DOI: 10.15514/ISPRAS-2019-1(2)-1



# Разработка безопасного компилятора на основе Clang

[1,2]П.Д. Дунаев, ORCID: 0000-0002-9142-0945, <herrpaulvondonau@outlook.com>
[1,2]А.А. Синкевич, ORCID: 0009-0002-3364-6468, <artsin666@gmail.com>
[1,2]А.М. Гранат, ORCID: 0009-0007-6589-3347, <artemiigranat@gmail.com>
[2]И.А. Батраева, ORCID: 0000-0002-6539-8473, <batraevaia@info.sgu.ru>
[2]С.В. Миронов, ORCID: 0000-0003-3699-5006, <mironovsv@sgu.ru>
[1,3]Н.Ю. Шугалей, ORCID: 0009-0000-9310-8317, <shugaley@ispras.ru>
[1]Институт системного программирования РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25.
[2]Саратовский государственный университет имени Н.Г. Чернышевского, 410012, Россия, Саратов, ул. Астраханская, 83.
[3]Московский физико-технический институт (национальный исследовательский университет), 117303, г. Москва, ул. Керченская, д.1 А, корп. 1.

**Аннотация.** В связи с использованием современными компиляторами C/C++ агрессивных оптимизаций, эксплуатирующих неопределённое поведение, существует потребность в безопасном компиляторе, который не проводит подобные оптимизации, а также предотвращает использование разработчиком небезопасных конструкций. В ИСП РАН был реализован безопасный компилятор на основе GCC, однако часть разработчиков предпочитает GCC Clang, который не лишён проблемы эксплуатации неопределённого поведения. В этой работе рассматриваются возможности Clang по осуществлению безопасной компиляции и описывается реализация безопасного компилятора на его основе. Для созданного безопасного компилятора показывается применимость на практике и оценивается влияние на производительность программ.

**Ключевые слова:** компилятор; уязвимость; неопределенное поведение; clang; llvm; c; c++

# Developing a Clang-Based Safe Compiler

[1,2]P.D. Dunaev, ORCID: 0000-0002-9142-0945, <herrpaulvondonau@outlook.com>
[1,2]A.A. Sinkevich, ORCID: 0009-0002-3364-6468, <artsin666@gmail.com>
[1,2]A.M. Granat, ORCID: 0009-0007-6589-3347, <artemiigranat@gmail.com>
[2]I.A. Batraeva, ORCID: 0000-0002-6539-8473, <batraevaia@info.sgu.ru>
[2]S.V. Mironov, ORCID: 0000-0003-3699-5006, <mironovsv@sgu.ru>
[1,3]N.U. Shugaley, ORCID: 0009-0000-9310-8317, <shugaley@ispras.ru>
[1]Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
[2]Saratov State University, 83 Astrakhanskaya Street, Saratov, 410012, Russia.
[3]Moscow Institute of Physics and Technology (National Research University), 1 A Kerchenskaya st., Moscow, 117303, Russia.

**Abstract.** Due to the use of aggressive optimizations by modern C/C++ compilers that exploit undefined behavior, there is a need for a safe compiler that does not perform such optimizations and prevents developers from using unsafe statements and expressions. Such a safe compiler based on GCC has been developed in ISP RAS, but some developers prefer Clang instead of GCC, which has mainly the same problems of exploiting undefined behavior. This paper examines the capabilities of Clang to perform safe compilation and describes the implementation of a safe compiler based on it. For the created safe compiler, the applicability in practice is shown and the impact on program performance is evaluated.

**Keywords:** compiler; vulnerability; undefined behavior; clang; llvm; c; c++

## *1. Введение*

Последнее время большую популярность в качестве компилятора языка C++ завоевал Clang [1]. Согласно статистике JetBrains [2], в 2023 году компилятор использовали более трети опрошенных, что характеризовало его как второй по популярности компилятор C++. Clang имеет определённые преимущества перед конкурирующими разработками, в том числе перед самым популярным компилятором — GCC [3]. Среди таких преимуществ можно выделить лицензию на основе Apache 2.0 [4], которая позволяет использовать исходный код компилятора для проектов под бóльшим числом лицензий. Существенным преимуществом Clang также является то, что он построен на компиляторной инфраструктуре LLVM [5], ценность которой заключается в том, что при доработке компилятора нередко можно полностью сфокусироваться на преобразованиях промежуточного представления LLVM IR, что было использовано, например, в работах [6-9].

При этом Clang, как и GCC, обладает существенным недостатком — он осуществляет оптимизации, эксплуатирующие небезопасное поведение. Так, в работе [10] приводится ряд недостатков, многие из которых напрямую относятся к Clang. Примером оптимизации, выполняемой рассматриваемым компилятором, является удаление проверок вида `if (1 << X == 0)`, где `X` — целое число. Результат выполнения инструкции побитового сдвига единицы влево, которое в подобных случаях обычно ожидается программистом, на некоторых архитектурах процессоров, таких как PowerPC, равен нулю. Компилятор же считает такую проверку избыточной, так как может доказать, что в отсутствие неопределённого поведения это утверждение всегда ложно, и потому при наличии такового снимает с себя всякие обязательства, если с помощью опций не задано иное. Отключение данной оптимизации с помощью опций невозможно. Таким образом, при компиляции проекта с помощью Clang, необходима дополнительная защита от внесения программистом и компилятором уязвимостей в выходной код.

