

Федеральное государственное бюджетное учреждение
науки Институт системного программирования им.
В. П. Иванникова Российской академии наук

Представление на соискание учёной степени кандидата
физико-математических наук по специальности 2.3.5 Математическое и
программное обеспечение вычислительных систем, комплексов и
компьютерных сетей

Поиск ошибок в бинарном коде методами
динамической символьной интерпретации

Выступающий: А. В. Вишняков
Руководитель: к.т.н. А. Н. Федотов

Москва, 2022

- 81 % программ в 2400 коммерческих кодовых базах содержали хотя бы одну уязвимость (отчет Synopsys 2022).
- Безопасный цикл разработки ПО (SDL) является стандартом индустрии и позволяет обнаруживать ошибки до ввода программы в эксплуатацию.
- Фаззинг обнаруживает ошибки, мутируя входные данные программы и применяя генетические алгоритмы, которые «скрещивают» входные файлы для открытия нового кода.
- Гибридный фаззинг дополнительно применяет динамическую символьную интерпретацию (DSE) для обнаружения сложных состояний программы за счет математического моделирования и учета семантики кода.
- DSE позволяет накладывать ограничения для порождения новых входных данных, активирующих дефекты.

Целью данной работы является разработка метода поиска ошибок в бинарном коде методами динамической символьной интерпретации.

- Метод должен иметь возможность применения в контексте гибридного фаззинга, сочетающего в себе динамическую символьную интерпретацию и фаззинг с обратной связью по покрытию.
- Метод должен обнаруживать ошибки деления на нуль, выхода за границу массива и целочисленного переполнения.
- Метод должен порождать входные данные для подтверждения ошибок.

1. Алгоритм слайсинга предиката пути, позволяющий устранять избыточные ограничения во время динамической символьной интерпретации бинарного кода. Алгоритм основан на анализе зависимостей символьных переменных по данным.
2. Метод построения предикатов безопасности для обнаружения ошибок деления на нуль, целочисленного переполнения и выхода за границу массива во время динамической символьной интерпретации пути выполнения программы. Метод добавляет к предикату пути дополнительные ограничения, которые описывают достаточные условия возникновения ошибки.
3. Метод автоматизированного поиска ошибок при помощи символьных предикатов безопасности во время динамического анализа программ на корпусе входных файлов, полученного в результате гибридного фаззинга.

Обзор существующих методов

- Запуск исследуемой программы на конкретных входных данных.
- Каждый входной байт программы моделируется свободной символьной переменной.
- Инструкции моделируются формулами в соответствии с операционной семантикой.
- Конкретные значения, не зависящие от входных данных, берутся из реального выполнения.
- Символьное состояние содержит текущее отображение регистров и памяти в формулы.
- Изменения регистров (памяти) обновляют символьное состояние.
- Предикат пути содержит ограничения на входные данные, которые описывают исследуемый путь выполнения.

Построение предиката пути и инвертирование перехода

Символьное состояние	Инструкция	Множество формул	Предикат пути Π
$rax = \phi_1, rbx = \phi_2, rcx = \phi_3$	—	\emptyset	<i>true</i>
$rax = \phi_1, rbx = \phi_2, rcx = \phi_3$	cmp rcx, 0 jz .exit	\emptyset	$\phi_3 \neq 0$
$rax = \phi_4, rbx = \phi_2, rcx = \phi_3$	add rax, rbx	$\phi_4 = \phi_1 + \phi_2$	$\phi_3 \neq 0$
$rax = \phi_5, rbx = \phi_2, rcx = \phi_3$	sub rax, 2	$\phi_4 = \phi_1 + \phi_2$ $\phi_5 = \phi_4 - 2$	$\phi_3 \neq 0$
$rax = \phi_5, rbx = \phi_2, rcx = \phi_3$	cmp rbx, 10 jl .exit	$\phi_4 = \phi_1 + \phi_2$ $\phi_5 = \phi_4 - 2$	$\phi_3 \neq 0 \wedge \phi_2 \geq 10$
$rax = \phi_5, rbx = \phi_2, rcx = \phi_3$	cmp rax, 30 jge .exit	$\phi_4 = \phi_1 + \phi_2$ $\phi_5 = \phi_4 - 2$	$\phi_3 \neq 0 \wedge \phi_2 \geq 10 \wedge \phi_5 < 30$

