DOI: 10.15514/ISPRAS-2021-33(4)-14

Предотвращение уязвимостей, возникающих в результате оптимизации кода с неопределенным поведением

Р.В. Баев, ORCID: 0000-0002-7999-7952 <baev@ispras.ru>
Л.В. Скворцов, ORCID: 0000-0002-1580-1244 <lvs@ispras.ru>
Е.А. Кудряшов, ORCID: 0000-0002-2361-7172 <kudryashov@ispras.ru>
Р.А. Бучацкий, ORCID: 0000-0001-8522-1811 <ruben@ispras.ru>
Р.А. Жуйков, ORCID: 0000-0002-0906-8146 <zhroma@ispras.ru>

Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. С развитием оптимизирующих компиляторов стали возникать случаи появления уязвимостей в программах во время оптимизации. Это связано с тем, что зачастую программисты используют конструкции с неопределенным поведением, опираясь на свое представление о том, в какой код такие конструкции транслировались знакомым им компилятором для определенной архитектуры. В то же время компилятор, руководствующийся стандартом языка, вправе проводить оптимизации так, как будто бы таких конструкций в коде не может существовать. В этой статье описываются подходы к обнаружению и устранению уязвимостей в программах, рассматривается применимость этих подходов для случая уязвимостей, появляющихся вследствие оптимизации, в условиях, когда возможность изменения исходного кода ограничена или отсутствует. В статье предлагается концепция безопасного компилятора, т.е. компилятора, обеспечивающего отсутствие внесения уязвимостей в программу во время оптимизации, и описывается реализация такого компилятора на базе компилятора GCC. Для безопасного компилятора приводится разделение реализованного функционала на три уровня защиты и описание применимости этих уровней, показывается применимость безопасного компилятора на практике, а также оценивается изменение производительности получаемой программы.

Ключевые слова: компилятор; уязвимость; неопределенное поведение

Для цитирования: Баев Р.В., Скворцов Л.В., Кудряшов Е.А., Бучацкий Р.А., Жуйков Р.А. Предотвращение уязвимостей, возникающих в результате оптимизации кода с неопределенным поведением. Труды ИСП РАН, том 33, вып. 4, 2021 г., стр. 195-210. DOI: 10.15514/ISPRAS–2021–33(4)–14

Prevention of vulnerabilities arising from optimization of code with Undefined Behavior

R.V. Baev, ORCID: 0000-0002-7999-7952 <baev@ispras.ru>
L.V. Skvortsov, ORCID: 0000-0002-1580-1244 <lvs@ispras.ru>
E.A. Kudryashov, ORCID: 0000-0002-2361-7172 <kudryashov@ispras.ru>
R.A. Buchatskiy, ORCID: 0000-0001-8522-1811 <ruben@ispras.ru>
R.A. Zhuykov, ORCID: 0000-0002-0906-8146 <zhroma@ispras.ru>

Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. Aggressive optimization in modern compilers may uncover vulnerabilities in program code that did not lead to bugs prior to optimization. The source of these vulnerabilities is in code with undefined behavior. Programmers use such constructs relying on some particular behavior these constructs showed before in their experience, but the compiler is not obliged to stick to that behavior and may change the behavior if it's needed for optimization since the behavior is undefined by language standard. This article describes approaches to detection and elimination of vulnerabilities arising from optimization in the case when source code is available but its modification is undesirable or impossible. Concept of a safe compiler (i.e. compiler that ensures no vulnerability is added to the program during optimization) is presented and implementation of such a compiler on top of GCC compiler is described. Implementation of safe compiler's functionality is divided into three security levels whose applicability is discussed in the article. Feasibility of using the safe compiler on real-world codebases is demonstrated and possible performance losses are estimated.

Keywords: compiler; vulnerability; undefined behavior

For citation: Baev R.V., Skvortsov L.V., Kudryashov E.A., Buchatskiy R.A., Zhuykov R.A. Prevention of vulnerabilities arising from optimization of code with Undefined Behavior. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 4, 2021. pp. 195-210 (in Russian). DOI: 10.15514/ISPRAS–2021–33(4)–14

1. Введение

К появлению уязвимостей в программах могут приводить не только ошибки, допущенные программистом, или злой умысел, но и агрессивные оптимизации кода, выполняемые компилятором, как это показано в работе [1]. В таких случаях одинаковый исходный код может быть скомпилирован в корректно работающий исполняемый код, если компилятор по какой-то причине не использует некую конкретную оптимизацию (например, все оптимизации отключены, или в выбранной версии компилятора такая оптимизация отсутствует, или оптимизация не применяется для выбранной аппаратной платформы), и в исполняемый код, содержащий уязвимость, если эта оптимизация используется. При этом невозможно точно предсказать, сохранится ли ожидаемая программистом семантика программы при смене флагов компиляции, версии компилятора, самого компилятора (так, например, некоторые подобные ошибки проявлялись при замене GCC на Clang) или целевой архитектуры.

В большей части таких случаев речь идет о компиляции кода, содержащего конструкции с неопределенным поведением (undefined behavior). Неопределенным поведением называется результат выполнения конструкций языка, не регламентированный стандартом языка. В случае языка С примерами таких конструкций являются разыменование нулевого указателя, переполнение знаковых целых чисел и др. Компилятор действует, исходя из предположения, что компилируемая программа строго соответствует стандарту языка и, таким образом, не вызывает неопределенного поведения. Это позволяет ему более эффективно оптимизировать код в ряде случаев, однако может привести к появлению уязвимости в случае, если в компилируемой программе присутствовало неопределенное поведение.