Потребность в избавлении кода программы от уязвимостей могут помочь удовлетворить различные методы, такие как, например, статический и динамический анализ, а также полноценное тестирование и использование стандартов безопасного кодирования. Авторы статьи [11] показывают, что существуют ситуации, когда ни один из данных способов не применим для решения проблемы устранения создаваемых компилятором уязвимостей. Авторы предлагают альтернативный путь решения — безопасный компилятор, в который встроена функциональность, предотвращающая уязвимости, вносимые агрессивными оптимизациями, и предупреждающая об уязвимостях, вносимых программистом. В статье описывается безопасный компилятор [12], основанный на GCC. Однако GCC не может выступать в качестве замены Clang, поскольку Clang не является полностью совместимым с GCC [13], и, по причине ряда вышеописанных преимуществ, вероятно, не все разработчики будут готовы отказаться от Clang. Таким образом, становится актуальной проблема безопасной компиляции посредством Clang.

## 2. Концепция безопасного компилятора

В статье [11] описана концепция безопасного компилятора. Безопасный компилятор, чтобы считаться таковым, должен удовлетворять следующим требованиям:

- Компилятор не может вносить во время выполнения оптимизаций уязвимости в генерируемый код;
- Компилятор обязан не удалять код, исходя из предположения об отсутствии неопределённого поведения, не сильно замедляя при этом работу выходной программы;
- Правки, требуемые для успешной компиляции исходного кода, минимально возможны;
- Компилятор не предоставляет возможность отключения опций, контролирующих выполнение первых двух требований.

Такой компилятор может работать в качестве замены ранее используемого, однако помимо основного своего преимущества — профилактики уязвимостей — компилятор будет наделён и недостатками в виде замедления работы компилируемого приложения и необходимости, пусть и минимальной, модификации исходного кода, что в ряде случаев может быть затруднительно или невозможно.

Детализация требований к безопасному компилятору приведена в стандарте [14]. Выделяется три класса требований, каждый из которых характеризуется строгостью механизмов предотвращения уязвимостей и скоростью работы генерируемых программ. Требованиям стандарта может соответствовать как написанный с нуля компилятор, так и доработанный либо правильно сконфигурированный существующий. Поскольку в данной работе в качестве целевой аудитории приняты пользователи Clang, вариант написания нового компилятора не рассматривается; возможности конфигурации Clang для выполнения требований к безопасному компилятору рассмотрены в разделе 3, в разделе 4 описаны работы, проведённые для доработки Clang до безопасного компилятора, а в разделе 5 приведены результаты тестирования безопасного Clang на реальных приложениях.

## 3. Соответствие возможностей Clang требованиям к безопасному компилятору

Безопасный компилятор третьего класса должен выполнять только безопасные преобразования исходного и машинного кода. Например, компилятор может преобразовать условие `N + 1 > N` в `true`, так как принимает за истину то, что неопределённого поведения произойти не может (в данном случае — целочисленного переполнения), и третий класс безопасности должен гарантировать защиту от преобразований такого вида. Ограничение на преобразования и соответствующие им опции Clang представлены в таблице 1.

Помимо этого, компилятор третьего класса безопасности должен включать механизмы повышенной защищенности. Требуемые механизмы и соответствующие им опции Clang представлены в таблице 2.

Также одной из задач третьего класса безопасности является выдача предупреждений в ходе сборки программы в некоторых случаях неопределенного поведения. Опции Clang, включающие выдачу соответствующих предупреждений, представлены в таблице 3.

*Табл. 1. Реализация требований 3 класса по отключению небезопасных преобразований в Clang*
*Table 1. Implementation of class 3 requirements for disabling unsafe transformations in Clang*

| Пункт стандарта | Требование | Опция |
|---|---|---|
| 5.2.1 а | Отключение преобразований, связанных с целочисленным переполнением | `-fwrapv` |
| 5.2.1 б | Отключение преобразований, связанных с тем фактом, что | `-fno-strict-` |

| Пункт стандарта | Требование | Опция |
|---|---|---|
| | значения указателей разных типов могут совпадать | `aliasing` |
| 5.2.1 в | Отключение преобразований, связанных с разыменованием нулевых указателей | `-fno-delete-null-pointer-checks` |
| 5.2.1 г | Отключение преобразований, связанных с делением на ноль и взятием нулевого остатка | — |
| 5.2.1 д | Отключение преобразований, связанных со значениями аргументов побитового сдвига | — |

*Табл. 2. Реализация требований 3 класса по включению механизмов повышенной защищённости в Clang*

