DOI: 10.15514/ISPRAS-2023-35(6)-7

Обнаружение возможной перезаписи переменных вследствие использования функций нелокальных переходов

[1, 2] *Н.Ю. Шугалей,* ORCID: 0009-0000-9310-8317, <*shugaley@ispras.ru*>
[1] *В.А. Иванишин,* ORCID: 0000-0002-9784-2911, <*vlad@ispras.ru*>
[1] *А.В. Монаков,* ORCID: 0000-0001-5118-0054, <*amonakov@ispras.ru*>

[1] *Институт системного программирования РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.
[2] *Московский физико-технический институт (национальный исследовательский университет), 117303, г. Москва, ул. Керченская, д.1 А, корп. 1.*

Аннотация. Причиной возникновения неопределенного поведения является исходный код, написанный с нарушением стандарта языка Си. Неопределенное поведение приводит к появлению уязвимостей в программном обеспечении. Одним из распространенных источников неопределенного поведения является некорректное использование функций нелокальных переходов (в частности setjmp и longjmp). В данной работе рассмотрены средства обнаружения такого типа неопределенного поведения, реализованные в основных современных компиляторах (GCC, Clang, MSVC). Сделаны выводы о том, что эти средства обладают существенными недостатками либо вовсе отсутствуют в отдельных компиляторах. Описана реализация нового метода компиляторной диагностики рассматриваемого неопределенного поведения. Приведенный в работе метод обладает точностью, достаточной для практического применения на реальных проектах. Рассмотрены преимущества представленного решения над похожими существующими.

Ключевые слова: компиляторы; си; нелокальные переходы; setjmp; volatile; неопределенное поведение; статический анализ.

Для цитирования: Шугалей Н.Ю., Иванишин В.А., Монаков А.В. Обнаружение возможной перезаписи переменных вследствие использования функций нелокальных переходов. Труды ИСП РАН, том 35, вып. 6, 2023 г., стр. 121-134. DOI: 10.15514/ISPRAS–2023–35(6)-7.

Detecting Potentially Clobbered Variables due to the Use of Nonlocal Jumps Functions

[1, 2] *N.U. Shugaley,* ORCID: 0009-0000-9310-8317, <*shugaley@ispras.ru*>
[1] *V.A. Ivanishin,* ORCID: 0000-0002-9784-2911, <*vlad@ispras.ru*>
[1] *A.V. Monakov,* ORCID: 0000-0001-5118-0054, <*amonakov@ispras.ru*>

[1] *Institute for System Programming of the Russian Academy of Sciences,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
[2] *Moscow Institute of Physics and Technology (National Research University),*
1 A, Kerchenskaya st., Moscow, 117303, Russia.

Abstract. The reason of undefined behavior is source code written in violation of the C language standard. Undefined behavior leads to vulnerabilities in software. One of the common sources of undefined behavior is an incorrect use of functions for nonlocal jumps (in particular setjmp and longjmp). This paper considers the

means of detecting this type of undefined behavior which are implemented in the major modern compilers (GCC, Clang, MSVC). We conclude that these means either have significant disadvantages or are absent in some compilers. This paper presents the implementation of a new method of compiler warning of the considered undefined behavior. The described method is accurate enough for practical application on real projects. We consider the advantages of the proposed solution over similar existing ones.

Keywords: compilers; c; nonlocal jumps; setjmp; volatile; undefined behavior; static analysis.

For citation: Shugaley N.U., Ivanishin V.A., Monakov A.V. Detecting potentially clobbered variables due to the use of nonlocal jumps functions. Trudy ISP RAN/Proc. ISP RAS, vol. 35, issue 6, 2023. pp. 121-134 (in Russian). DOI: 10.15514/ISPRAS-2023-35(6)-7.

1. Введение

Механизм работы функций семейства `setjmp`

Функция `setjmp` обладает свойством двойного возврата. При первом непосредственном вызове `setjmp` сохраняет информацию о текущем состоянии вызова (значения регистров) в специальный буфер с типом `jmp_buf` (в дальнейшем сам буфер будем также обозначать как `jmp_buf`). Таким образом, `setjmp` динамически устанавливает точку, в которую позже функция `longjmp` передает управление. После прямого вызова `setjmp` происходит нормальный возврат из него с нулевым возвращаемым значением. После этого `jmp_buf` передается в `longjmp` для осуществления нелокального перехода и восстановления исходного состояния. Происходит возврат не из `longjmp`, а второй возврат из `setjmp` – анормальный (англ. abnormal) возврат с ненулевым возвращаемым значением.

Пара функций `sigsetjmp` / `siglongjmp` аналогична по поведению паре `setjmp` / `longjmp` за исключением того, что предоставляет предсказуемую обработку маски сигналов процесса. Реализовано это с помощью отдельного передаваемого аргумента, который явно отвечает за сохранение маски сигналов. Для дальнейшего повествования данное различие не имеет значения, поэтому при упоминании пары `setjmp` / `longjmp` подразумевается также и пара с префиксом `sig`.

