех пор, пока множество *vars* пополняется новыми элементами. В результате будут получены все переменные *vars*, которые транзитивно зависят от переменных в целевом ограничении *cond*. В конце производится последний проход по ограничениям из предиката пути П. Ограничения

из П, используемые переменные которых пересекаются с *vars*, составляются в конъюнкцию вместе с ограничением *cond*. Полученный предикат Π_S будет отвечать решаемой задаче: инвертированию перехода или проверке предиката безопасности. Таким образом, берется лишь часть ограничений из предиката пути. При этом далее будет показано, что алгоритм решает ту же задачу.

Алгоритм 1. Алгоритм слайсинга предиката пути
Algorithm 1. Path predicate slicing algorithm

```
Входные данные: cond – ограничение для инвертирования целевого  перехода
(или предикат безопасности), П – предикат пути (ограничения пути до
целевого перехода).
    vars ← used_variables (cond)                ▷переменные слайсинга
    change ← vars
    while change ≠ ∅
      change ← vars
      for all c ∈ П                 ▷итерирование по ограничениям пути
        if vars ∩ used_variables(c) ≠ ∅ then
          vars ← vars ∪ used_variables (c)
      change ← vars \ change
    П_S ← cond   ▷ограничение для инвертирования/предикат безопасности
    for all c ∈ П do                ▷итерирование по ограничениями пути
      if vars ∩ used_variables(c) ≠ ∅ then
        П_S ← П_S ∧ c
    return П_S
```

Перед тем как приступить к изучению свойств и характеристик алгоритма 1 слайсинга предиката пути, приведем необходимые определения.

Определение 1. *Модель для предиката* $P(v_1,\dots,v_n)$ *– это такая подстановка* $v_i \leftarrow c_i,\ i = 1..n$, *где* c_i *– константы, при которой предикат истинен:* $P(c_1,\dots,c_n) \equiv 1$.

Определение 2. *Символьная переменная – свободная переменная, которая ставится в соответствие с каждым входным байтом программы.*

Определение 3. *Предикат пути* П, *построенный для пути выполнения, заданного конкретными значениями символьных переменных* α_i *(входными байтами программы) – это конъюнкция ограничений над символьными переменными и константами, каждая модель которой проведет программу по тому же пути выполнения.*

Следует отметить, что из определения предиката пути следует, что значения символьных переменных α_i являются для него моделью, т.к. это изначальные входные байты программы, которые проводят ее по заданному пути.

3. Свойства и характеристики алгоритма слайсинга предиката пути

Теорема 1. *Алгоритм 1 слайсинга предиката пути конечен, т.е. завершается за конечное число итераций.*

Доказательство. Алгоритму требуются предварительно вычисленные множества используемых символьных переменных *used_variables (c)* для всех ограничений в предикате пути и ограничения *cond* ($c \in \Pi \vee c = cond$). Производится обход деревьев символьных выражений (для ограничений *c*) в ширину и составляется множество номеров, используемых в них переменных. Алгоритм поиска в ширину является конечным для деревьев с конечным числом вершин, что справедливо для деревьев символьных выражений. Множество

ограничений в предикате пути П также конечно. Следовательно, алгоритм вычисления используемых символьных переменных конечен.

Проход по всем ограничениям $c \in \Pi$ осуществляется за конечное число шагов, т.к. множество ограничений в предикате пути П конечно. Из этого следует, что вложенный цикл на каждой итерации **while**, который обновляет *vars*, и цикл, конструирующий итоговый предикат Π_S, конечны. Для доказательства конечности всего алгоритма 1 остается только доказать, что итераций цикла **while** с условием завершения *change* = ∅ конечное число.

Каждая итерация цикла **while** заключается в переборе всех ограничений *c* из предиката пути П с возможным обновлением множества *vars*. При этом, если на итерации не происходит изменение множества *vars*, выполняется условие завершения цикла.

Для $i \in \mathbb{N}$ обозначим $V_i = vars_i$ – множество переменных слайсинга *vars* в начале *i*-й итерации. Тогда $V_1 = used\_variables(cond)$, а $V_i \subseteq V_{i+1}$, так как множество переменных *vars* на каждой итерации включает в себя множество переменных предыдущей итерации: $V_{i+1} = V_i \cup U_i$, где $V_i \cap U_i = \emptyset$, и U_i – множество добавленных в *vars* переменных.