Уязвимости, возникающие вследствие оптимизации кода компилятором, не ограничиваются случаями неопределенного поведения: есть пример корректной с точки зрения стандарта

языка оптимизации, вносящей уязвимость в код без неопределенного поведения. Перезапись буфера памяти без его последующего использования может быть удалена компилятором как не влияющая на результат выполнения программы, что приведет к тому, что чувствительные данные могут остаться в памяти, откуда их сможет получить злоумышленник.

Код, который может быть удален компилятором вследствие того, что компилятор полагается на отсутствие в программе неопределенного поведения, называется нестабильным кодом [2]. В рамках этой работы так же будет называться код, в котором удалена перезапись буфера памяти.

В разд. 2 демонстрируются примеры уязвимостей, возникающих в результате агрессивных оптимизаций кода компилятором. В разд. 3 обсуждаются подходы, применяемые на практике для обнаружения и предотвращения уязвимостей. В разд. 4 описывается безопасный компилятор и его область применимости. В разд. 5 приводятся некоторые подробности реализации безопасного компилятора и уровней защиты в нем. Разд. 6 посвящен результатам тестирования безопасного компилятора на реальных приложениях.

2. Примеры уязвимостей

В этом разделе приведены примеры некоторых типов уязвимостей, возникающих в результате агрессивных оптимизаций кода компилятором. В работе [1] приведено еще 4 типа уязвимостей, основанных на оптимизации неопределенного поведения. В рамках данного исследования было найдено еще несколько типов. Подробнее о поиске таких уязвимостей рассказывается в подразд. 6.1.

2.1 Неопределенное поведение

2.1.1 Слишком большой аргумент операции сдвига

На листинге 1 продемонстрирован фрагмент кода файловой системы ext4. Изначально этот фрагмент содержал уязвимость: в некоторых случаях значение переменной `groups_per_flex` становилось нулем, что приводило выполнению деления на ноль и, соответственно, исключению. Такая ситуация могла случиться, например, на 32-битной архитектуре PowerPC, если в `sbi->s_log_groups_per_flex` было значение 32 (на PowerPC для хранения аргумента сдвига отведено 6 бит, на x86 — только 5, поэтому выполнить сдвиг на 32 бита не получится). Исправление, добавляющее проверку `groups_per_flex` на равенство нулю, должно было устранить эту уязвимость, однако на самом деле этого не произошло. Причина в том, что с точки зрения стандарта языка C сдвиг знакового целого числа на число бит, большее или равное ширине его типа, является неопределенным поведением (п. 6.5.7. стандарта C11 [3]).

```
groups_per_flex = 1 << sbi->s_log_groups_per_flex;
/* There are some situations, after shift the
    value of 'groups_per_flex' can become zero
    and division with 0 will result in fixpoint
    divide exception */
if (groups_per_flex == 0)
    return 1;
flex_group_count = ... / groups_per_flex;
```

Листинг 1. Код из fs/ext4/super.c в ядре Linux 2.6.31
Listing 1. Code from fs/ext4/super.c in Linux kernel 2.6.31

Компилятор может доказать, что в отсутствие неопределенного поведения значение `groups_per_flex` строго положительно, поэтому выражение `groups_per_flex == 0` тождественно ложно, и проверку можно удалить. Именно так поступает компилятор Clang. Компиляторы GCC и Clang не предоставляют опции отключения подобной оптимизации.

2.1.2 Переполнение знакового целого

На листинге 2 приведен фрагмент реализации системного вызова `vfs_fallocate` в текущей (5.14) версии ядра Linux. Значения переменных offset и len приходят из пользовательского пространства, поэтому требуют проверки. Сначала проверяется, что значение `offset` неотрицательно, а значение len положительно. Кроме того, программист, писавший этот код, понимал (о чем свидетельствует комментарий), что при сложении может произойти целочисленное переполнение, что приведет к тому, что сумма `offset + len` будет меньше значения `inode->i_sb_s_maxbytes`, и выполнение функции продолжится вместо возврата со значением -EFBIG. Для устранения этой уязвимости он добавил в условие проверку на переполнение при сложении `offset + len`. Как и в случае, рассмотренном в предыдущем подразделе, добавленная проверка не приводит к устранению уязвимости.

```
int vfs_fallocate(..., loff_t offset, loff_t len)
{
    struct inode *inode = ...;
    if (offset < 0 || len <= 0)
        return -EINVAL;
    /* Check for wrap through zero too */
    if ((offset + len > inode->i_sb->s_maxbytes)
        || (offset + len < 0))
        return -EFBIG;
    …
}
```

Листинг 2. Код из fs/open.c в ядре Linux 5.14
Listing 2. Code from fs/open.c in Linux kernel 5.14

Компилятор GCC способен доказать, что проверка на переполнение происходит только в случае неотрицательного значения `offset` и положительного значения `len`. Переполнение знакового целого числа является неопределенным поведением (п. 6.5. стандарта C11 [3]), поэтому компилятор вправе рассчитывать, что этого не произойдет. Поскольку, с точки зрения компилятора, в проверке написано тождественно ложное выражение (некое положительное значение меньше нуля), он вправе удалить это выражение (`expr || false` ⇔ `expr`). Таким образом, уязвимость остается на месте. Подобное недопонимание между компилятором и программистом происходит достаточно часто, поэтому и в GCC, и в Clang была добавлена опция `-fno-strict-overflow`, которая позволяет запретить компилятору полагаться на отсутствие целочисленного переполнения. Ядро Linux, начиная с версии 2.6.31, и другие крупные проекты в обязательном порядке используют эту опцию при сборке.