Неопределенное поведение вследствие перетирания переменных

Компилятор может оптимизировать доступ к переменным, размещая их на регистры. А `longjmp` восстанавливает значения регистров вместе с указателем на стек и счетчиком команд. Следовательно, при выполнении некоторых условий значение переменной с автоматическим классом памяти после `longjmp` будет иметь неопределенное значение. Это значение зависит от размещения компилятором переменной в ходе оптимизаций. Условия возникновения неопределенного значения после вызова `longjmp` [1]:

- Переменная локальна для функции, внутри которой происходит непосредственный вызов `setjmp`;
- Ее значение изменяется между вызовами `setjmp` и `longjmp`;
- Она не объявлена как `volatile`.

Если соблюдены эти условия и неопределенное значение переменной читается после вызова `longjmp`, то мы имеем неопределенное поведение.

Наиболее простой способ устранить данное неопределенное поведение без изменения общей логики программы – объявить такую переменную как `volatile`. Дело в том, что компилятор гарантирует не оптимизировать доступ к `volatile` переменной, то есть каждое чтение и запись такой переменной будут выполняться непосредственно из памяти, а не из регистра процессора. Таким образом, перетирание регистров перестает влиять на значение переменной. Значение переменной будет сохраняться после вызова `longjmp`.

2. Постановка задачи

Актуальность работы

Функции `setjmp` / `longjmp` могут использоваться для:

- Обработки ошибок в глубоко вложенных функциях – аналог исключений из C++. Используется в исходном коде на языке Си, когда C++ не применим: встроенные системы с жесткими ограничениями по памяти [2], старый программный код. Является наиболее распространенным механизмом применения.
- Передачи управления от обработчиков сигналов в определенную точку программы. Не рекомендуется, поскольку может привести к неопределенному поведению в результате использования non-async-signal-safe функций [3].
- Реализация сопрограмм [4].

В дистрибутиве Alpine Linux 3.16 данные функции встречаются в исходном коде около 800 пакетов программ из репозиториев main и community. Всего эти два репозитория содержат 6250 пакетов.

Постановка задачи

Целью данной работы является разработка метода обнаружения случаев неопределенного поведения в языке Си при использовании функций нелокальных переходов (англ. nonlocal jumps или nonlocal gotos), в частности `setjmp` и `longjmp`. Рассматривается только описанное выше неопределенное поведение, связанное с перезаписью переменных.

Данная работа нацелена на реализацию предупреждения в компиляторе, а не в полноценном статическом анализаторе. Это накладывает некоторые ограничения на возможности диагностики. Если функции определены в разных единицах трансляции, то в общем случае компилятор не может проанализировать определение вызываемой функции в точке ее вызова. Искомый метод диагностики должен быть полезным, даже когда применение межпроцедурного анализа невозможно.

Диагностика должна быть применима для реальных программ, написанных с использованием функций нелокальных переходов. Общее число срабатываний должно быть реальным для ручного рассмотрения и исправления, а количество ложно-положительных срабатываний должно быть минимизировано. Для этого на основе анализа реального программного кода должны быть выделены основные шаблоны использования `setjmp`, а на их основе сформулированы и реализованы необходимые эвристики.

Данная задача включает в себя следующие подзадачи:

- Анализ существующих решений и выявление их недостатков.
- Разработка более точного алгоритма и эвристик, повышающих точность диагностики.
- Реализация этого алгоритма в Безопасном компиляторе [5], разрабатываемом в ИСП РАН на основе GCC (опция `-Wclobbered`).
- Тестирование реализации сборкой популярного дистрибутива ОС Linux.

3. Обзор существующих решений

Существующие решения

В GCC с версии 4.3.0 (2008 г.) добавлена опция `-Wclobbered` (включена в `-Wextra`). Опция включает компиляторную диагностику рассматриваемого неопределенного поведения. Данная диагностика реализована на промежуточном представлении RTL [6].

В компиляторах Clang и MSVC данная диагностика отсутствует.

Недостатки существующих решений

Основной недостаток реализации на RTL – зависимость от размещения переменной компилятором. Разные компиляторы, в том числе и разные версии одного компилятора, могут как помещать переменную на регистр в ходе оптимизаций, так и сохранять ее непосредственно в памяти. Во втором случае, анализируя RTL, мы лишаемся возможности диагностики неопределенного поведения. Перенос реализации с RTL на GIMPLE [7], [8] позволил бы обнаруживать потенциальные проблемы без привязки к результатам оптимизации переменной с автоматическим классом памяти. Такой перенос устраняет рассмотренную зависимость от текущего процесса компиляции конкретным компилятором.

Кроме того, реализация на GIMPLE позволит анализировать возвращаемое значение `setjmp` и различать нормальный и анормальный возвраты из `setjmp`. Благодаря этому можно подавить распространенное в GCC ложно-положительное срабатывание со следующим сценарием:

1) Вызов `setjmp`;
2) Запись в локальную не `volatile` переменную;
3) Вызов `longjmp` с соответствующим `jmp_buf`;
4) Запись в эту переменную (или вовсе отсутствие дальнейшего использования переменной).