Инвертирование последнего перехода

$$\phi_3 \neq 0 \wedge \phi_2 \geq 10 \wedge \phi_5 \geq 30$$

**Преодоление
избыточной и
недостаточной
помеченности**

- Помеченными считаются данные, которые зависят от пользовательского ввода.
- Недостаточная помеченность возникает, когда значение, вычисление которого зависит от входных данных, не помечено.
- Избыточная помеченность — это ситуация, когда в предикат пути попадает чрезмерное число ограничений.

- KLEE разбивает ограничения в предикате пути на независимые подмножества.
 - Метод применим только к исходному коду — необходимо разработать алгоритм для бинарного кода.
- Моделирование семантики функций: несколько состояний описываются одной формулой.

**Поиск ошибок с
ПОМОЩЬЮ СИМВОЛЬНОЙ
интерпретации**

KLEE добавляет дополнительные условные переходы, которые проверяют наличие ошибки и порождают входные данные для ее подтверждения.

Не применим для поиска ошибок в бинарном коде:

- Для обнаружения целочисленного переполнения требуется собрать программу с санитайзерами — KLEE инвертирует переходы в коде санитайзеров.
- KLEE осуществляет полностью символьную эмуляцию (без запуска программы) с клонированием состояний, что влечет проблемы с масштабированием.

SAVIOR — гибридный фаззер, нацеленный на поиск ошибок выхода за границу массива, битового сдвига на большое значение, целочисленного переполнения.

Не применим для поиска ошибок в бинарном коде:

- Работает с исходным кодом.
- Знаковость и размеры операндов получается с помощью статического анализа исходного кода.
- Требуется применение патчей к коду компилятора — не является переносимым решением.
- Авторы отмечают проблемы с анализом C++ кода.

- Источник ошибки — арифметическая инструкция, где может произойти ошибка целочисленного переполнения.
- Сток ошибки — место в программе, где использование переполненного значения может привести к дальнейшим уязвимостям: выделение памяти (аргумент `malloc`); доступ к памяти (индекс массива); условный переход.
- Знаковость получается на основе анализа помеченных инструкций условных переходов: `JG` (знаковый), `JA` (беззнаковый) и др.

Недостатки:

- Обход срезов графа потока управления в глубину и символьная интерпретация промежуточного представления.
- Входные данные для подтверждения ошибки не генерируются.
- Инструмент и его код недоступны.

**Предлагаемый
алгоритм слайсинга
предиката пути для
устранения избыточных
ограничений**

Алгоритм слайсинга предиката пути

Входные данные: $cond$ — ограничение для инвертирования целевого перехода (или предикат безопасности), Π — предикат пути (ограничения пути до целевого перехода).

Результат: Π_S — предикат с устраненными избыточными ограничениями.

```
 $vars \leftarrow used\_variables(cond)$  /* переменные слайсинга */
```

```
 $change \leftarrow vars$ 
```

```
while  $change \neq \emptyset$  do
```

```
   $change \leftarrow vars$ 
```

```
  /* итерирование по ограничениям пути */
```

```
  forall  $c \in \Pi$  do
```

```
    if  $vars \cap used\_variables(c) \neq \emptyset$  then
```

```
       $vars \leftarrow vars \cup used\_variables(c)$ 
```

```
     $change \leftarrow vars \setminus change$ 
```

```
  /* ограничение для инвертирования/предикат безопасности */
```

```
 $\Pi_S \leftarrow cond$ 
```

```
  /* итерирование по ограничениями пути */
```

```
  forall  $c \in \Pi$  do
```

```
    if  $vars \cap used\_variables(c) \neq \emptyset$  then
```

```
       $\Pi_S \leftarrow \Pi_S \wedge c$ 
```

```
return  $\Pi_S$ 
```


- Предикат пути содержит только релевантные ограничения для инвертирования целевого перехода.
- Математический решатель расходует меньше памяти и времени.
- Устраняются потенциально недостаточно помеченные символьные переменные.
- Решатель возвращает модель для подмножества символьных переменных.
- Остальные байты берутся из начальных входных данных.