*Table 2. Implementation of class 3 requirements for enabling increased safety mechanisms in Clang*

| Пункт стандарта | Требование | Опция |
|---|---|---|
| 5.2.2 а | Защита от переполнения буфера постоянного размера при вызове функций стандартной библиотеки | `-D_FORTIFY_SOURCE=2` с оговоркой: при возникновении ошибки, данные функции выводят избыточную информацию, в то время как, согласно требованиям, программа должна немедленно завершаться |
| 5.2.2 б | Механизм контроля за целостностью стека | `-fstack-protector-strong` |
| 5.2.2 в | Механизм рандомизации размещения кода в адресном пространстве | `-fpic/-fPIC/-fPIE` |
| 5.2.2 г,д | Запрет на замену вызовов некоторых функций форматированного вывода и работы с памятью на эквивалентные последовательности машинных инструкций | `-fno-builtin-*`, при этом некоторые функции, работающие с widechar-строками, не имеют встроенных аналогов в Clang, поэтому для них это требование выполняется автоматически |
| 5.2.6 | Опциональные механизмы (например, контроль за целостностью потока управления) | Существует решение в виде `-fsanitize=cfi`, противоречащее, однако, логике третьего класса, поскольку способно существенно замедлить программу |

*Табл. 3. Реализация требований 3 класса по включению предупреждений о небезопасных конструкциях в Clang*

*Table 3. Implementation of class 3 requirements for enabling warnings about unsafe statements in Clang*

| Пункт стандарта | Требование | Опция |
|---|---|---|
| 5.2.3 а | Затирание переменной, размещённой в автоматической памяти, при вызове `longjmp` | — |
| 5.2.3 б | Чтение или запись по некорректному индексу элемента массива | `-Warray-bounds` и `-Warray-bounds-` |

| Пункт стандарта | Требование | Опция |
|---|---|---|
| | | `pointer-arithmetic` |
| 5.2.3 в | Целочисленное деление или взятие остатка от целого числа с делителем, равным нулю | `-Wdivision-by-zero` |
| 5.2.3 г | Операция побитового сдвига со вторым аргументом меньше нуля или больше ширины типа сдвигаемого значения | `-Wshift-count-negative/-Wshift-count-overflow` |

Основными задачами безопасного компилятора второго класса являются предотвращение уязвимостей, связанных с некорректной работой с памятью, и недопущение использования неопределённых конструкций — тех же, о которых предупреждает компилятор третьего класса — путём остановки компиляции с ошибкой. Clang реализует лишь некоторые из них. Полный список требований и опций, с помощью которых они реализованы, указан в таблице 4.

*Табл. 4. Реализация требований 2 класса в Clang*

*Table 4. Implementation of class 2 requirements in Clang*

| Пункт стандарта | Требование | Опция |
|---|---|---|
| 5.3.1 а | Сохранение побочных эффектов записи в память | — |
| 5.3.1 б | Автоматическая инициализация переменных нулями | `-ftrivial-auto-var-init=zero` |
| 5.3.1 доп. а | Запрет оптимизаций побитовых сдвигов, если величина сдвига может оказаться отрицательной или больше или равна размеру типа | — |
| 5.3.1 доп. б | Использование операций с векторными машинными регистрами, не требующих выравнивания данных | — |
| 5.3.1 доп. в | Запрет оптимизаций адресной арифметики по информации о размерах объектов | Не требуется |
| 5.3.2 а | Остановка компиляции с ошибкой для предупреждений 3 класса | `-Werror=*` вместо `-W*`, см. табл. 3 |
| 5.3.2 б | Запрет использования функции `gets` | `-Werror=deprecated-declarations` |

Одной из основных задач безопасного компилятора первого класса является динамический контроль неопределённых конструкций. При этом от компилятора требуется включать в выходной файл машинный код, который предотвращает ошибочное выполнение неопределённых конструкций во время работы программы путём её аварийной остановки. Эта возможность в Clang реализована с помощью встроенного инструмента UndefinedBehaviourSanitizer (UBSan) [15], использование которого достигается с помощью указания в аргументах командной строки компиляции различных опций вида `-fsanitize=*`, где `*` — идентификатор проверки, определяющей тот или иной вид неопределённых конструкций. UBSan выполняет большую часть видов проверок, которые должен осуществлять безопасный компилятор (см. таблицу 5).