Очевидно, что данный сценарий не приводит к неопределенному поведению, так как неопределенное значений переменной нигде не используется.

Таким образом, существующая в GCC реализация диагностики на RTL является малоприменимой для реальных проектов ввиду приведенных выше фактов, следствием которых являются многочисленные ложно-отрицательные и ложно-положительные срабатывания [9].

4. Решение: новый алгоритм на GIMPLE

4.1 Детали реализации функций нелокального перехода в GIMPLE

В GIMPLE нелокальные переходы моделируются с помощью функции `abnormal_dispatcher`. Ее вызов происходит в отдельном базовом блоке, который обозначим как B_{ab}. Пользовательская функция всегда обладает единственным B_{ab} (либо его вообще нет, если функция не вызывает `setjmp`). С остальными базовыми блоками B_{ab} соединен специальными анормальными ребрами:

- Исходящими в базовые блоки с вызовом `setjmp` (обозначим такой блок как B_{sj});
- Входящими от базовых блоков с вызовом любой другой функции.

На рис. 1 показан пример ГПУ [10] с B_{ab}. На этом и последующих примерах ГПУ имеет упрощенный схематичный вид. Обозначения ребер здесь и далее:

- Пунктирные линии – анормальные ребра;
- Сплошные линии – нормальные (все остальные) ребра.

Непосредственно перед вызовом `setjmp` для каждой живой локальной не `volatile` переменной компилятор создает phi-функцию. Такую phi-функцию кратко обозначим как `phi0`.

На рис. 2:

- B_{ab} находится в блоке <bb 5>;
- B_{sj} находится в блоке <bb 2> (в нем же `phi0` переменной `i`).

4.2 Фундаментальное ограничение алгоритма на GIMPLE

Фундаментальное ограничение диагностики на GIMPLE – зависимость от наличия phi-функции `phi0`. В случае отсутствия `phi0` мы имеем потенциальное ложно-отрицательное срабатывание. В частности, `phi0` не обладают регистровые ассемблерные переменные из-за их текущей реализации в GCC.

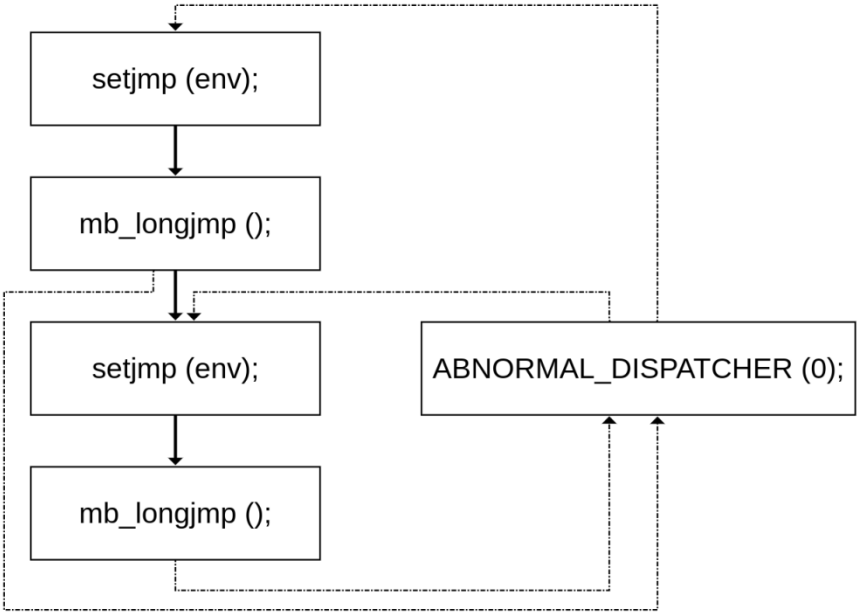

Рис. 1. Пример ГПУ с abnormal_dispatcher
Fig. 1. Example of CFG with abnormal_dispatcher

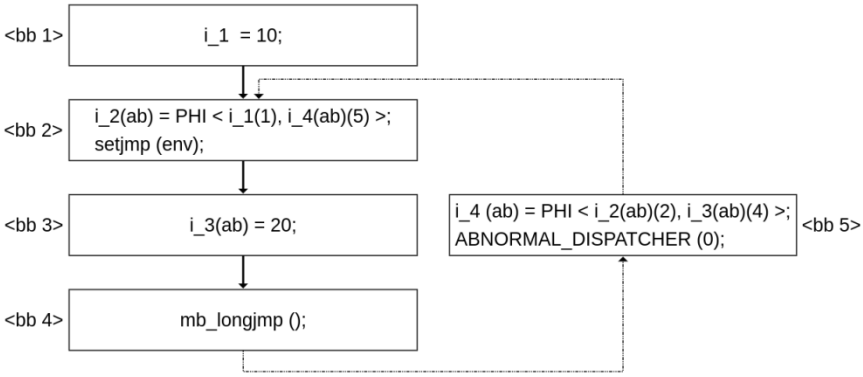

Рис. 2. Пример ГПУ с phi0
Fig. 2. Example of CFG with phi0

В теории также источником ложно-отрицательных срабатываний могут быть поля структур, которые GCC SRA [11] не скаляризирует (не заменяет на локальные переменные), и, как следствие, они не имеют `phi0`, однако другой компилятор может их скаляризировать.