2.1.2 Чтение неинициализированной переменной

На листинге 3 приведен код из функции `srandomdev` в библиотеке libc в FreeBSD 8.0. В случае компиляции без оптимизаций наблюдаемым поведением в этом случае будет хранение в переменной `junk` некого “мусорного” значения со стека. Судя по комментарию и сообщению в соответствующем коммите, автор этого фрагмента рассчитывал именно на такое поведение программы для использования в качестве источника энтропии. Здесь имеется сразу две проблемы: во-первых, стандарт языка ничего не говорит о размещении переменных, это решение принимает компилятор (GCC с включенными оптимизациями в данном случае размещает переменную на регистре), а во-вторых, чтение неинициализированной переменной является неопределенным поведением, что позволяет компилятору, например, заменить все выражение в аргументе функции srandom константой (именно таким образом поступает Clang).

```
struct timeval tv;
unsigned long junk; /* XXX left uninitialized on purpose */

gettimeofday(&tv, NULL);
srandom((getpid() << 16) ^ tv.tv_sec ^ tv.tv_usec ^ junk);
```

Листинг 3. Код из lib/libc/stdlib/rand.c из FreeBSD 8.0
Listing 3. Code from lib/libc/stdlib/rand.c in FreeBSD 8.0

На момент написания статьи в Clang было принято изменение, добавляющее опцию `-ftrivial-auto-var-init`. Эта опция инициализирует автоматические переменные указанным значением, не оставляя, таким образом, в программе неинициализированных переменных. Схожее изменение было предложено в GCC в феврале 2021 года, однако на момент написания статьи оно еще не было принято.

2.2 Удаление записи в память

На листинге 4 приведен пример кода, который не содержит неопределенного поведения, но тем не менее, в результате работы оптимизирующего компилятора в получающемся коде может появиться уязвимость. Стандарт языка C не обязывает компилятор сохранять побочные эффекты, которые не влияют на результат вычислений. В приведенном фрагменте кода таким побочным эффектом будет запись в память в результате вызова функции `memset`.

```
void func(void) {
    char password[1000] = {0};
    get_password(password);
    use_password(password);
    memset(password, 0, 1000); // delete password from memory
}
```

Листинг 4. Код, стирающий чувствительные данные из памяти
Listing 4. Code to erase sensitive data from memory

Компилятор может рассудить, что раз эти данные никогда не будут прочитаны, то эффективнее их не записывать, и удалить вызов `memset`. В результате такого решения чувствительные данные останутся в памяти, и злоумышленник сможет их прочитать, воспользовавшись другой уязвимостью (в том числе, возможно, в другой программе). Ни GCC, ни Clang не предоставляют надежного способа избежать подобной уязвимости.

3. Подходы к обнаружению и предотвращению уязвимостей

3.1 Инструменты статической проверки кода

Одним из основных способов поиска уязвимостей в программе является статический анализ. Статический анализ работает с исходным кодом программы. Статический анализ является мощным инструментом поиска ошибок, однако не лишен недостатков.

Во-первых, некоторые типы ошибок проявляются только во время исполнения, статический анализатор, который будет пытаться отмечать все возможные места таких ошибок, будет бесполезен на практике, поскольку ложноположительных срабатываний будет на несколько порядков больше, чем истинных.

Во-вторых, в общем случае, статический анализатор не имеет информации о компиляторе, который будет использоваться, и о целевой архитектуре, которая необходима для поиска уязвимостей, связанных с нестабильным кодом. В качестве исключения можно привести в пример инструмент STACK [4], который нацелен на поиск кода, нестабильного с точки зрения Clang. Этот инструмент основывается на знании устройства оптимизаций в Clang и способен находить случаи, подобные описанным в 2.1, но со временем оптимизатор может

научиться использовать другие типы неопределенного поведения, о чем может быть неизвестно инструменту STACK.

В-третьих, общим местом для всех инструментов статического анализа является тот факт, что статические анализаторы только сигнализируют о возможной ошибке, но не исправляют и не предотвращают ошибки. Это может быть недостатком в случаях, когда модификация исходного кода по каким-то причинам невозможна или затруднительна.

3.2 Инструменты динамической проверки кода

К инструментам динамического анализа относятся такие инструменты как Undefined Behavior Sanitizer (UBSan) [5] и Address Sanitizer (ASan) [6], входящие в состав GCC и Clang, интерпретатор tis-interpreter [7]. Санитайзеры вставляют в код программы проверки, обнаруживающие ошибки (в том числе случаи неопределенного поведения). При срабатывании этих проверок программа аварийно завершается, но в некоторых случаях UBSan позволяет продолжить выполнение программы после ошибки (так, например, есть возможность продолжить выполнение программы после целочисленного переполнения, в этом случае поведение программы совпадает с поведением программы, скомпилированной с флагом -fwrapv). Стоит отметить, что инструментация, используемая UBSan, не позволяет компилятору удалить нестабильный код, причиной нестабильности которого является неопределенное поведение. Инструмент tis-interpreter является интерпретатором языка C, который во время выполнения конструкций программы проверяет, способны ли они вызывать неопределенное поведение.

Преимуществом средств динамического анализа является большая точность по сравнению со средствами статического анализа. Общими недостатками средств динамического анализа являются заметное замедление анализируемой программы (замедление в 2-3 раза считается нормальным при использовании UBSan [8]), а также зависимость качества анализа от входных данных (динамический анализ не найдет ошибок на невыполнявшихся путях). Также упомянутые средства динамического анализа неспособны устранить уязвимость, приведенную в 2.2.

В отличие от средств статического анализа, UBSan способен предотвратить уязвимости вследствие наличия нестабильного кода, однако для большого числа реальных программ замедление, вызываемое использованием UBSan является неприемлемым.