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

Теорема 2. Пусть предикат пути Π 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 Π и ограничения $cond$ ($\Pi \wedge cond$) выполнима, то предикат Π_S (полученный в результате применения алгоритма слайсинга предиката пути к Π 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 $\Pi \wedge cond$.

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 слайсинга предиката пути.

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

Экспериментальная оценка эффективности алгоритма

Приложение	Слайсинг выключен			Слайсинг включен		
	Точность	SAT	Запросы	Точность	SAT	Запросы
bzip2recover	100.0 %	2101	5131	100.0 %	2101	5131
cjpeg	100.0 %	50	198	100.0 %	50	197
faad	99.23 %	389	585	99.07 %	430	652
foo2lava	87.1 %	31	6252	87.1 %	31	6127
hdp	25.0 %	464	2427	78.01 %	1037	3828
jasper	0.05 %	1987	5639	99.53 %	6798	18207
libxml2	12.46 %	1043	13520	50.98 %	1069	17532
minigzip	10.73 %	3961	4183	51.47 %	7569	8977
muraster	99.97 %	3235	4739	99.97 %	3228	4726
pk2bm	98.91 %	183	3672	99.45 %	183	3673
pnmhistmap_pgm	99.97 %	3159	4681	99.99 %	17089	25446
pnmhistmap_ppm	99.07 %	107	8247	99.07 %	107	8247
readelf	61.93 %	218	2046	86.47 %	739	6141
yices-smt2	2.5 %	521	2135	78.33 %	2699	9647
yodl	8.31 %	313	5201	57.51 %	313	5201

Точность — процентное отношение числа корректных входных данных для инвертирования переходов (с учетом пути) к общему числу сгенерированных входных данных (SAT).

**Разработанный метод
построения предикатов
безопасности**

Предикат безопасности — дополнительные ограничения на предикат пути, описывающие условие возникновения ошибки (например, равенство делителя нулю).

- Символьно интерпретируется программа (x86/AMD64) на входных данных, которые не приводят к ошибке.
- Конструируется предикат пути, который проверяет наличие ошибок неопределенного поведения и работы с памятью.
- Конъюнкция предиката безопасности с предикатом пути после слайсинга передается математическому решателю.
- Предикат пути отвечает за достижение точки с ошибкой, а предикат безопасности — за ее проявление.
- Если предикат выполним, то порождаются входные данные для подтверждения ошибки.

- Проверяется разыменование символического адреса (который зависит от пользовательских данных).
- Границы определяются из теневого стека и кучи.
- Не всегда можно определить обе границы.
- Базовый адрес массива вычисляется эвристически из константной части символического выражения адреса:
 - $[rdx + rax]$, где rax — константная база массива, а rdx — символический индекс;
- Функции копирования (`memcpy`, `memmove`, `memset` и т. д.) проверяются на переполнение буфера.
- Сначала проверяется наихудшая ситуация — возможность перезаписи адреса возврата.

Обнаружение ошибки целочисленного переполнения

- Целочисленное переполнение часто встречается в бинарном коде — проверка каждой арифметической операции замедляет анализ и приводит к ложным срабатываниям.
- **Источник** — арифметическая инструкция.
- **Сток** — опасное место использования переполненного значения: условные переходы, индекс массива, аргументы функций.
- Предикаты безопасности для беззнакового (CF) и знакового (OF) переполнения истинны, когда соответствующий флаг равен 1.
- Знаковость определяется обратным слайсингом по инструкциям и анализом отобранных условных переходов (например, $j \perp$ — знаковый переход).
- Для функций выделения памяти производится попытка получить размер меньше, чем в оригинальном выполнении.