4.3 Базовый алгоритм

Краткий базовый алгоритм новой реализации диагностики `-Wclobbered` на GIMPLE:

1) Поиск B_{ab}. Если у анализируемой функции нет B_{ab}, то диагностика завершается.
2) Вычисление M_{ab} – множества базовых блоков, достигающих B_{ab} (кроме B_{sj}). Здесь и далее достижимость определяется с помощью алгоритма DFS [12]. На данном шаге анормальные ребра считаются проходимы в ходе DFS обхода, на остальных – нет.
3) Вычисление M_{sj} – множества базовых блоков, достижимых из B_{sj}.
4) Вычисление $M_{sj,\,a}$ – множества базовых блоков, достижимых из B_{sj} и совместимых с анормальным возвратом (производится анализ предиката возвращаемого значения `setjmp`).
5) Вычисление M_{int} – пересечения M_{ab} и M_{sj}. То есть M_{int} является множеством базовых блоков между `setjmp` и `abnormal_dispatcher`.
6) Поиск в M_{int} переопределения каждой переменной, обладающей `phi0`. Реализовано это посредством рекурсивного обхода аргументов `phi0`. Если в M_{int} обнаружены константа или явное определение аргумента, то искомое переопределение найдено. Если нет, то диагностика для данной переменной завершается.
7) Поиск использований `phi0` в $M_{sj,\,a}$. Если такое использование найдено, то на переопределение из шага 6 выдается срабатывание как на источник потенциального неопределенного поведения.

Таким образом, весь базовый алгоритм можно обобщить тремя следующими пунктами:

1) Шаг 1: поиск B_{ab}. Наличие B_{ab} – условие запуска алгоритма.
2) Шаги 2-4, 6: поиск переопределения переменной, обладающей `phi0`, между `setjmp` и `abnormal_dispatcher`.
3) Шаги 5, 7: поиск использования `phi0` после анормального возврата.

Если найдено переопределение и использование, то компилятор выдает срабатывание.
Пример поиска переопределения (рис. 3):

- M_{ab} – visited справа;
- M_{sj} – visited слева;
- M_{int} – visited с двух сторон (в <bb 3> искомое на шаге 6 переопределение).

Пример поиска использования (рис. 4):

- $M_{sj,\,a}$ – visited;
- Нет искомого на шаге 7 использования `phi0`, достижимого через $M_{sj,\,a}$.

4.4 Аналогия с алгоритмом анализа циклов

Можно провести некоторую аналогию между базовым алгоритмом и алгоритмом анализа циклов [13]. Рассматриваются все циклы, которые:

- Начинаются с B_{sj};