3.3 Тестирование

Для обнаружения уязвимостей в скомпилированной программе может использоваться тестирование, в том числе автоматизированное. Генерация тестовых наборов также может производиться автоматически с помощью инструментов таких как KLEE [9]. Важно, чтобы используемые инструменты корректно моделировали неопределенное поведение с учетом выбранной целевой архитектуры. Такое тестирование должно проводиться для той же целевой архитектуры и версии компилятора, которые будут выбраны для финальной сборки. Важно отметить, что, как и статический анализ, тестирование способно сигнализировать об ошибке, но не предотвратить появление уязвимости.

3.4 Изменение стратегий оптимизации в компиляторе

Поскольку проблема, обсуждаемая в данной работе, состоит в том, что нестабильный код удаляется компилятором в ходе оптимизаций, одним из возможных способов борьбы с этим является отключение оптимизаций. Для некоторых случаев нестабильного кода существуют флаги компилятора, отключающие соответствующие оптимизации или меняющие их поведение (к таким флагам относятся `-fno-strict-overflow`, `-fno-delete-null-pointer-checks` и другие), но для некоторых (как для случая, приведенного в 2.1.1) таких

флагов не предусмотрено. При этом полное отключение оптимизаций (уровень оптимизации `-O0`) с одной стороны является крайне нежелательным с точки зрения производительности, а с другой стороны не гарантирует отсутствия уязвимостей (так, например, GCC с уровнем оптимизации `-O0` способен удалять такие проверки как `x + 100 < x`). К преимуществам этого подхода относится отсутствие необходимости модификации исходного кода.

3.5 Использование безопасных функций

Некоторые уязвимости могут быть предотвращены с помощью использования безопасных аналогов уязвимых функций. В языке C, начиная с версии C11, представлен набор функций (описаны в приложении K к стандарту языка C [3]) с суффиксом `_s`. В отличие от соответствующих функций без суффикса `_s`, они должны проверять ограничения во время выполнения (так, например, функция `memcpy_s` должна проверять, что указатели в ее аргументах не равны NULL, значения размеров не превосходят максимального значения, буферы не пересекаются и т.д.). Применительно к проблеме нестабильного кода интересна функция `memset_s`: ее спецификация требует обязательного сохранения записи в память (в отличие от спецификации функции `memset`). Это отличие позволяет избежать уязвимостей подобных описанной в 2.2.

Функции из приложения K к стандарту языка C не являются обязательными для реализации в библиотеке языка C для соответствия стандарту, многие реализации библиотеки (glibc и musl в их числе) не включают эти функции и не собираются их включать в обозримом будущем.

Многие реализации библиотеки языка C включают свои реализации функции со схожим функционалом (`explicit_bzero` в glibc, `SecureZeroMemory` в Windows), однако они не являются стандартными и не гарантируется, что компилятор никогда не сможет оптимизировать их аналогично `memset`.

Использование данного подхода требует модификации исходного кода, причем заметно более нетривиальной [10], чем задумывалось создателями функций из приложения K.

3.6 Проверка семантики

В статье [11] описывается "негативная" семантика языка C. Под "негативной" понимается семантика, описывающая неопределенное поведение. На основе негативной семантики реализован инструмент kcc, основывающийся на фреймворке K. kcc способен находить большое число случаев неопределенного поведения. kcc является академическим проектом и не является эффективным с точки зрения применимости, в первую очередь из-за ограниченной поддержки стандартных библиотек. Коммерческим продолжением kcc является RV-Match [12]. RV-Match поддерживает стандарт C11. В отличие от kcc, RV-Match не требует, чтобы всех используемых в проекте библиотек была определена семантика. По задумке авторов, kcc (и, соответственно, RV-Match) должно быть возможно использовать как прямую замену gcc без изменения системы сборки. Известно, что для некоторых программ, например, gzip, такая замена действительно возможна. Эффективность RV-Match показана в работе [13]. В этой же работе заявляется работоспособность RV-Match для приложений объемом ~300 тысяч строк кода. К недостаткам использования kcc/RV-Match можно отнести следующее:

- неполная поддержка стандартных библиотек;
- часть типов неопределенного поведения обнаруживается только в момент выполнения, т.е. эффективность зависит от полноты тестового покрытия;
- неизвестно качество работы инструмента на больших проектах; размер некоторых современных проектов (например, PostrgeSQL или ядро Linux) превосходит 1 миллион строк кода;

- позволяет обнаруживать неопределенное поведение, но не позволяет предотвратить уязвимость: kcc нельзя использовать в релизной сборке программы из-за низкой производительности, для предотвращения уязвимости придется исправлять исходный код программы;
- инструмент нацелен на поиск неопределенного поведения, поэтому не сможет найти нестабильный код, подобный приведенному в 2.2.

3.7 Использование стандартов безопасного программирования

При создании программ, используемых в областях, в которых безопасность критически важна (медицина, авиация и т.д.), применяются стандарты программирования, нацеленные на повышение безопасности, такие как MISRA [14], SEI CERT C Coding Standard [15] и другие. Эти стандарты ограничивают используемое подмножество языка и вводят указания по использованию этого подмножества. Это позволяет писать более безопасный код с меньшим числом уязвимостей. В рамках задачи, описанной в этой статье, использование стандартов безопасного программирования не может быть сочтено решением, поскольку подразумевается, что исходный код уже написан, а не пишется с нуля с соблюдением стандартов.