*Табл. 5. Реализация требований 1 класса в Clang*

*Table 5. Implementation of class 1 requirements in Clang*

| Пункт стандарта | Идентификатор | Покрываемое требование: проверяет, что… |
|---|---|---|
| 5.4.3 а | `bool` | Значение типа `bool` равно `0` или `1`. |
| 5.4.3 б | `float-cast-overflow` | При преобразовании значения `float` в `int` не происходит переполнения. Не проверяются преобразования между вещественными типами. |
| 5.4.3 в | `shift` | Правый операнд сдвига неотрицателен и меньше ширины типа. |
| 5.4.3 г | `signed-integer-overflow` | Не происходит знакового переполнения. |
| 5.4.3 д | `alignment` | Чтение или запись совершаются только по указателю, чей адрес выровнен по размеру операнда. |
| 5.4.3 е | `null` | Нет использования нулевого указателя. Не проверяется непрямой вызов функции по нулевому указателю. |
| 5.4.3 ж | `bounds` | Чтение или запись осуществляется по индексу, не выходящему за пределы массива. |
| 5.4.3 и | `pointer-overflow` | Нет переполнения типа указателя. |
| 5.4.3 к | `function` | При непрямом вызове сигнатура функции соответствует типу указателя на неё. Работает только для C++ и x86(-64). |
| 5.4.3 л | `return` | Вызов функции завершается оператором `return`. Работает только для C++. |
| 5.4.3 м | `builtin` | Во встроенные функции передаются корректные параметры. |
| 5.4.3 н | `unreachable` | Не передаётся управление коду, помеченному как недостижимый. |
| 5.4.3 п | `integer-divide-by-zero` | Нет деления на ноль или взятия остатка от нуля. |
| 5.4.3 р | `vla-bound` | Массив, выделяемый в автоматической памяти, имеет положительный размер. |

Другой задачей безопасного компилятора является управление распределением автоматической и статической памяти. В рамках этой задачи компилятор первого класса должен поддерживать возможность динамической компоновки программы, при которой при каждом запуске программы функции будут расположены в памяти в случайном порядке. Если полноценная реализация этого механизма невозможна (чему может препятствовать загрузчик операционной системы) или нецелесообразна (как в случае компиляции ядра операционной системы) распределение должно быть статическим — уникальным для каждого процесса компиляции. Clang не поддерживает статическое случайное распределение памяти, но может поддерживать динамическое распределение при использовании компоновщика и динамического загрузчика, в которых реализована эта возможность.

Необходимо заметить, что безопасный компилятор второго класса заимствует также часть требований третьего класса, а компилятор первого класса — все требования второго.

Таким образом, не существует конфигурации Clang, которая удовлетворяла бы требованиям хотя бы одного из классов безопасной компиляции, однако компилятор предоставляет ряд возможностей для предотвращения небезопасных оптимизаций и выполнения небезопасных конструкций, что может быть использовано, в частности, при разработке безопасного компилятора на его основе.

## 4. Реализация безопасного компилятора

Реализованный безопасный компилятор разработан на базе Clang 16.0.6. Были добавлены опции `-Safe3`, `-Safe2`, `-Safe1`, включающие доступные в Clang и созданные в этой работе опции соответствующих классов безопасности, а для удобного управления функциональностью опций был разработан предметно-ориентированный язык на основе TableGen [16], позволяющий описать включение опций в следующем формате:

```
defm fno_strict_aliasing : Force<"-relaxed-aliasing", 1, 3>;
defm d_fortify_source_2 : Force<"-D_FORTIFY_SOURCE=2", 3, 3>;
```

Для 3 класса были реализованы следующие опции:

- `-fkeep-oversized-shifts`: предотвращает оптимизацию побитовых сдвигов в случаях, когда второй аргумент оператора сдвига меньше нуля или больше или равен ширине типа. Эта опция реализована заменой инструкций сдвига LLVM IR вызовами новых intrinsic-функций. Проходы `InstCombine` и `SCCP` дополнены оптимизациями этих вызовов, осуществляемыми только в тех случаях, когда компилятор способен доказать, что второй аргумент неотрицателен и строго меньше ширины типа. В противном случае вызовы intrinsic-функций раскрываются (заменяются на соответствующие инструкции) после всех оптимизационных проходов, избегая, таким образом, оптимизаций.
- Заголовочные файлы, содержащие fortified-версии функций стандартной библиотеки, и которые используются в Alpine Linux и безопасном компиляторе SAFEC, были доработаны для поддержки Clang. Благодаря использованию заголовочных файлов, включённых в компилятор, поддерживается и стандартная библиотека glibc, и musl. Кроме опции `-D_FORTIFY_SOURCE=2`, была добавлена поддержка `-D_FORTIFY_SOURCE=3` (включаемая в `-Safe2`), проверяющая не только вызовы с объектами константного размера, но и с теми, для которых можно во время компиляции составить выражение вычисления размера. Также с помощью этих заголовочных файлов было реализовано немедленное завершение программы при ошибке во время выполнения fortified-функций, предотвращена замена функций на встроенные в Clang аналоги, и добавлено предупреждение (ошибка в `-Safe2`) при использовании функции `gets`.
- Вместо предупреждения о затирании переменной, размещённой в автоматической памяти, при вызове `longjmp`, была реализована опция `-fforce-volatile-before-setjmp`, отмечающая все локальные переменные, доступные в момент вызова `setjmp`, как `volatile`, что не позволяет компилятору разместить эти переменные на регистрах и предотвращает их затирание после вызова `longjmp`. Эта опция реализована как проход в начале оптимизационного конвейера, работающего с промежуточным представлением LLVM IR. Этот проход обнаруживает уязвимые переменные (выделения на стеке) и помечает все использующие их инструкции как `volatile`.