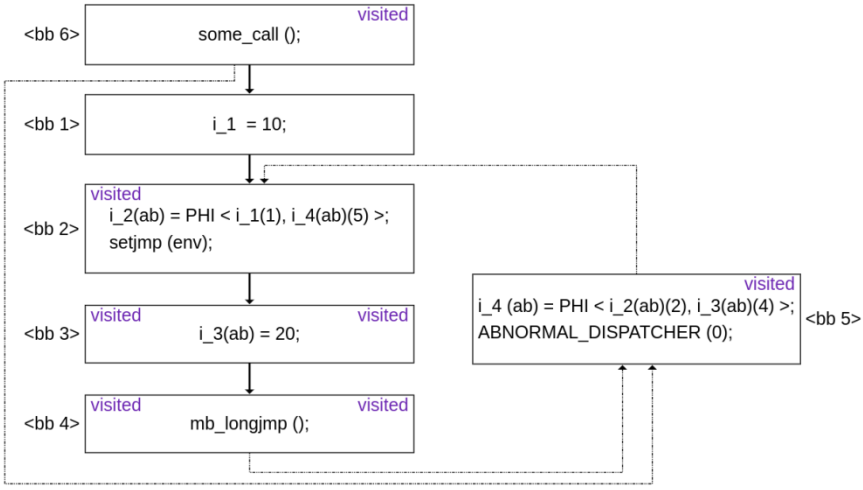

Рис. 3. Поиск переопределения
Fig. 3. Search of redefinition

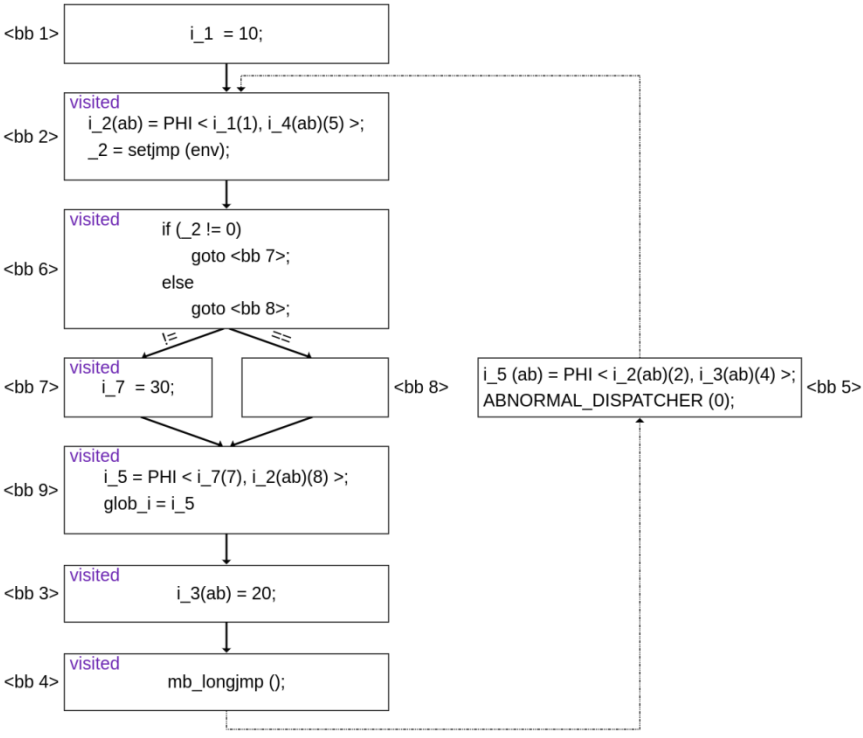

Рис. 4. Поиск использования
Fig. 4. Search of usage

- Включают в себя B_{ab} по произвольному пути;
- Возвращаются в B_{sj}.

Происходит поиск переменных, которые являются живыми на входе и изменяются в таком цикле. Полученный в результате поиска набор кандидатов на срабатывание фильтруется: остаются лишь те определения из цикла, которые в конечном итоге используются после анормального возврата из `setjmp`.

4.5 Эвристика

Данный алгоритм обеспечивает широкое покрытие на исходном коде программ, однако никак не считается с реальным практическим применением `setjmp` программистами.

Рассмотрим следующие мотивационные примеры (см Листинг 1 и Листинг 2).

```
if (setjmp (...) == 0)
   foo ();
else
   handle_longjmp ();
bar ();
```

Листинг 1. Первый шаблон использования в реальном коде
Listing 1. The first pattern of using in real code

```
if (setjmp (...) != 0)
   handle_longjmp ();
bar ();
```

Листинг 2. Второй шаблон использования в реальном коде
Listing 2. The second pattern of using in real code

Принципиальное различие между двумя этими шаблонами:

- Листинг 1: есть отдельный условный блок кода, выполняющийся только после нормального возврата;
- Листинг 2: нет такого условного блока.

На основе анализа исходного кода реальных программ были выделены именно такие шаблоны использования `setjmp`. Сформулируем следующую эвристику:

- Вызов `longjmp` не ожидается из условного блока, выполняющегося только после анормального возврата. Обычно программист не ожидает еще один `longjmp` (по крайней мере, с тем же `jmp_buf`) из кода, отвечающего за обработку предыдущего `longjmp`.
- Вызов `longjmp` ожидается из условного блока, выполняющегося только после нормального возврата, если он есть (листинг 1). При наличии такого выделенного кода программист ожидает `longjmp` именно из него.

В приведенных примерах в базовом алгоритме без эвристики `longjmp` ожидался из любого вызова, так как на шаге 6 переопределение искалось в M_{int}. На шаге 5 M_{int} строится на основе M_{sj}, содержащего все достижимые из B_{sj} блоки. С эвристикой же:

- Листинг 1: `longjmp` ожидается из вызова `foo` (реализуются оба пункта эвристики);
- Листинг 2: `longjmp` ожидается из вызова `bar` (реализуется только первый пункт эвристики, так как условного блока нормального возврата нет).

На рис. 5 и рис. 6 жирным шрифтом выделены базовые блоки, из которых, согласно эвристике, ожидается `longjmp`.