Заметим, что на каждой итерации возможны 2 варианта работы алгоритма:

1) происходит перебор всех ограничений из предиката пути П без изменения множества V_i, и цикл завершается;
2) множество V_i пополняется новыми элементами.

Покажем, что цикл **while**, отбирающий символьные переменные для слайсинга, имеет конечное число итераций. Докажем «от противного»: предположим, что цикл бесконечен. Следовательно, никогда не выполняется условие завершения цикла. Это возможно (в соответствии с вариантами его работы) в том и только в том случае, когда на каждой итерации *i* множество V_i пополняется хотя бы одним новым элементом x_i (при этом x_i может быть не единственным новым элементом на *i*-й итерации: $V_{i+1} = V_i \cup U_i$, где $V_i \cap U_i = \emptyset$, а $x_i \in U_i$). Более того, множества V_i строго вложены ($V_i \subset V_{i+1}$), в противном случае стало бы истинным условие завершения цикла $change = \emptyset$ (другими словами, $V_i = V_{i+1}$). Исходя из строгой вложенности множеств V_i, все элементы x_i различны. Тогда существует бесконечная последовательность $\{x\}_i$, такая что $x_i \in V_{i+1}$, состоящая из различных элементов. Обозначим множество элементов последовательности как $X = \{x \mid x \in \{x\}_i\}$. Поскольку каждое из множеств V_i является подмножеством множества V_P всех символьных переменных программы, то множество элементов последовательности $X \subseteq V_P$. Следовательно, мощность множества $|X| \leq |V_P|$. Но множество всех символьных переменных V_P конечно: $|V_P| = N$, где $N \in \mathbb{N}$. А значит, $|X| \leq N$, и последовательность $\{x\}_i$ не может содержать бесконечное число различных элементов. Получили противоречие с изначальным определением $\{x\}_i$, из чего следует, что цикл отбора символьных переменных для слайсинга конечен, а значит, и алгоритм 1 слайсинга предиката пути конечен.

Теорема 2. *Пусть предикат пути П строился по пути выполнения, заданному конкретными значениями символьных переменных* $v_i \leftarrow \alpha_i,\ i = 1..n,\ n \in \mathbb{N}$, *где* v_i *– символьная переменная, а* α_i *– ее значение (входной байт программы). Если конъюнкция ограничений предиката пути* П *и ограничения cond (*$\Pi \wedge cond$*) выполнима, то предикат* Π_S *(полученный в результате применения алгоритма 1 слайсинга предиката пути к* П *и cond) тоже выполним, и для любой его модели* $v_j \leftarrow \beta_j, j \in J \subseteq \{1,\dots,n\}$ *подстановка* $v_j \leftarrow \beta_j$, $v_k \leftarrow \alpha_k$, $k \in K = \{1,\dots,n\} \setminus J$ *(α_k – входные байты программы) является моделью для предиката* $\Pi \wedge cond$.

Доказательство. Исходя из описания алгоритма, предикат Π_S состоит из конъюнкции части ограничений из предиката пути П и ограничения *cond*: $\Pi_S = P \wedge cond$, $\Pi = P \wedge Q$. Предикат $\Pi \wedge cond$ выполним, а значит, и предикат $P \wedge Q \wedge cond$ выполним. Следовательно, предикаты $P \wedge cond = \Pi_S$ и Q тоже выполнимы.

Обратим внимание на последнюю итерацию цикла **while**, когда выполняется условие выхода из цикла, а именно множество переменных *vars* остается неизменным в результате

итерирования по всем ограничениям *c* из предиката пути Π. Для этого на каждой итерации вложенного цикла **for** не должно происходить пополнение множества *vars*. Это возможно в том и только том случае, когда либо *c* не использует ни одну переменную из *vars*, либо $used\_variables(c) \subseteq vars$ (в противном случае множество *vars* пополнилось бы переменными из $used\_variables(c) \setminus vars$). Другими словами, $\forall c \in \Pi\ (used\_variables(c) \subseteq vars) \vee (used\_variables(c) \cap vars = \emptyset)$.