Для 2 класса были реализованы следующие опции:

- Для сохранения побочных эффектов записи в память были созданы опции `-mllvm -preserve-memory-writes-{earlycse,instcombine,memcpyopt,dse}`, предотвращающие DSE (Dead Store Elimination) в нескольких проходах оптимизационного конвейера. Благодаря этим опциям сохраняется, например, очистка памяти, содержащей чувствительные данные. В проходе `EarlyCSE`, устраняющем тривиально избыточные инструкции, опция отключает удаление последовательных записей в тот же участок памяти без чтения между ними. В `InstCombine`, выполняющем объединение и удаление инструкций, отключается аналогичная оптимизация. В проходе `MemCpyOptimizer`, оптимизирующем такие инструкции работы с памятью, как `memset` и `memcpy`, отключается объединение пересекающихся записей в память в один `memset`. Наконец, в проходе `DeadStoreElimination`, выполняющем основную работу по

оптимизации избыточных записей, вместо их удаления производится установка флага `volatile`, чтобы эти инструкции не могли быть оптимизированы следующими проходами.

- Опция `-fassume-unaligned` включает в начало оптимизационного конвейера проход, удаляющий выравнивание у инструкций `load` и `store`, а также у аргументов-указателей в вызовах функций. Благодаря этому вместо векторных инструкций, ожидающих выравненную память, генерируются инструкции для невыравненной памяти и предотвращаются аварийные завершения программ в случаях, когда используемый участок памяти имел некорректное выравнивание.

Для 1 класса были реализованы следующие опции:

- Так как проверка `float-cast-overflow` в UndefinedBehaviorSanitizer проверяет наличие переполнения только при преобразовании из вещественных типов с плавающей запятой в целочисленные типы, была создана опция `-fsanitize=float-to-float-cast-overflow`, проверяющая преобразования между вещественными типами. Реализация основана на старой версии проверки `float-cast-overflow`, поведение которой было изменено в Clang 9 из-за того, что такой вид переполнения определён стандартом IEEE 754 [17].
- Опция `-fsanitize=null-call` проверяет, что при непрямом вызове функции не используется нулевой указатель. Такая проверка отсутствует в `-fsanitize=null` из UBSan.
- В Clang 16 опция `-fsanitize=function`, проверяющая соответствие формального типа функции и фактического типа указателя, поддерживает только C++ и архитектуру x86(-64), поэтому из Clang 17 были портированы улучшения, позволяющие использовать её для C и других архитектур. Основным изменением является использование хешей типов вместо RTTI (Run-Time Type Information), доступного только для C++.
- Стандартная опция `-fsanitize=return` проверяет наличие операции возврата при выходе из функции, имеющей возвращаемое значение, но только для C++, так как в C неопределённым поведением считается не отсутствие возврата, а использование значения, возвращаемого такой функцией. Чтобы сделать поведение проверки единообразным, была реализована опция `-fsanitize=return-c`, работающая для C так же, как и для C++.
- Для поддержки уникального распределения статической памяти программы на этапе компиляции были добавлены опции `-frandom-func-reorder` и `-frandom-func-and-globals-reorder`, перемешивающие только функции или функции и глобальные переменные соответственно в каждой единице компиляции. Эти опции включают в конец оптимизационного конвейера, работающего с LLVM IR, проход, который задаёт случайный порядок функций и глобальных переменных на основе содержимого модуля и числа, задаваемого опцией `-mllvm -rng-seed`.
- Для рандомизации автоматической памяти создана опция `-floc-var-per`, перемешивающая локальные переменные (точнее, выделения на стеке константного размера) в случайном порядке. Эта функциональность реализована в том же проходе, который используется для предыдущих двух опций. Также поддерживается опция `-fadd-loc-var`, задающая количество локальных переменных, которые добавляются при перемешивании. Без неё при использовании `-floc-var-per` добавляется небольшое случайное количество переменных для большей случайности автоматической памяти.

# *5. Результаты*

## 5.1. Корректность

Разработанный безопасный компилятор на основе Clang успешно проходит все тесты из набора, созданного для проверки корректности безопасного компилятора SAFEC [11] и его соответствия стандарту. Этот набор включает в себя тесты, проверяющие наличие вывода требуемых диагностик, тесты, проверяющие срабатывание динамических проверок во время выполнения, и тесты, проверяющие результат кодогенерации.

## 5.2. Исследование производительности

Производительность программ, скомпилированных безопасным компилятором, оценивалась с помощью тех же 5 тестов, что и для SAFEC:

- воспроизведение партии в го с помощью GNU Go 3.8;
- перекодирование файлов из формата WAV в MP3 с помощью LAME 3.100;
- выполнение теста fannkuch из The Computer Language Benchmarks Game;
- перекодирование файлов из формата YUV в MKV с помощью x264 (x264-snapshot-20190407-2245-stable);
- сжатие текстового файла с помощью zlib 1.2.11.