4. Концепция безопасного компилятора

В ситуациях, когда исходный код программы доступен, однако по каким-либо причинам не может быть модифицирован, или модификация потребует слишком больших усилий, большинство из способов, описанных в разделе 3, не приведут к желаемому результату: предотвращению уязвимостей. Инструменты динамического анализа в общем случае не способны заметить, что уязвимость, подобная описанным в 2.1, появилась в результате работы компилятора. Отдельно надо сказать об Undefined Behavior Sanitizer: при его использовании инструментация вставляется в промежуточное представление программы до этапа работы оптимизатора, что приводит к тому, что оптимизатор не будет удалять нестабильный код, причиной нестабильности которого является неопределенное поведение. Способ, описанный в 3.4, также в некоторой степени помогает: часть оптимизаций, способных удалить нестабильный код, можно отключить. Этот способ не является полным: некоторые оптимизации нельзя отключить, не установив уровень оптимизаций -O0, и даже в этом случае, если рассматривать GCC, некоторые оптимизации не отключаются. Также нельзя назвать этот способ надежным: потребуется вносить изменения в систему сборки, но эти изменения могут быть отменены другими флагами позднее, при этом сам факт изменений в системе сборки может быть нежелательным.

В таких ситуациях возможным способом предотвратить появление уязвимостей из-за нестабильного кода будет использование безопасного компилятора, не вносящего в код уязвимости при выполнении оптимизаций. Неудаление нестабильного кода без больших потерь в производительности скомпилированной программы является основным требованием к безопасному компилятору, при этом безопасный компилятор может предоставлять дополнительные возможности для обнаружения и предотвращения уязвимостей. Также безопасный компилятор должен работать при условии, что вносимые в исходный код модификации минимальны или вообще отсутствуют. Такой безопасный компилятор можно использовать в качестве замены штатного компилятора в системе сборки, что позволит ограничиться минимальными изменения в системе сборки или даже обойтись без них. При этом безопасный компилятор не должен предоставлять возможность отключения опций, контролирующих предотвращение уязвимостей.

5. Реализация безопасного компилятора

Реализованный безопасный компилятор разработан на базе GCC 9.3.0. Он реализует статические и динамические методы защиты, а также может работать вместе со средствами безопасности ОС. К статическим методам защиты относятся диагностики, настройки оптимизаций (как отключение конкретных оптимизаций, так и изменения их поведения). К динамическим методам относится использование Undefined Behavior Sanitizer, методов защиты от переполнения буфера (FORTIFY_SOURCE [16], stack protector [17]). Безопасный компилятор предоставляет возможность использовать улучшенную версию ASLR [18].

Очевидным следствием применения новых диагностик, проверок, оптимизаций и условия невозможности модификации исходного кода программ будет то, что некоторые программы не смогут быть собраны безопасным компилятором. Это может накладывать ограничения на применимость безопасного компилятора. Разделение реализованных в безопасном компиляторе методов защиты на несколько уровней позволяет более гибко подойти к вопросу применения безопасного компилятора для компиляции реальных программ. Было предложено разделение на три уровня защиты.

Использование безопасного компилятора с уровнем защиты 3 должно позволять собирать без ошибок большинство приложений. На этом уровне компилятор должен включать опции, относящиеся к “лучшим практикам”, используемым в больших проектах (ядре Linux, Firefox, PostgreSQL). К ним относятся такие опции как `-fPIE` (компиляция в позиционно-независимый код), `-fstack-protector-strong` (использование механизма `stack protector` для функций, которые содержат что-либо из перечисленного: локальный массив, вызов функции `alloca`, локальную переменную, у который берется адрес), `-fno-strict-aliasing` (запрет компилятору считать указатели различных типов обязательно различающимися) и другие, а также использование fortified-функций, включаемых опцией `FORTIFY_SOURCE`. Эти функции являются реализацией функций стандартной библиотеки с дополнительными проверками для обнаружения выходов за границы буферов. Fortified-функции были портированы из библиотеки Musl и представлены в заголовочных файлах в составе безопасного компилятора. Также диагностика -Wclobbered, присутствующая в GCC, была переработана (в частности, новая реализация использует другое внутреннее представление GCC) в рамках безопасного компилятора. Эта диагностика потребовала переработки, т.к. версия диагностики, присутствующая в GCC, имеет большое число ложных срабатываний [19]. Ожидается, что использование уровня защиты 3 может приводить к небольшому замедлению скомпилированной программы по сравнению с программой, скомпилированной GCC 9.3.0. В обоих случаях предполагается использование уровня оптимизации `-O2`, являющегося де-факто стандартным уровнем оптимизации при сборке реальных приложений.

На уровне защиты 2 добавляются флаги управления оптимизациями и настройки оптимизаций, разработанные специально для безопасного компилятора, а также включаются стандартные опции, необходимые для их работы. К добавленным на уровне 2 флагам управления оптимизациями относятся, в частности, флаги `-fkeep-oversized-shifts` и `-fpreserve-memory-writes`. Флаг `-fkeep-oversized-shifts` запрещает компилятору проводить сворачивание констант и продвижение констант в случаях, когда второй аргумент оператора сдвига больше или равен ширине типа. Это позволяет сохранить операцию сдвига в получающемся ассемблерном коде, повторяя поведение компилятора без оптимизаций. Флаг `-fpreserve-memory-writes` указывает компилятору сохранять все побочные эффекты всех операций записи, затрагивающих участок памяти из которого есть хотя бы одно чтение. Этот флаг позволяет устранить уязвимости вследствие нестабильного кода, подобные описанной в 2.2. Использование второго уровня защиты может приводить к тому, что некоторые приложения не будут собираться. Ожидается, что переход с уровня защиты 3 на уровень защиты 2 не приведет к сильному увеличению количества программ,

которые не смогут быть собраны безопасным компилятором, но скорость работы скомпилированных программ может уменьшиться.