Из последнего цикла **for** алгоритма следует, что конъюнкты *P* зависят хотя бы от одной переменной из множества переменных *vars*, а конъюнкты *Q,* напротив, не зависят ни от одной переменной из *vars*. Из доказанного выше утверждения следует, что каждый конъюнкт *P* зависит от подмножества переменных из *vars*. Таким образом, каждый конъюнкт *P* содержит только переменные из *vars* и не содержит других переменных. Значит, множества переменных в предикатах *P* и *Q* не пересекаются.

Из первой строчки алгоритма следует, что множество переменных в *cond* является подмножеством *vars*, а значит, множества переменных в *cond* и *Q* тоже не пересекаются. Следовательно, множество переменных предиката $P \wedge cond = \Pi_S$ не пересекается с множеством переменных предиката *Q*.

Пусть $v_j \leftarrow \beta_j,\ j \in J \subseteq \{1,\dots,n\}$ – модель для предиката Π_S. Тогда $v_k \leftarrow \alpha_k,\ k \in K = \{1,\dots,n\} \setminus J$ – модель для предиката *Q*, т.к. *Q* выполним, множества переменных в Π_S и *Q* не пересекаются, $\Pi = P \wedge Q$, а $v_i \leftarrow \alpha_i$ – модель для предиката пути Π из определения. Осталось найти модель для предиката $\Pi \wedge cond = P \wedge Q \wedge cond = (P \wedge cond) \wedge Q = \Pi_S \wedge Q$. Моделью является только что найденная подстановка непересекающихся множеств переменных: $v_j \leftarrow \beta_j, v_k \leftarrow \alpha_k, j \in J \subseteq \{1,\dots,n\}, k \in K = \{1,\dots,n\} \setminus J$.

Следствие 2.1. *Для получения входных байтов программы, приводящих к инвертированию целевого перехода (или проявлению ошибки), достаточно заменить часть входных байтов v_j на значения β_j из модели для предиката Π_S, полученного в результате применения алгоритма 1 слайсинга предиката пути.*

Теорема 3. *Асимптотическая сложность алгоритма 1 – $O(|\Pi| * |V_P|^2 * log(|V_P|))$, где $|\Pi|$ – число ограничений в предикате пути, а $|V_P|$ – число всех символьных переменных (входных байтов) программы.*

Доказательство. Номера используемых символьных переменных хранятся в двоичных (красно-черных) деревьях поиска. Операция объединения двух деревьев имеет сложность $O(|V_P| * log(|V_P|))$. Операция определения факта пересечения (пересекаются/не пересекаются) двух отсортированных контейнеров имеет линейную сложность $O(|V_P|)$. Определение факта изменения множества *vars* имеет константную сложность: для этого достаточно сравнить количество переменных *vars* в начале и в конце итерации цикла. Конъюнкция двух предикатов также имеет константную сложность, т.к. предикаты представляются абстрактными синтаксическими деревьями, а их конъюнкция равносильна созданию одной вершины.

Предварительно до начала работы алгоритма вычисляются множества используемых символьных переменных для ограничений *c* из предиката пути Π и ограничения *cond*. Для этого производится обход по абстрактным синтаксическим деревьям. Таким образом, вычисление используемых переменных имеет сложность $O(|\Pi| * |V|)$, где $|V|$ – максимальное число вершин в дереве. Следует отметить, что используемые переменные в ограничениях предиката пути Π могут быть вычислены единожды, а алгоритм применен многократно.

Оценим вычислительную сложность непосредственно самого алгоритма 1 слайсинга предиката пути. В худшем случае цикл **while** будет иметь $|V_P|$ итераций. Это возможно, когда на каждой итерации множество *vars* пополняется ровно одной переменной. Таким образом, асимптотическая сложность цикла **while**: $O(|V_P| * |\Pi| * (|V_P| + |V_P| * log(|V_P|))) = O(|\Pi| * |V_P|^2 * log(|V_P|))$. Вычислительная сложность каждой итерации внутреннего цикла

for состоит из суммы сложности пересечения и объединения множеств. Последний цикл **for** осуществляет $|\Pi|$ пересечений множеств: $O(|\Pi| * |V|)$. Итого асимптотическая сложность алгоритма – $O(|\Pi| * |V_P|^2 * log(|V_P|) + |\Pi| * |V_P|) = O(|\Pi| * |V_P|^2 * log(|V_P|))$.