*Табл. 6. Результаты измерения производительности*
*Table 6. Performance evaluation results*

| Тест | Baseline | `-Safe3` | `-Safe3` замедл. | `-Safe2` | `-Safe2` замедл. | `-Safe1` | `-Safe1` замедл. |
|---|---|---|---|---|---|---|---|
| GNU Go | 3.93 с | 4.03 с | 2.54% | 4.26 с | 8.12% | 6.66 с | 69.04% |
| LAME | 5.21 с | 5.27 с | 1.15% | 4.90 с | -5.82% | 13.19 с | 153.40% |
| fannkuch | 2.24 с | 2.10 с | -6.25% | 2.08 с | -7.14% | 2.72 с | 21.43% |
| x264 | 1.69 с | 1.81 с | 7.42% | 1.79 с | 5.20% | 6.32 с | 274.74% |
| zlib | 1.54 с | 1.63 с | 5.88% | 1.62 с | 5.87% | 2.40 с | 55.60% |

В табл. 6 приведены результаты измерения времени выполнения тестов на компьютере с процессором AMD Ryzen™ 5 4600H (архитектура x86-64) и ОС Manjaro Linux 23.1.4. Столбец Baseline соответствует запуску компилятора с опцией `-O2`, а столбцы `-Safe3`, `-Safe2`, `-Safe1` соответствуют запуску с уровнем оптимизации `-O2` и соответствующим классом защиты. Каждое значение вычислялось как округлённое до 0.01 с среднее по 5 запускам. Также для каждого уровня безопасности указано замедление относительно базового времени.

Из представленных данных следует, что при использовании 3 или 2 класса защиты замедление не превышает 10%, при этом программы, скомпилированные с 2 уровнем безопасности иногда оказываются быстрее. Это может происходить из-за того, что в 2 классе, в отличие от 3 класса, отсутствует запрет на замену вызовов функций работы с памятью из стандартной библиотеки на эквивалентные последовательности машинных инструкций. При использовании 1 класса защиты замедление составляет от 21% до 275%, что в случае x264 превышает допустимые 200%.

Существенное замедление работы x264 можно объяснить добавлением санитайзером `pointer-overflow` большого количества проверок переполнения указателей — без него замедление составляет 165%. Также было обнаружено, что базовая версия x264, скомпилированная Clang, выполнялась на 64% быстрее чем та, что была скомпилирована SAFEC 11.4.0, при этом с 1 уровнем безопасности время работы программ приблизительно совпадало, а значит при сравнении 1 класса безопасности Clang с базовой версией GCC замедление составит менее 200%.

Кроме того, в результате выполнения тестов выяснилось, что санитайзер `pointer-overflow` в Clang, в отличие от GCC и основанного на нём SAFEC, находит в GNU Go переполнение при добавлении беззнакового числа к указателю. Это известное ограничение GCC [18].

## 5.3. Сборка дистрибутива Linux

Кроме тестирования нескольких приложений отдельно с помощью безопасного Clang, была произведена оценка применимости безопасного компилятора с помощью сборки дистрибутива Linux. Для этой задачи был выбран Alpine Linux 3.18 [19] — дистрибутив, ориентированный на легковесность и безопасность, использующий musl, BusyBox и OpenRC. Так как в дистрибутиве для сборки из исходных кодов по умолчанию используется GCC, то в сборочный мета-пакет `build-base` был добавлен Clang как зависимость и произведена замена `/usr/bin/gcc`, `/usr/bin/cc` и подобных файлов на символические ссылки на Clang. Также в пакеты `clang16` и `llvm16` были добавлены патчи безопасного компилятора. В результате была выполнена сборка каждого пакета из репозиториев `main` и `community` со всеми классами безопасности: от небезопасного режима до 1 класса.

*Табл. 7. Результаты сборки пакетов Alpine Linux*
*Table 7. Alpine Linux package build results*

| Класс | Успешно собрано | Ошибки сборки |
|---|---|---|
| Baseline | 3587 | 910 |
| `-Safe3` | 3581 | 6 |
| `-Safe2` | 3500 | 81 |
| `-Safe1` | 3456 | 44 |

В таблице 7 для каждого класса безопасности приведено количество пакетов, которые удалось собрать, и количество пакетов, чья сборка завершилась ошибкой. Большое количество пакетов, не собранных Clang в небезопасном режиме, во многих случаях объясняется невозможностью загрузить исходный код пакетов, а также несовместимостью Clang с GCC из-за включения по умолчанию некоторых предупреждений в качестве ошибок. Также стоит отметить, что приблизительно треть от всех пакетов (2382 из 6879) не использует компилятор C/C++, поэтому в работе они не рассматриваются.