**Метод
автоматизированного
поиска ошибок при
помощи символьных
предикатов
безопасности
в гибридном фаззинге**

- Гибридный фаззинг: фаззер libFuzzer/AFL++ и инструмент динамической символьной интерпретации Sydr.
- Минимизация корпуса для получения меньшего числа файлов, дающих то же покрытие.
- Проверка предикатов безопасности на каждом файле из минимизированного корпуса:
 - деление на нуль;
 - выход за границы массива;
 - целочисленное переполнение.
- Верификация срабатываний предикатов безопасности на санитайзерах.
 - Срабатывание Sydr подтверждается на той же строке санитайзерами.
 - Санитайзеры выдали новые предупреждения, которых не было на изначальном файле из корпуса.
- Дедупликация срабатываний.

**Результаты поиска
ошибок в бинарном
коде методами
динамической
символьной
интерпретации**

Результаты тестирования на наборе тестов Juliet

Экспериментальная оценка метода построения предикатов безопасности.

CWE	P=N	Текстовые срабатывания			Верификация санитайзерами		
		TPR	TNR	ACC	TPR	TNR	ACC
121: Stack Based Buffer Overflow	188	100%	100%	100%	100%	100%	100%
122: Heap Based Buffer Overflow	376	100%	100%	100%	100%	100%	100%
124: Buffer Underwrite	188	100%	100%	100%	100%	100%	100%
126: Buffer Overread	188	100%	100%	100%	100%	100%	100%
127: Buffer Underread	188	100%	100%	100%	100%	100%	100%
190: Integer Overflow	2580	99.92%	90.89%	95.41%	98.10%	90.89%	94.50%
191: Integer Underflow	1922	99.90%	91%	95.45%	97.45%	91%	94.22%
194: Unexpected Sign Extension	752	100%	100%	100%	100%	100%	100%
195: Signed to Unsigned Conversation	752	99.87%	100%	99.93%	99.87%	100%	99.93%
369: Divide by Zero	564	66.67%	100%	83.33%	66.67%	100%	83.33%
680: Integer Overflow to Buffer Overflow	188	100%	100%	100%	100%	100%	100%
ИТОГО	7886	97.55%	94.83%	96.19%	96.36%	94.83%	95.59%

Апробация метода автоматизированного поиска ошибок на проектах с открытым исходным кодом

Обнаружено **17** новых ошибок в **10** различных проектах (OpenJPEG, Poppler, miniz, unbound и другие):

- 15 ошибок целочисленного переполнения (2 приводят к выходу за границы массива);
- 1 ошибка выхода за границу массива;
- 1 ошибка деления на ноль.

Информация обо всех ошибках доведена до разработчиков, часть из них уже исправлены.

Примеры найденных ошибок

Целочисленное переполнение умножения в OpenJPEG:

```
l_y1 = p_cp->ty0 + (p_cp->th - 1U) * p_cp->tdy; /* can't overflow */
```

Целочисленное переполнение умножения в Poppler:

```
bitmap = readGenericBitmap(mmr, (grayMax + 1) * patternW, patternH,  
    templ, false, false, nullptr, atx, aty, length - 7);
```

Целочисленное переполнение в Rizin, приводящее к выходу за границы массива:

```
symbols_size = (symbols_count + 1) * 2 * sizeof(struct symbol_t);  
if (symbols_size < 1) {  
    ht_pp_free(hash);  
    return NULL;  
}  
if (!(symbols = calloc(1, symbols_size))) {  
    ht_pp_free(hash);  
    return NULL;  
}  
...  
symbols[j].last = true;
```

Заключение

1. Предложенный алгоритм слайсинга предиката пути, который устраняет избыточные ограничения, полученные в результате динамической символьной интерпретации бинарного кода. Для алгоритма доказаны теоремы о его конечности, корректности, и проведена оценка его вычислительной сложности.
2. Разработанный метод построения предикатов безопасности для ошибок деления на ноль, выхода за границу массива и целочисленного переполнения во время динамической символьной интерпретации пути выполнения программы.
3. Разработанный метод автоматизированного поиска ошибок при помощи символьных предикатов безопасности.