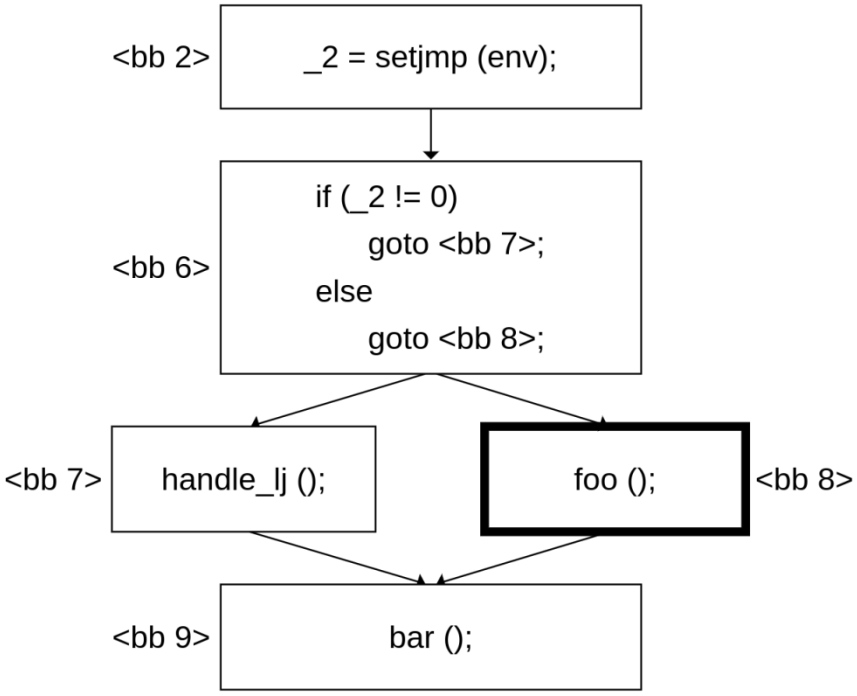

Рис. 5. Эвристика для листинга 1
Fig. 5. Heuristic for listing 1

Таким образом, суть эвристики – определить базовые блоки, где `longjmp` на практике ожидается, то есть сузить множество M_{int}. Для внедрения эвристики модифицируются шаги 4 и 5 базового алгоритма.

Шаг 4 разбивается на два подпункта:

- Вычисление $M_{\text{sj, a}}$ – множества базовых блоков, достижимых из B_{sj} и совместимых с анормальным возвратом (шаг 4 из базового алгоритма).
- Вычисление $M_{\text{sj, n}}$ – множества базовых блоков, достижимых из B_{sj} и совместимых с нормальным возвратом.

На шаге 5, помимо непосредственно вычисления M_{int}, оно также сужается (уточняется с учетом эвристики) до $M_{\text{int, h}}$. Для вычисления $M_{\text{int, h}}$ к шагу 5 добавляются следующие подпункты, соответствующие одному из шаблонов из листингов 1 и 2:

- Если $M_{\text{sj, n}}$ не насыщенное подмножество $M_{\text{sj, a}}$, то $M_{\text{int, h}}$ есть пересечение M_{int} и разности $M_{\text{sj, n}}$ и $M_{\text{sj, a}}$. Соответствует шаблону из листинга 1.
- Если $M_{\text{sj, n}}$ насыщенное подмножество $M_{\text{sj, a}}$, то $M_{\text{int, h}}$ есть пересечение M_{int} и $M_{\text{sj, n}}$. Соответствует шаблону из листинга 2.
- Под понятием насыщенного подможество здесь подразумевается: множество базовых блоков A является насыщенным подмножеством множества базовых блоков B, если разность A и B является пустым множеством или содержит только пустые базовые блоки.
- На шаге 6 (поиск переопределения) вместо M_{int} используется новое $M_{\text{int, h}}$.