Уровень защиты 1 отличается от предыдущих в первую очередь использованием UBSan. В безопасном компиляторе используется UBSan с добавленными опциями, в том числе опцией `-fsanitize=function`, добавляющей инструментацию непрямых вызовов функций проверками типов. Эта опция является реализацией аналогичной опции, присутствующей в Clang. Также на этом уровне добавлены опции `-fsanitize=null` (помимо других использований нулевого указателя, добавлен запрет вызова функции по нулевому указателю) и `-fsanitize=return` (достижение конца функции типа отличного от void без возврата значения, раньше такая проверка выполнялась только для кода на языке C++). Также на уровне защиты 1 используется модифицированная версия ASLR. В отличие от обычной реализации ASLR, позволяющей размещать загружаемую программу по рандомизированному адресу в памяти, реализация ASLR в безопасном компиляторе позволяет проводить рандомизацию расположения функций в файле программы, а также рандомизацию порядка размещения локальных переменных внутри фрейма функции. Уровень защиты 1 добавляет строгие проверки, многие приложения не смогут быть собраны на этом уровне без изменений в исходном коде. Использование этого уровня позволяет защититься от большего набора уязвимостей и снизить уровень угрозы, поскольку переведет часть уязвимостей из разряда “утечка данных” и “выполнение произвольного кода” в разряд “отказ в обслуживании”. Этот уровень предназначен для случаев, в которых безопасность является наивысшим приоритетом (в пользу чего можно пожертвовать производительностью, вплоть до замедления скомпилированной программы в 2-3 раза, и возможностью собирать недостаточно надежные приложения). Табл. 1 суммирует информацию о разделении методов защиты на уровни, приведенную в этом разделе.

Включение уровня защиты в безопасном компиляторе реализовано с помощью добавления опции `-Safe` с целочисленным аргументом, аналогично опции `-O`, контролирующей уровень оптимизации. Соответственно, безопасный компилятор может быть запущен с опциями `-Safe3, -Safe2, -Safe1` для включения соответствующего уровня защиты.

Табл. 1. Примеры используемых методов защиты по уровням
Table 1. Examples of used methods of protection by level

Методы защиты	Уровень защиты 3	Уровень защиты 2	Уровень защиты 1
-fPIE	X	X	X
-fstack-protector-strong	X	X	X
-fno-strict-aliasing	X	X	X
FORTIFY_SOURCE	X	X	X
-Wclobbered (улучш.)	X	X	X
-fkeep-oversized-shifts		X	X
-fpreserve-memory-writes		X	X
-fsanitize=undefined			X
-fsanitize=function			X
-fsanitize=null			X
-fsanitize=return			X
ASLR (улучш.)			X

6. Результаты

6.1 Исследование случаев неопределенного поведения

В ходе данной работы была проанализирована возможность появления в программах нестабильного кода в результате работы оптимизирующего компилятора. Для изучения поведения компиляторов при компиляции конструкций с неопределенным поведением был составлен тестовый набор, содержащий по одному и более фрагменту кода на каждый из 203 случаев, перечисленных в приложении J к стандарту языка C версии C11. Анализ показал, что знание о 17 из 203 типов неопределенного поведения используется компиляторами при проведении оптимизаций. Еще 9 типов могли бы влиять на работу оптимизаций, однако не удалось найти подтверждений что компиляторы (исследовались GCC и Clang) используют эту возможность. Остальные типы неопределенного поведения либо обнаруживаются диагностиками компилятора в виде предупреждения или ошибки, либо компилируются в код с некой предсказуемой семантикой. Под предсказуемой семантикой понимается, что либо существует единственный разумный способ скомпилировать конструкцию с таким случаем неопределенного поведения (ярким примером может служить неопределенное поведение из п. 6.4.2.1 стандарта C11 “Два идентификатора отличаются только в незначимых символах” — все современные компиляторы используют полные имена идентификаторов, а не только первые 16 символов), либо результатом компиляции будет код, содержащий ошибку, причем этот результат не будет зависеть от используемой версии компилятора, уровня оптимизаций и целевой архитектуры.

Проведение подобного исследования для языка C++ представляется заметно более трудоемким по следующим причинам: язык C++ более сложный, чем язык C (так, стандарт C++17 занимает 1448 страниц, а стандарт C11 — 696), причем для языка C++ не существует исчерпывающего списка неопределенного поведения, аналогичного приложению J к стандарту C11. Составление подобного списка само по себе является нетривиальной задачей. В 2019 году в рабочую группу по стандартизации языка C++ было внесено предложение [20] о составлении такого списка, однако на момент написания статьи оно не было принято.

6.2 Корректность

Для проверки корректности работы безопасного компилятора был составлен тестовый набор, включающий в себя 3 типа тестов: тесты, проверяющие наличие вывода компилятором требуемой диагностики, тесты, проверяющие срабатывание динамических проверок во время выполнения, и тесты, проверяющие результат кодогенерации. Тесты, проверяющие результат кодогенерации, построены на основе исследования случаев нестабильного кода. Тестовый набор разделен на 3 уровня соответственно уровням защиты. Безопасный компилятор, запущенный с опцией, устанавливающей уровень защиты, должен проходить все тесты соответствующего уровня. Разработанный в рамках этой работы безопасный компилятор успешно проходит тесты.

6.3 Сборка дистрибутивов Linux

Помимо сборки реальных приложений по отдельности с помощью безопасного компилятора, о чем будет подробнее рассказано в следующем разделе, применимость безопасного компилятора в реальных условиях оценивалась на сборке дистрибутивов Linux. Необходимо отметить, что сборка дистрибутива Linux компилятором, отличным от стандартного для данного дистрибутива, чаще всего является нетривиальной задачей — например, даже при обновлении версии GCC чаще всего требуется изменение исходного кода в некоторых пакетах, не говоря уже сборке каким-либо другим компилятором, например, Clang – на данный момент такая задача решена не для всех пакетов. Несмотря на то, что безопасный компилятор разработан на основе GCC, с рядом пакетов дистрибутива Linux (прежде всего,

ядра и некоторых системных) возникают трудности. Данный вопрос заслуживает подробного анализа и рассмотрения в рамках отдельной работы, а в данной статье мы приводим результаты предварительных экспериментов.