- Разработаны методы обнаружения ошибок в больших программных системах в контексте гибридного фаззинга.
- Разработанные методы позволяют генерировать входные данные для подтверждения дефектов.
- Автоматически выделяются истинно положительные срабатывания символьных предикатов безопасности.
- Предложен алгоритм слайсинга предиката пути, устраняющий избыточные ограничения, и формально исследованы его свойства.
- Предложенные методы встраиваются в системы непрерывной интеграции (CI) в рамках следования безопасному циклу разработки ПО (SDL).
- Методы позволили обнаружить 17 новых ошибок в 10 различных проектах с открытым исходным кодом.
- Разработанные методы используются в Центре доверенного искусственного интеллекта ИСП РАН.

Свидетельства о регистрации программ для ЭВМ

РОССИЙСКАЯ ФЕДЕРАЦИЯ



СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2020662214

«Инструмент динамической символической интерпретации
«Sydr»

Правообладатель: *Федеральное государственное бюджетное учреждение науки Институт системного программирования им. В.П. Иванникова Российской академии наук (RU)*

Авторы: *Вишняков Алексей Вадимович (RU), Федотов Андрей Николаевич (RU), Куц Даниил Олегович (RU), Новиков Александр Андреевич (RU), Курмангалеев Шамиль Фаимович (RU)*

Заявка № 2020661261

Дата поступления 30 сентября 2020 г.

Дата государственной регистрации

в Реестре программ для ЭВМ 09 октября 2020 г.

Руководитель Федеральной службы
по интеллектуальной собственности

 Г.П. Иванова



РОССИЙСКАЯ ФЕДЕРАЦИЯ



СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2021665874

«Sydr-fuzz»

Правообладатель: *Федеральное государственное бюджетное учреждение науки Институт системного программирования им. В.П. Иванникова Российской академии наук (RU)*

Авторы: *Федотов Андрей Николаевич (RU), Вишняков Алексей Вадимович (RU), Куц Даниил Олегович (RU)*


Заявка № 2021664867

Дата поступления 24 сентября 2021 г.

Дата государственной регистрации

в Реестре программ для ЭВМ 04 октября 2021 г.

Руководитель Федеральной службы
по интеллектуальной собственности

 Г.П. Иванова



1. Оценка критичности программных дефектов в условиях работы современных защитных механизмов. / — А. Н. Федотов, В. А. Падарян, В. В. Каушан, Ш. Ф. Курмангалеев, А. В. Вишняков, А. Р. Нурмухаметов // Труды института системного программирования РАН. — 2016.
2. *Вишняков, А. В.* — Поиск ошибок в бинарном коде методами динамической символьной интерпретации. / — А. В. Вишняков, И. А. Кобрин, А. Н. Федотов // Труды института системного программирования РАН. — 2022.
3. Sydr: Cutting Edge Dynamic Symbolic Execution. / — A. Vishnyakov [et al.] // 2020 Ivannikov ISPRAS Open Conference (ISPRAS). — 2020.
4. Symbolic Security Predicates: Hunt Program Weaknesses. / — A. Vishnyakov [et al.] // 2021 Ivannikov ISPRAS Open Conference (ISPRAS). — 2021.
5. *Вишняков, А. В.* — Символьные предикаты безопасности в гибридном фаззинге. / — А. В. Вишняков, И. А. Кобрин, А. Н. Федотов // Материалы 31-й научно-технической конференции «Методы и технические средства обеспечения безопасности информации» (МитСОБИ). — 2022.
6. *Кобрин, И. А.* — Гибридный фаззинг фреймворка машинного обучения TensorFlow. / — И. А. Кобрин, А. В. Вишняков, А. Н. Федотов // Материалы 31-й научно-технической конференции «Методы и технические средства обеспечения безопасности информации» (МитСОБИ). — 2022.

1. Открытая конференция ИСП РАН имени В.П. Иванникова. Москва. 11 декабря 2020.
2. Открытая конференция ИСП РАН имени В.П. Иванникова. Москва. 3 декабря 2021.
3. 31-я научно-техническая конференция «Методы и технические средства обеспечения безопасности информации» памяти П.Д. Зегжды (МитСОБИ). Санкт-Петербург. 28–29 июня 2022.