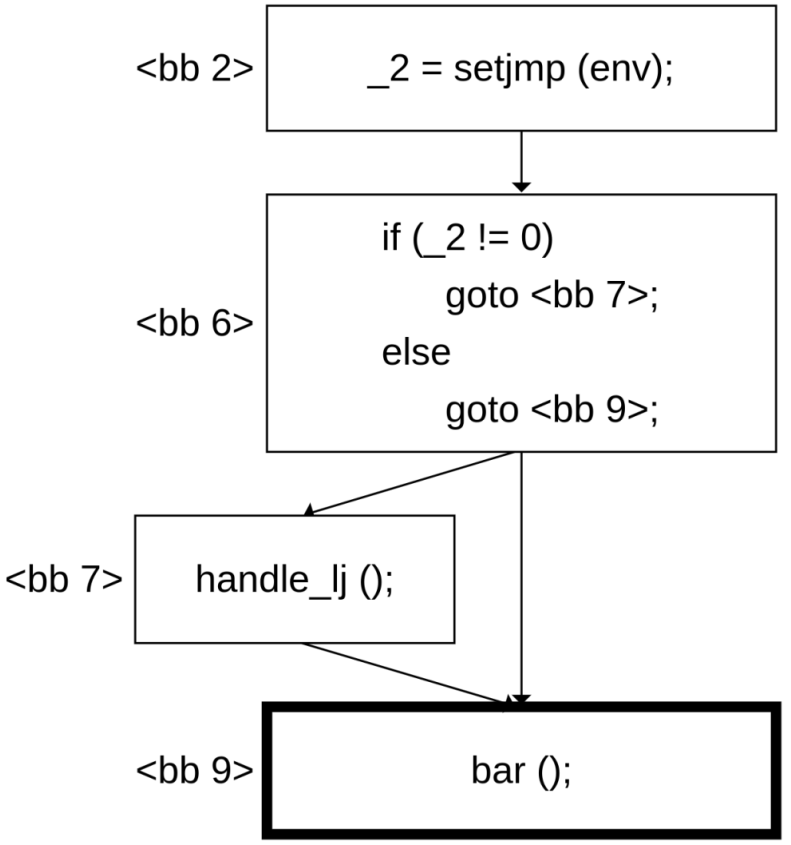

Рис. 6. Эвристика для листинга 2
Fig. 6. Heuristic for listing 2

4.6 Дополнительное уточнение алгоритма с эвристикой

С эвристикой может возникнуть ситуация, когда найденное на шаге 6 переопределение окажется присваиванием возвращаемого значения функции, внутри которой ожидается потенциальный `longjmp`. Данная ситуация может спровоцировать ложно-положительное срабатывание. Рассмотрим следующее выражение:

```
var = mb_longjmp ();
```

Листинг 3. Пример выражения, которое может привести к ложно-положительному срабатыванию в алгоритме с эвристикой
Listing 3. The example of an expression that may lead to a false positive

Рассмотрим ситуацию, когда базовый блок с данным выражением является крайним в $M_{\text{int, h}}$. Под крайним здесь имеется в виду то, что ни один из наследников этого базового блока не принадлежит $M_{\text{int, h}}$. То есть из вызовов внутри наследников по эвристике не может произойти `longjmp`. Тогда и присваивания переменной `var` в случае вызова `longjmp` внутри `mb_longjmp` не произойдет. Таким образом, шаг 6 базового алгоритма дополняется следующим образом:

- Базовый блок с искомым переопределением должен содержаться в $M_{\text{int, h}}$, и хотя бы один из наследников данного базового блока также должен содержаться в $M_{\text{int, h}}$.

4.7 Учет области действия `jmp_buf`

Рассмотрим следующий пример (Листинг 4).

```
{
   jmp_buf env;
   setjmp (env);

   ... some code ...

   mb_longjmp_local (env);
   var = 10;
   no_longjmp ();
}
```

Листинг 4. Пример кода, который может привести к ложно-положительному срабатыванию
Listing 4. The example of a code that may lead to a false positive

Здесь `jmp_buf env` объявлен локальным. Тогда `env` может достигнуть `longjmp` внутри некоторого вызова, только если он явно передается в этот вызов. Таким образом, после переопределения переменной `var` вызов `longjmp` невозможен. Аналогичные примеры можно привести для `jmp_buf` с любой областью действия. Для уточнения алгоритма для этого случая планируется модификация шага 2 базового алгоритма:

- Вычисление $M_{\text{sj, jb}}$ – множества базовых блоков, достигающих B_{ab} с учетом возможности утечки `jmp_buf` в вызов функции (то есть анализ сигнатуры и места определения вызываемой функции исходя из области действия `jmp_buf`). Например, если `jmp_buf` локальный, как в примере, то его утечка в некоторый вызов возможна, только если он явно передается в этот вызов.

На остальных шагах вместо M_{sj} используется новое $M_{\text{sj, jb}}$.

5. Тестирование и анализ результатов

Тестирование заключалось в сборке дистрибутива Alpine Linux 3.16. Для сборки использовался Безопасный компилятор на базе GCC 11.2.1 с флагами `-O2 -Safe2` (опция `-Safe2` принудительно включает `-Werror=clobbered`). Безопасным компилятором собираются около 3700 пакетов дистрибутива, из которых около 700 используют `setjmp`.

Количество пакетов со срабатыванием `-Wclobbered`:

- Алгоритм на RTL (GCC): 125 пакетов.
- Алгоритм на GIMPLE базовый: 102 пакета (по сравнению с RTL устранено 14 ложно-отрицательных и 37 ложно-положительных).
- Алгоритм на GIMPLE с эвристикой: 85 пакетов.
- Алгоритм на GIMPLE с учетом области действия `jmp_buf` на данный момент находится в разработке.

Большинство срабатываний нового алгоритма на GIMPLE были определены как истинные в рамках существующих ограничений. Небольшое количество выявленных ложно-положительных срабатываний вызвано сложными для анализа случаями. Например, сюда относятся запутанные условия совместимости кода с анормальным возвратом (большое количество нетривиально пересекающихся предикатов). В дальнейшем данные ошибки планируется исправить.

Разработчикам ряда пакетов было сообщено об обнаруженных срабатываниях. Нами были предложены изменения программного кода, устраняющие потенциальное неопределенное поведение. В 5 пакетах предложенные изменения были приняты и включены разработчиками в более новые версии пакетов.