Для экспериментов использовались CRUX 3.5 [21] (версия ядра 4.19.48) и Debian 10.2.0. В случае CRUX безопасный компилятор с уровнем защиты 3 обнаружил проблемы в 28 пакетах из 100. Эти проблемы удалось устранить с помощью модификации исходного кода и собрать CRUX с уровнем защиты 3. Попытка собрать CRUX с уровнем защиты 2 показала, что необходима заметно большая модификация исходного кода, которая на данном этапе не выполнялась. В случае Debian возникли проблемы со сборкой 83 пакетов из 983 с уровнем защиты 3, устранение этих проблем потребовало бы слишком больших усилий, и в рамках данной работы также не проводилась.

При сборке реального дистрибутива сложности возникают прежде всего с ядром Linux, системными библиотеками и программами (например, grub и busybox), средами исполнения (python), а также программами, содержащими очень старый код, не соответствующий стандартам (cpio). Так, например, ядро не может быть собрано с опцией stack protector, системные программы используют в коде неподдерживаемые техники оптимизации, что противоречит настройкам безопасного компилятора для всех уровней безопасности. Работа по исправлению исходного кода пакетов, который не соответствует стандарту языка C, предполагает не только чисто технические исправления, но зачастую требует сложного анализа, для того чтобы переписать несовместимые части приложения, не повлияв на его функциональность. Такая работа должна вестись разработчиками дистрибутива с привлечением разработчиков приложений.

Работа, проведенная в рамках проекта безопасного компилятора, показала, что такая большая задача, как сборка дистрибутива операционной системы, является выполнимой, но требует взаимодействия с разработчиками операционной системы. На момент написания этой статьи эта работа была в начальной стадии.

6.4 Исследование производительности

Производительность программ, скомпилированных безопасным компилятором тестировалась в 5 сценариях:

- воспроизведение части партии в го с помощью приложения GNU Go 3.8 [22], являющегося частью набора тестов SPEC [23];
- перекодирование набора WAV-файлов в формат MP3 с помощью приложения LAME 3.100 [24], являющегося частью набора тестов Phoronix Test Suite [25];
- запуск реализации бенчмарка fannkuch, входящего в состав набора тестов The Computer Language Benchmarks Game [26];
- перекодирование набора видеофайлов из формата YUV в формат MKV при помощи библиотеки x264 [27], входящей в состав набора тестов SPEC (x264-snapshot-20190407-2245-stable);
- сжатие текстового файла при помощи библиотеки zlib 1.2.11 [28], являющейся частью набора SPEC.

Табл. 2. Результаты измерения производительности
Table 2. Performance evaluation results

Сценарий	Baseline	Safe3	Safe3 замедл.	Safe2	Safe2 замедл.	Safe1	Safe1 замедл.
GNU Go	4.67 с	4.85 с	3.85%	5.23 с	11.99%	9.32 с	99.57%
LAME	3.45 с	3.55 с	2.89%	3.55 с	2.83%	9.89 с	186.31%
fannkuch	2.03 с	2.42 с	19.21%	2.42 с	19.21%	3.51 с	72.91%

x264	7.91 с	7.90 с	-0.23%	8.06 с	1.78%	18.30 с	131.21%
zlib	2.25 с	2.29 с	2.00%	2.28 с	1.33%	2.80 с	24.44%

В табл. 2 приведены результаты измерения времени выполнения тестовых сценариев на компьютере с процессором Intel® Core™ i5-7600K (архитектура x86_64) и операционной системой Ubuntu 20.04.2. Столбец baseline соответствует запуску компилятора GCC версии 9.3.0 с уровнем оптимизации -O2. Столбцы Safe3, Safe2, Safe1 соответствуют запускам безопасного компилятора с уровнями защиты 3, 2 и 1 соответственно и уровнем оптимизации -O2. Для каждого из этих столбцов в столбце справа указано замедление соответствующего сценария в процентах. В столбцах указано среднее по 20 запускам время, округленное до сотой секунды, это приводит к тому, что для одинаковых значений времени может быть небольшое отличие в значении замедления. В двух случаях (между Baseline и Safe3 в сценарии x264, и между Safe3 и Safe2 в сценарии zlib) наблюдается ускорение в среднем на 0.01 с, что не является значимым результатом.

Представленные данные показывают, что в случае использования уровня защиты 3, замедление скомпилированной программы составляет не более 20%, причем в 4 сценариях из 5 не превышает 4%. Использование уровня защиты 2 ухудшает производительность в 2 сценариях из 5 относительно уровня защиты 3, но замедление также не превышает 20%. Нужно заметить, что замедление может усиливаться в случае программ, в которых заметный выигрыш от оптимизации достигается удалением лишних записей в память. При использовании уровня защиты 1 замедление составляет от 24% до 186%, что согласуется с ожиданием: в [29] указывается, что при использовании санитайзеров программа может замедляться в 2-3 раза.

7. Заключение

В рамках данной работы было проведено исследование возникновения уязвимостей в результате работы компиляторных оптимизаций. Основным видом уязвимостей, рассмотренных в этой работе, были уязвимости в результате наличия нестабильного кода, т.е. кода, который может быть удален в результате агрессивных оптимизаций. В работе были рассмотрены подходы, применяемые для обнаружения и устранения уязвимостей в коде программ. Для решения описанной проблемы была предложена концепция безопасного компилятора. Также в статье описана реализация безопасного компилятора на базе компилятора GCC, в нем реализовано три уровня защиты.