Основные результаты работы

1. Разработан алгоритм слайсинга предиката пути, устраняющий избыточные ограничения. Доказаны теоремы о его конечности и корректности, произведена оценка асимптотической сложности. Экспериментальная оценка эффективности алгоритма показала повышение скорости и точности порождения входных данных.
2. Разработан метод построения предикатов безопасности для обнаружения ошибок деления на нуль, выхода за границу массива и целочисленного переполнения при помощи динамической символьной интерпретации. Экспериментальная оценка метода на наборе тестов Juliet показала общую точность **95.59 %** для 11 классов ошибок CWE (15772 теста).
3. Разработан метод автоматизированного поиска ошибок при помощи символьных предикатов безопасности после гибридного фаззинга. Обнаружено **17** новых ошибок в 10 различных проектах с открытым исходным кодом.
4. Методы реализованы в программных системах и используются в Центре доверенного искусственного интеллекта ИСП РАН.

Спасибо за внимание!

1. Не раскрыт вопрос обработки ошибочных ситуаций при моделировании семантики функций — целью данной работы является поиск ошибок непосредственно в разрабатываемом коде, для поиска ошибок в стандартных библиотечных функциях можно отключить моделирование семантики функций, более того, упрощение формул повышает вероятность обнаружения ошибок.
2. Не указано время построения предикатов безопасности и обнаружения ошибок — время построения предикатов безопасности на порядки меньше времени решения запросов математическим решателем, а время поиска ошибок в 15772 тестах Juliet в 32 потока занимает 12 минут.

Не рассмотрен вопрос об ограничениях по типам ошибок, к которым можно применять разработанные методы:

- На данный момент находятся ошибки деления на нуль, разыменованного нулевого указателя, целочисленного переполнения, выхода за границу массива: CWE 118–129, 131, 190, 191, 193–195, 369, 680.
- В дальнейшем планируется расширить методы для поиска ошибок целочисленного усечения, форматной строки, инъекции команд.

Ответы на замечания оф. оппонента Маркина Д. О.

1. Не приводится анализ и оценка достаточности и избыточности ограничений предиката пути — производилась экспериментальная оценка точности предикатов безопасности на тестах Juliet и были найдены новые ошибки при помощи метода, однако, к сожалению, формальная оценка не приводится в тексте диссертации.
2. Производительность DSE не имеет решающего значения для поиска уязвимостей — производительность DSE непосредственно влияет на помощь открытия новых путей фаззеру.
3. Замена функции на ее модель может снижать общую результативность поиска дефектов — упрощение формул повышает вероятность обнаружения ошибок, а также необходимо для обнаружения ошибок некорректного использования функций (например, переполнение в `memcpy`).
4. Область применения алгоритма слайсинга предиката пути — алгоритм применим к программам любого объема, сложности и процессорной архитектуры.

Пример слайсинга предиката пути

```
1 char* syms = "SLICING FIX IT!\n";
2 // b - входные данные.
3 int len = strlen(syms);
4 if (b[0] < len)
5     if (syms[b[0] % len] == '!')
6         if (b[2] > '@')
7             if (b[5] + b[4] < 'B')
8                 if (b[3] + b[5] > '@')
9                     if (b[1] + b[3] > '@')
10                        if (b[4] < '9')
11                            if (b[1] > '@')
12                                // Целевой переход.
13                                printf("OK\n");
14                            else
15                                // Изначальный путь.
16                                printf("FAIL\n");
```

Переменная `b[0]` на строке 5
недостаточно помечена.

Пример слайсинга предиката пути

```
1 char* syms = "SLICING FIX IT!\n";
2 // b - входные данные.
3 int len = strlen(syms);
4 if (b[0] < len)
5     if (syms[b[0] % len] == '!')
6         if (b[2] > '@')
7             if (b[5] + b[4] < 'B')
8                 if (b[3] + b[5] > '@')
9                     if (b[1] + b[3] > '@')
10                        if (b[4] < '9')
11                            if (b[1] > '@')
12                                // Целевой переход.
13                                printf("OK\n");
14                            else
15                                // Изначальный путь.
16                                printf("FAIL\n");
```

Переменная `b[0]` на строке 5
недостаточно помечена.