6. Заключение

В рамках данной работы были рассмотрены существующие и предложен новый метод обнаружения случаев неопределенного поведения в языке Си при использовании функций нелокальных переходов. Были решены следующие задачи:

- Рассмотрены существующие методы. Ввиду их недостатков они были признаны неэффективными для практического применения на реальных проектах.
- Разработан и описан новый алгоритм диагностики, а также эвристика, повышающая точность срабатываний на реальном коде. Фундаментальное ограничение данного алгоритма – зависимость он наличия phi-функции перед вызовом `setjmp`. Если у некоторой переменной нет такой phi-функции, то ее анализ невозможен.
- Представленный алгоритм с эвристикой реализован в Безопасном компиляторе, разрабатываемом в ИСП РАН на основе GCC.
- Реализация была протестирована сборкой Alpine Linux 3.16. Общее количество пакетов со срабатыванием: 85.

Дальнейшая планируемая в рамках задачи работа:

- Подробный анализ срабатываний.
- Исправление обнаруженных недостатков.
- Внедрение учета области действия `jmp_buf` в текущую реализацию.

Список литературы / References

[1]. C11 Standard ISO/IEC 9899:2011 // Programming language – C. – 2011. – P. 561. – International standard.
[2]. Herity, Dominic. C++ in embedded systems: Myth and reality / Dominic Herity // Embedded Systems Programming. – 1998. – Vol. 11, no. 2. – Pp. 48–71.
[3]. ISO/IEC/IEEE 9945:2009 // Portable Operating System Interface (POSIX®) Base Specifications, Issue 7. – 2009. – P. 37. – International standard.
[4]. Xu, Xiao. Research on coroutine-based process interaction simulation mechanism in c++ / Xiao Xu, Ge Li // AsiaSim 2012: Asia Simulation Conference 2012, Shanghai, China, October 27-30, 2012. Proceedings, Part III / Springer. – 2012. – Pp. 178–187.
[5]. Baev, Roman Vyacheslavovich. Prevention of vulnerabilities arising from optimization of code with Undefined Behavior / Roman Vyacheslavovich Baev, Leonid Vladlenovich Skvortsov, Evgeny Alekseevich Kudryashov, Ruben Arturovich Buchatskiy, Roman Aleksandrovich Zhuykov // Proc. Inst. Syst. Program. RAS. – 2021. – Vol. 33, no. 4. – Pp. 195–210.
[6]. Novillo, Diego. GCC Internals-Internal Representations / Diego Novillo // GCC IR-2. – 2007.
[7]. The internals of the GNU compilers. – Accessed: 2023-11-04. Available at:https://gcc.gnu.org/onlinedocs/gccint/.
[8]. Merrill, Jason. Generic and gimple: A new tree representation for entire functions / Jason Merrill // Proceedings of the 2003 GCC Summit. – 2003. – Pp. 171–180.
[9]. [9/10/11/12 Regression] clobbered by longjmp warning ignores the data flow // Bug 21161, GCC. – Accessed: 2023-11-04. Available at: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=21161.
[10]. Aho, Alfred V. Compilers: principles, techniques, and tools / Alfred V Aho, Ravi Sethi, Jeffrey D Ullman // Addison-wesley Reading – 2007. – Vol. 2. - Pp. 399 - 410.
[11]. Jambor, Martin. The new intraprocedural scalar replacement of aggregates / Martin Jambor //GCC Developers' Summit. – 2010. – Vol. 47.
[12]. Tarjan, Robert. Depth-first search and linear graph algorithms / Robert Tarjan // SIAM journal on computing. – 1972. – Vol. 1, no. 2. – Pp. 146–160.
[13]. Haghighat, Mohammad R. Symbolic analysis for parallelizing compilers / Mohammad R Haghighat. No. 1880. – Springer Science & Business Media, 1995.

Информация об авторах / Information about authors

Никита Юрьевич ШУГАЛЕЙ – старший лаборант кафедры Системного программирования Московского физико-технического института. Сфера научных интересов: компиляторные технологии, безопасность программного обеспечения, методы статического анализа кода, оптимизация программ.

Nikita Yurievich SHUGALEY – senior laboratory technician of the Department of system programming of the Moscow Institute of Physics and Technology. Research interests: compiler technologies, software security, methods of static code analysis, program optimization.

Владислав Анатольевич ИВАНИШИН – научный сотрудник компиляторного отдела Института системного программирования. Научные интересы включают в себя компиляторные технологии, операционные системы, безопасность программного обеспечения, методы статического анализа кода, оптимизацию программ.

Vladislav Anatolevich IVANISHIN – researcher at the Compiler Department of the Institute for System Programming of the RAS. Research interests include compiler technologies, operating systems, software security, methods of static code analysis, and program optimization.

Александр Владимирович МОНАКОВ – старший научный компиляторного отдела Института системного программирования. Сферой научных интересов является компиляторные технологии, оптимизация программ.

Alexander Vladimirovich MONAKOV – researcher at the Compiler Department of the Institute for System Programming of the RAS. Research interests: compiler technologies, program optimization.