На 3 уровне безопасности не удалось собрать всего 6 пакетов — в 2 случаях возникла ошибка из-за попытки установки `_FORTIFY_SOURCE` в 0 при наличии `-Werror`. Ещё в одном пакете выполнялось удаление ключевого слова `const` с помощью `#define const`, что мешает компиляции заголовочных файлов fortified-функций. Также в одном пакете, компилируемом с `-Werror`, присутствовало предупреждение `-Warray-bounds-pointer-arithmetic`. Ещё два пакета оказались несовместимы с механизмом контроля целостности стека. Так как в 2 классе безопасности предупреждения из 3 класса становятся ошибками, то на этом уровне из-за `-Warray-bounds-pointer-arithmetic` не скомпилировались 68 пакетов, ещё 6 — из-за `-Warray-bounds`, и 7 из-за `-Wshift-count-overflow`.

В 1 класс безопасности включаются санитайзеры, из-за чего могут аварийно завершиться программы, скомпилированные и запущенные во время сборки пакетов. По этой причине не удалось собрать 39 пакетов. Также сборка 2 пакетов завершилась ошибками из-за несовместимости `-fsanitize=function` с WebAssembly. Ещё один пакет не удалось собрать из-за `-Werror=shift-count-overflow`. Кроме того, при сборке пакетов `community/cabextract` и `community/libu2f-server` была обнаружена ошибка в исходном Clang 16.0.6, приводящая к аварийному завершению компилятора. Выяснилось, что она была исправлена в Clang 17.

В этой работе проверялось только то, что пакеты дистрибутива успешно собираются, так как тестирование работоспособности всех программ потребовало бы слишком больших затрат времени. Было показано, что 96.3% пакетов, собираемых небезопасным Clang, могут быть

собраны и безопасной версией с 1 классом безопасности. Но необходимо дальнейшее тестирование, так как может оказаться, что многие успешно собранные программы не будут работать, например, из-за наличия в них неопределённого поведения, обнаруживаемого санитайзерами.

### *6. Заключение*

В рамках данной работы был реализован безопасный компилятор на основе Clang. В работе была рассмотрена концепция безопасного компилятора применительно к исследуемому. Были выделены компоненты, реализующие указанные возможности, а также рассмотрены требования, которым данный компилятор не удовлетворяет. Было обосновано решение о разработке безопасного компилятора на основе Clang. Были реализованы и описаны все недостающие компоненты. С помощью созданного безопасного компилятора удалось собрать большую часть пакетов дистрибутива ОС Alpine Linux. Было показано, что производительность программ, собранных безопасным компилятором, почти во всех случаях удовлетворяет требованиям.

## Список литературы / References


[1]. Clang, Available at: https://clang.llvm.org/, accessed 24.04.2024.

[2]. Jetbrains Developer Ecosystem: C++, Available at: https://www.jetbrains.com/lp/devecosystem-2023/cpp/, accessed 24.04.2024.

[3]. GCC, Available at: https://gcc.gnu.org/, accessed 24.04.2024.

[4]. LLVM Developer Policy, Available at: https://llvm.org/docs/DeveloperPolicy.html, accessed 24.04.2024.

[5]. LLVM, Available at: https://llvm.org/, accessed 24.04.2024.

[6]. Скворцов Л.В., Баев Р.В., Долгорукова К.Ю., Шарыгин Е.Ю. Разработка компилятора для стековой процессорной архитектуры TF16 на основе LLVM. Труды ИСП РАН, том 33, вып. 5, 2021 г., стр. 137-154. DOI: 10.15514/ISPRAS–2021–33(5)–8./ Skvortsov L.V., Baev R.V., Dolgorukova K.Y., Sharygin E.Y. Developing an LLVM-based compiler for stack based TF16 processor architecture. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 5, 2021, pp. 137-154 (in Russian). DOI: 10.15514/ISPRAS–2021–33(5)–8.

[7]. Мельник Д., Курмангалеев Ш., Аветисян А., Белеванцев А., Плотников Д., Варданян М. Оптимизация приложений для заданных статических компиляторов и целевых архитектур: методы и инструменты. Труды ИСП РАН, том 26, вып. 1, 2014 г., стр. 343-356. DOI: 10.15514/ISPRAS-2014-26(1)-13./ Melnik D., Kurmangaleev S., Avetisyan A., Belevantsev A., Plotnikov D., Vardanyan M. Optimizing programs for given hardware architectures with static compilation: methods and tools. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 1, 2014, pp. 343-356 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-13.

[8]. Иванников В., Курмангалеев Ш., Белеванцев А., Нурмухаметов А., Савченко В., Матевосян Р., Аветисян А. Реализация запутывающих преобразований в компиляторной инфраструктуре LLVM. Труды ИСП РАН, том 26, вып. 1, 2014 г., стр. 327-342. DOI: 10.15514/ISPRAS-2014-26(1)-12./ Ivannikov V., Kurmangaleev S., Belevantsev A., Nurmukhametov A., Savchenko V., Matevosyan H., Avetisyan A. Implementing Obfuscating Transformations in the LLVM Compiler Infrastructure. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 1, 2014, pp. 327-342 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-12.