Строка	Переменные слайсинга
11	<code>b[1]</code>

Пример слайсинга предиката пути

```
1 char* syms = "SLICING FIX IT!\n";
2 // b - входные данные.
3 int len = strlen(syms);
4 if (b[0] < len)
5     if (syms[b[0] % len] == '!')
6         if (b[2] > '@')
7             if (b[5] + b[4] < 'B')
8                 if (b[3] + b[5] > '@')
9                     if (b[1] + b[3] > '@')
10                        if (b[4] < '9')
11                            if (b[1] > '@')
12                                // Целевой переход.
13                                printf("OK\n");
14                            else
15                                // Изначальный путь.
16                                printf("FAIL\n");
```

Переменная `b[0]` на строке 5
недостаточно помечена.

Строка	Переменные слайсинга
11	<code>b[1]</code>
9	<code>b[1], b[3]</code>

Пример слайсинга предиката пути

```
1 char* syms = "SLICING FIX IT!\n";
2 // b - входные данные.
3 int len = strlen(syms);
4 if (b[0] < len)
5     if (syms[b[0] % len] == '!')
6         if (b[2] > '@')
7             if (b[5] + b[4] < 'B')
8                 if (b[3] + b[5] > '@')
9                     if (b[1] + b[3] > '@')
10                        if (b[4] < '9')
11                            if (b[1] > '@')
12                                // Целевой переход.
13                                printf("OK\n");
14                            else
15                                // Изначальный путь.
16                                printf("FAIL\n");
```

Переменная `b[0]` на строке 5
недостаточно помечена.

Строка	Переменные слайсинга
11	<code>b[1]</code>
9	<code>b[1], b[3]</code>
8	<code>b[1], b[3], b[5]</code>

Пример слайсинга предиката пути

```
1 char* syms = "SLICING FIX IT!\n";
2 // b - входные данные.
3 int len = strlen(syms);
4 if (b[0] < len)
5     if (syms[b[0] % len] == '!')
6         if (b[2] > '@')
7             if (b[5] + b[4] < 'B')
8                 if (b[3] + b[5] > '@')
9                     if (b[1] + b[3] > '@')
10                        if (b[4] < '9')
11                            if (b[1] > '@')
12                                // Целевой переход.
13                                printf("OK\n");
14                            else
15                                // Изначальный путь.
16                                printf("FAIL\n");
```

Переменная `b[0]` на строке 5
недостаточно помечена.

Строка	Переменные слайсинга
11	<code>b[1]</code>
9	<code>b[1], b[3]</code>
8	<code>b[1], b[3], b[5]</code>
7	<code>b[1], b[3], b[4], b[5]</code>

Пример слайсинга предиката пути

```
1 char* syms = "SLICING FIX IT!\n";
2 // b - входные данные.
3 int len = strlen(syms);
4 if (b[0] < len)
5     if (syms[b[0] % len] == '!')
6         if (b[2] > '@')
7             if (b[5] + b[4] < 'B')
8                 if (b[3] + b[5] > '@')
9                     if (b[1] + b[3] > '@')
10                        if (b[4] < '9')
11                            if (b[1] > '@')
12                                // Целевой переход.
13                                printf("OK\n");
14                            else
15                                // Изначальный путь.
16                                printf("FAIL\n");
```

Переменная `b[0]` на строке 5
недостаточно помечена.

Строка	Переменные слайсинга
11	<code>b[1]</code>
9	<code>b[1], b[3]</code>
8	<code>b[1], b[3], b[5]</code>
7	<code>b[1], b[3], b[4], b[5]</code>
10	<code>b[1], b[3], b[4], b[5]</code>

Метод моделирования семантики функций

- Пропуск символьной интерпретации тела функций.
- Описание эффектов выполнения символьными формулами.
- Объединяются несколько символьных состояний, что позволяет исследовать больше путей программы.
- Набор функций для моделирования получен путем анализа результатов динамической символьной интерпретации различных Linux программ.
- Промоделирована 41 функция стандартной библиотеки: они обрабатывают символьные данные.