Реализованный безопасный компилятор был успешно применен для сборки реальных программ, а также для сборки дистрибутивов операционных систем CRUX и Debian. В случае Debian не удалось добиться полной сборки дистрибутива при сохранении условия минимальности вносимых изменений, задачу сборки дистрибутива планируется решить в дальнейшем во взаимодействии с разработчиками дистрибутива, разработав набор исправлений его исходного кода.

Список литературы / References

[1]. Wang X., Chen H. et al. Undefined behavior: what happened to my code? In Proc. of the Asia-Pacific Workshop on Systems, 2012, pp. 1-7.
[2]. Wang X., Zeldovich N. et al. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In Proc. of the Twenty-Fourth ACM Symposium on Operating Systems Principles, 2013, pp. 260-275.
[3]. C11 Standard ISO/IEC 9899:2011, Programming languages – C. ISO/IEC, 2011.
[4]. STACK. Available at https://css.csail.mit.edu/stack/, accessed 25.08.2021.
[5]. Undefined Behavior Sanitizer. Available at https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, accessed 25.08.2021.
[6]. Address Sanitizer. Available at https://clang.llvm.org/docs/AddressSanitizer.html, accessed 25.08.2021.
[7]. Tis-interpreter. Available at https://github.com/TrustInSoft/tis-interpreter, accessed 25.08.2021.

[8]. Serebryany K. Sanitize, Fuzz, and Harden Your C++ Code. In Proc. of the USENIX ENIGMA Conference, 2006, 35 p.
[9]. Cadar C., Dunbar D., and Engler D. 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, 2008, pp. 209-224.
[10]. Field Experience with Annex K – Bounds Checking Interfaces. Available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1967.htm, accessed 25.08.2021.
[11]. Hathhorn C., Ellison C., and Roşu G. Defining the undefinedness of C. ACM SIGPLAN Notices, vol. 50, issue 6, 2015, pp. 336–345.
[12]. RV-Match. Available at https://runtimeverification.com/match/, accessed 25.08.2021.
[13]. Guth D., Hathhorn C. et al. Rv-match: Practical semantics-based program analysis. Lecture Notes in Computer Science, vol. 9779, 2016, pp. 447-453
[14]. MISRA. Available at https://www.misra.org.uk/, accessed 25.08.2021.
[15]. SEI CERT C Coding Standard. Available at https://wiki.sei.cmu.edu/confluence/display/c, accessed 25.08.2021.
[16]. FORTIFY_SOURCE. Available at https://access.redhat.com/blogs/766093/posts/3606481, accessed 25.08.2021.
[17]. Stack Smashing Protector. Available at https://www.linuxfromscratch.org/hints/downloads/files/ssp.txt, accessed 25.08.2021.
[18]. Address Space Layout Randomization. Available at https://docs.oracle.com/en/operating-systems/oracle-linux/6/security/ol_aslr_sec.html, accessed 25.08.2021.
[19]. [9/10/11/12 Regression] "clobbered by longjmp" warning ignores the data flow. Bug 21161, GCC, 2005. Available at https://gcc.gnu.org/bugzilla/show_bug.cgi?id=21161, accessed 25.08.2021.
[20]. P1705R1 Enumerating Core Undefined Behavior. Available at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1705r1.html, accessed 25.08.2021.
[21]. CRUX. Available at https://crux.nu/, accessed 25.08.2021.
[22]. GNU Go. Available at https://www.gnu.org/software/gnugo/, accessed 25.08.2021.
[23]. SPEC's Benchmarks. Available at https://www.spec.org/benchmarks.html, accessed 25.08.2021.
[24]. The LAME Project. Available at https://lame.sourceforge.io/, accessed 25.08.2021.
[25]. Phoronix Test Suite. Available at https://www.phoronix-test-suite.com/, accessed 25.08.2021.
[26]. The Computer Language Benchmarks Game. Available at https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html, accessed 25.08.2021.
[27]. x264. Available at http://www.videolan.org/developers/x264.html, accessed 25.08.2021.
[28]. Zlib. Available at http://www.zlib.net/, accessed 25.08.2021.
[29]. Song D., Lettner J. et al. SoK: Sanitizing for Security. In Proc. of the IEEE Symposium on Security and Privacy (SP), 2019, pp. 1275-1295.

Информация об авторах / Information about authors

Роман Вячеславович БАЕВ – старший лаборант отдела компиляторных технологий. Научные интересы: компиляторные технологии, оптимизации.

Roman Vyacheslavovich BAEV – Researcher in Compiler Technology department. Research interests: compiler technologies, optimizations.

Леонид Владленович СКВОРЦОВ – стажер-исследователь отдела компиляторных технологий. Научные интересы: компиляторные технологии, оптимизации.

Leonid Vladlenovich SKVORTSOV – Researcher in Compiler Technology department. Research interests: compiler technologies, optimizations.

Евгений Алексеевич КУДРЯШОВ – стажер-исследователь отдела компиляторных технологий. Научные интересы: компиляторные технологии, оптимизации.

Evgeny Alekseevich KUDRYASHOV – Researcher in Compiler Technology department. Research interests: compiler technologies, optimizations.

Рубен Артурович БУЧАЦКИЙ – научный сотрудник отдела компиляторных технологий. Научные интересы: компиляторные технологии, оптимизации.

Ruben Arturovich BUCHATSKIY – Researcher in Compiler Technology department. Research interests: compiler technologies, optimizations.

Роман Александрович ЖУЙКОВ – старший научный сотрудник отдела компиляторных технологий. Научные интересы: компиляторные технологии, оптимизации.

Roman Aleksandrovich ZHUYKOV – Researcher in Compiler Technology department. Research interests: compiler technologies, optimizations