4. Заключение

В данной статье были изучены свойства и характеристики алгоритма слайсинга предиката пути [13]. Было показано, что алгоритм конечен и позволяет уменьшать число избыточных ограничений из предиката пути без потери решений. Более того, была проведена оценка асимптотической сложности алгоритма.

Список литературы / References

[1] M. Howard and S. Lipner. The security development lifecycle. SDL: A Process for Developing Demonstrably More Secure Software. Microsoft Press, Redmond, 2006, 348 p.

[2] ISO/IEC 15408-3:2008: Information Technology – Security Techniques –Evaluation Criteria for It Security – Part 3: Security Assurance Components. ISO Geneva, Switzerland, 2008.

[3] ГОСТ Р 56939-2016: Защита информации. Разработка безопасного программного обеспечения. Общие требования. Национальный стандарт Российской Федерации, 2016 / GOST R 56939-2016: Information Protection. Secure Software Development. General Requirements, National Standard of Russian Federation, 2016 (in Russian).

[4] K. Serebryany. Continuous Fuzzing with libFuzzer and AddressSanitizer. In Proc. of the the 2016 IEEE Cybersecurity Development (SecDev), 2016, p. 157.

[5] A. Fioraldi, D. Maier et al. AFL++: combining incremental steps of fuzzing research. In Proc. of the 14th USENIX Workshop on Offensive Technologies (WOOT 20), 2020, 12 p.

[6] I. Yun, S. Lee et al. QSYM: a practical concolic execution engine tailored for hybrid fuzzing. In Proc. of the 27th USENIX Security Symposium, 2018, pp. 745-761.

[7] S. Poeplau and A. Francillon. Symbolic execution with SymCC: don't interpret, compile! In Proc. of the 9th USENIX Security Symposium (USENIX Security 20), 2020, pp. 181-198.

[8] S. Poeplau and A. Francillon. SymQEMU: compilation-based symbolic execution for binaries. In Proc. of the 2021 Network and Distributed System Security Symposium, 2021, 18 p.

[9] L. Borzacchiello, E. Coppa, and C. Demetrescu. FUZZOLIC: mixing fuzzing and concolic execution. Computers & Security, vol. 108, 2021, article no. 102368.

[10] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In Proc. of the 2010 IEEE Symposium on Security and Privacy, 2010, pp. 317-331.

[11] R. Baldoni, E. Coppa et al. A survey of symbolic execution techniques. ACM Computing Surveys, vol. 51, issue 3, 2018, article no. 50, pp. 1-39.

[12] A. Vishnyakov, V. Logunova et al. Symbolic security predicates: hunt program weaknesses. In Proc. of the 2021 Ivannikov ISPRAS Open Conference (ISPRAS), 2021, pp. 76-85.

[13] A. Vishnyakov, A. Fedotov et al. Sydr: cutting edge dynamic symbolic execution. In Proc. of the 2020 Ivannikov ISPRAS Open Conference (ISPRAS), 2020, pp. 46-54.

[14] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation, pp. 209-224, 2008.

Информация об авторах / Information about authors

Алексей Вадимович ВИШНЯКОВ – младший научный сотрудник отдела компиляторных технологий в Институте системного программирования им. В.П. Иванникова РАН, закончил бакалавриат и магистратуру ВМК МГУ. Сфера научных интересов: компьютерная безопасность, жизненный цикл безопасной разработки (SDL), символьная интерпретация, фаззинг, автоматическое обнаружение ошибок, анализ бинарного кода и компиляторы.

Alexey Vadimovich VISHNYAKOV works for the Compiler Technology Department at Ivannikov Institute for System Programming of the RAS, obtained BSc degree and M.D. in the Faculty of Computational Mathematics and Cybernetics at Lomonosov Moscow State University. Research interests: computer security, security development lifecycle (SDL), symbolic execution, fuzzing, automatic bug detection, binary analysis, and compilers.