Разработанные модели функций

- Пропуск интерпретации функции и конкретизация ее выполнения.
 - Динамическое выделение памяти (`*alloc, free`).
 - Печать в стандартный поток вывода (`printf, std::cout`).
 - Ускоряет символьную интерпретацию и уменьшает избыточную помеченность.
- Копирование данных (`str*cpy, memcpy`).
- Поиск символа или подстроки в строке (`memchr, strstr`).
- Лексикографическое сравнение строк (`memcmp, strcmp`).
- Преобразование строки в число (`strto*l, atoi`).
 - Функция `strto*l` вызывается внутри `scanf("%d", &x)`.
 - Тесты Juliet читают входные числа с помощью `scanf`.

Поиск символа в строке

`memchr(ptr, ch, count):`

$$ite\left(\bigwedge_{k=0}^{count-1} ptr[k] \neq ch, 0, ptr + \sum_{i=0}^{count-1} ite\left(\bigwedge_{k=0}^i ptr[k] \neq ch, 1, 0\right)\right)$$

Сравнение построения предиката пути

Приложение	Без моделирования		С моделированием	
	Ветвления	Время	Ветвления	Время
bzip2recover	5131	6s	5131	6s
cjpeg	8008	19s	6992	18s
faad	470585	21m	466697	15m52s
foo2lava	910737	21m9s	905592	18m20s
hdp	66070	43s	29265	20s
jasper	837643	14m47s	771806	10m37s
libxml2	53400	40s	8873	12s
minigzip	8977	1m4s	8977	1m3s
muraster	7102	5s	4453	4s
pk2bm	3665	2s	658	1s
pnmhistmap_pgm	967187	9m21s	967155	9m2s
pnmhistmap_ppm	7864	12s	7822	11s
readelf	62713	41s	13649	10s
yices-smt2	19352	17s	10340	11s
yodl	8329	9s	5340	5s

Сравнение результатов инвертирования переходов за 2 часа

Приложение	Без моделирования				С моделированием			
	Точность	SAT	Запр.	Время	Точность	SAT	Запр.	Время
bzip2recover	100 %	2101	5131	47m35s	100 %	2101	5131	45m38s
cjpeg	100 %	50	2656	120m	100 %	50	3750	120m
faad	97.11 %	1974	3072	120m	98.91 %	1560	2414	120m
foo2lava	87.1 %	31	5998	120m	99.02 %	205	6668	120m
hdp	76.69 %	1171	4122	120m	72.22 %	5893	12172	120m
jasper	99.62 %	8457	22538	120m	96.61 %	9528	24472	120m
libxml2	51.27 %	1063	18485	120m	82.44 %	1247	8970	5m53s
minigzip	51.47 %	7569	8977	16m16s	51.47 %	7569	8977	16m16s
muraster	99.94 %	3304	6041	120m	100 %	360	470	120m
pk2bm	99.45 %	183	3664	15m55s	100 %	189	657	4m55s
pnmhistmap_pgm	99.99 %	19351	28932	120m	100 %	19964	29369	120m
pnmhistmap_ppm	99.07 %	107	7990	27m26s	99.12 %	114	7948	25m31s
readelf	87.38 %	1022	9541	120m	85.82 %	2363	6541	120m
yices-smt2	73.79 %	4258	16222	120m	70.27 %	5534	11753	11m5s
yodl	36.25 %	1153	9403	51m3s	98.26 %	1150	6414	1m50s

Целочисленное переполнение, приводящее к переполнению буфера

- 32-битная программа
- Ввод: +00000000002
- strtol на строке 6
- Целочисленное переполнение на строке 9
- Переполнение буфера на строке 12
- Решение:
+01073741825

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int size;
6      fscanf(stdin, "%d", &size);
7      if (size <= 0) return 1;
8      size_t i;
9      int *p = malloc(size * sizeof(int));
10     if (p == NULL) return 1;
11     for (i = 0; i < (size_t)size; i++) {
12         p[i] = 0;
13     }
14     printf("%d\n", p[0]);
15     free(p);
16 }
```

Схема инструмента Sydr (Symbolic DynamoRIO)