[9]. Гайсарян С.С., Курмангалеев Ш.Ф., Долгорукова К.Ю., Савченко В.В., Саргсян С.С. Применение метода двухфазной компиляции на основе LLVM для распространения приложений с использованием облачного хранилища. Труды ИСП РАН, том 26, вып. 1, 2014 г., стр. 315-326. DOI: 10.15514/ISPRAS-2014-26(1)-11./ Gaissaryan S., Kurmangaleev S., Dolgorukova K., Savchenko V., Sargsyan S. Applying two-stage LLVM-based compilation approach to application deployment via cloud storage. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 1, 2014, pp. 315-326 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-11.

[10]. 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.

[11]. 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./ 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.

[12]. Безопасный компилятор SAFEC. Доступно по ссылке: https://www.ispras.ru/technologies/safecomp/, доступ осуществлён 24.04.2024.

[13]. Clang: Language Compatibility, Available at: https://clang.llvm.org/compatibility.html, accessed 24.04.2024.

[14]. ГОСТ Р 71206-2024 «Защита информации. Разработка безопасного программного обеспечения. Безопасный компилятор языков С/С++. Общие требования». М., Российский институт стандартизации, 2024, 20 с.

[15]. UndefinedBehaviourSanitizer, Available at: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, accessed 24.04.2024.

[16]. TableGen Overview, Available at: https://llvm.org/docs/TableGen/, accessed 24.04.2024.

[17]. IEEE 754-2019, Standard for Floating-Point Arithmetic, 2019. pp. 1-84. DOI: 10.1109/IEEESTD.2019.8766229.

[18]. Missing pointer overflow detection with -fsanitize=pointer-overflow, Available at: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=82079, accessed 24.04.2024.

[19]. Alpine Linux, Available at: https://www.alpinelinux.org/, accessed 24.04.2024.

## *Информация об авторах / Information about authors*

Павел Дмитриевич ДУНАЕВ — Старший лаборант Института системного программирования им. В.П. Иванникова Российской академии наук, студент 2 курса магистратуры Саратовского государственного университета. Сфера научных интересов: компиляторы, операционные системы, дискретная математика.

Pavel Dmitrievich DUNAEV — Senior Laboratory technician at Ivannikov Institute for System Programming of the Russian Academy of Sciences, 2nd year Master’s student at Saratov State University. Research interests: compilers, operating systems, discrete mathematics.

Артем Александрович СИНКЕВИЧ — Старший лаборант Института системного программирования им. В.П. Иванникова Российской академии наук, студент 1 курса магистратуры Саратовского государственного университета. Сфера научных интересов: компиляторы, дискретная математика, нейронные сети.

Artem Aleksandrovich SINKEVICH — Senior Laboratory technician at Ivannikov Institute for System Programming of the Russian Academy of Sciences, 1st year Master’s student at Saratov State University. Research interests: compilers, discrete mathematics, neural networks.

Артемий Максимович ГРАНАТ — Лаборант Института системного программирования им. В.П. Иванникова Российской академии наук, студент 4 курса бакалавриата Саратовского государственного университета. Сфера научных интересов: компиляторы, операционные системы, компьютерные сети.

Artemiy Maksimovich GRANAT — Laboratory technician at Ivannikov Institute for System Programming of the Russian Academy of Sciences, 4th year Bachelor’s student at Saratov State University. Research interests: compilers, operating systems, computer networks.

Инна Александровна БАТРАЕВА — кандидат физико-математических наук, доцент, заведующая кафедрой технологий программирования. Сфера научных интересов: дискретная математика, теория автоматов, теория формальных языков и грамматик, информационные системы в теоретической и прикладной лингвистике.

Inna Aleksandrovna BATRAEVA — Candidate of Science in Physics and Mathematics, Associate Professor, Head of the Department of Programming Technologies. Research interests: discrete mathematics, automata theory, theory of formal languages and grammars, information systems in theoretical and applied linguistics.

Сергей Владимирович МИРОНОВ — кандидат физико-математических наук, доцент, декан факультета компьютерных наук и информационных технологий. Сфера научных интересов:

методы сокращения диагностической информации с использованием словарей неисправностей, формальные языки и грамматики, функциональное программирование.

Sergei Vladimirovich MIRONOV — Candidate of Science in Physics and Mathematics, Associate Professor, Dean of the Faculty of Computer Science and Information Technologies. Research interests: methods of diagnostic information compression using fault dictionaries, formal languages and grammars, functional programming.

Никита Юрьевич ШУГАЛЕЙ — Старший лаборант Института системного программирования им. В.П. Иванникова Российской академии наук, студент 1 курса магистратуры Московского Физико-технического Института Физтех-школы Радиотехники и Компьютерных технологий по направлению Прикладные Физика и Математика.

Nikita Yurievich SHUGALEY — Senior Laboratory technician at Ivannikov Institute for System Programming of the Russian Academy of Sciences, 1st year Master's student at the Moscow Institute of Physics and Technology, Phystech School of Radio Engineering and Computer Technology, field of Applied Physics and Mathematics.