МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра математической кибернетики и компьютерных наук

РАЗРАБОТКА МЕТОДОВ ДИАГНОСТИКИ НЕОПРЕДЕЛЕННОГО ПОВЕДЕНИЯ В БЕЗОПАСНОМ КОМПИЛЯТОРЕ ЯЗЫКОВ C/C++

АВТОРЕФЕРАТ МАГИСТЕРСКОЙ РАБОТЫ

студента 2 курса 273 группы
направления 02.04.03 — Математическое обеспечение и администрирование
информационных систем
факультета КНиИТ
Синкевича Артема Александровича

Научный руководитель
зав. к. техн. пр., к. ф.-м. н., доцент _____ И. А. Батраева

Заведующий кафедрой
к. ф.-м. н., доцент _____ С. В. Миронов

Саратов 2025

ВВЕДЕНИЕ

Актуальность темы. Последнее время большую популярность в качестве компилятора языка C++ завоевал Clang. Согласно статистике JetBrains, 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. Среди них можно выделить лицензию на основе Apache 2.0, 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, ценность которой заключается в том, что при доработке компилятора нередко можно полностью сфокусироваться на преобразованиях промежуточного представления LLVM IR.

При этом Clang, как и GCC, обладает особенностью — он осуществляет оптимизации, эксплуатирующие небезопасное поведение, и многие из которых невозможно отключить с помощью опций. Так, в работе [1] приводится ряд уязвимостей, проявляющихся при оптимизациях, многие из которых включены в Clang. 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 программирования. Авторы статьи [2] 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, основанный на GCC. Однако GCC не может выступать в качестве замены Clang, поскольку Clang не является полностью совместимым с GCC, и, по причине ряда вышеописанных преимуществ, вероятно, не все разработчики будут готовы отказаться от Clang. Таким образом, становится актуальной проблема

безопасной компиляции посредством Clang, которой и посвящена данная работа.

Чтобы компилятор считался безопасным, он должен соответствовать ГОСТ Р 71206-2024 [3]. В этом стандарте выделяется три класса требований, каждый из которых характеризуется строгостью механизмов предотвращения уязвимостей и скоростью работы генерируемых программ.

Цель магистерской работы — разработка методов диагностики неопределённого поведения в программах на языках C/C++ и их реализация в безопасном компиляторе на основе Clang, обеспечивающего выявление и предотвращение такого поведения в соответствии с требованиями по разработке безопасного программного обеспечения.

Поставленная цель определила **следующие задачи**:

- изучение методов предотвращения уязвимостей в программах;
- проверка соответствия возможностей Clang стандарту безопасного компилятора;
- реализация опций, включающих существующие функции для частичного соответствия классам безопасности;
- доработка Clang для соответствия 3, 2 и 1 классам безопасности;
- тестирование корректности работы и производительности безопасного Clang.

Структура и объём работы. Магистерская работа состоит из введения, двух разделов, заключения, списка использованных источников, 2 приложений и цифрового носителя. Общий объем работы — 89 страниц, из них 86 страниц — основное содержание, включая 7 таблиц, список использованных источников информации содержит 51 наименование.

1 Теоретическая часть

В современном мире программные системы играют неотъемлемую роль в различных аспектах жизни — от личного общения и развлечений до управления критически важной инфраструктурой и обеспечения национальной безопасности. По мере того, как эти системы становятся всё более сложными, обеспечение их надёжности становится первостепенной задачей как для разработчиков, так и для организаций и правительств. Для предотвращения появления уязвимостей в программах и их устранения может использоваться комплекс мер, реализуемый в процессах жизненного цикла программного обеспечения. Общие требования к содержанию работ по созданию безопасного (защищённого) ПО описаны в стандарте [4].

В частности, согласно ГОСТ Р 56939-2016, разработчик должен использовать набор правил и рекомендаций, направленных на устранение недостатков программы (стандарты безопасного программирования), пользоваться инструментальными средствами (в том числе компиляторами) для обнаружения потенциальных уязвимостей с помощью статического и динамического анализа кода, проводить тестирование ПО для определения соответствия требованиям безопасности.

1.1 Уязвимости

Уязвимости программного обеспечения — это недостатки в системе, которые позволяют злоумышленнику обойти принятые меры безопасности. Ошибки не являются чем-то новым в мире ПО; большая и сложная программная система может содержать большое количество ошибок, а ошибки безопасности могут быть использованы злоумышленниками для нанесения ущерба или получения выгоды. Проблемы безопасности, связанные с уязвимостями ПО, могут стать особенно серьёзными в случае, например, компрометации информации личного характера.

Программы на C и C++ особенно подвержены ошибкам работы с памятью, таким как переполнение буфера, потому что такое поведение считается неопределённым и зачастую не обнаруживается ни во время компиляции, ни во время выполнения программы. По той же причине возникают ошибки работы с указателями, числами, строками, проблемы инициализации и очистки памяти, управления ресурсами [5].

1.2 Стандарты безопасного программирования

Для предотвращения уязвимостей, обеспечения надёжности, безопасности и портируемости ПО были разработаны стандарты безопасного программирования, содержащие лучшие практики и ограничивающие использование небезопасных конструкций языка. Одним из наиболее известных стандартов является MISRA C, изначально разработанный для применения в автомобильной промышленности, но получивший распространение во многих отраслях, таких как аэрокосмическая промышленность, телекоммуникации, разработка медицинского оборудования, оборонная промышленность, железнодорожный транспорт и другие. Также распространены стандарты MISRA C++, AUTOSAR C++, SEI CERT C и SEI CERT C++, C++ Core Guidelines, High Integrity C++.

Стандарты безопасного программирования не рекомендуется применять, если исходный код проекта уже написан, так как изменения для обеспечения соответствия стандарту могут внести новые ошибки.

1.3 Статический анализ

Статический анализ кода — автоматический анализ программ без их выполнения. С его помощью выполняется поиск уязвимостей и ошибок, нарушений стиля и соглашений о использовании библиотек и конструкций языка программирования. В отличие от предупреждений компилятора, в инструментах статического анализа часто используются более сложные и затратные по времени алгоритмы для более точных диагностик.

У всех анализаторов присутствуют как ложноположительные, так и ложноотрицательные результаты, так как при использовании только статического анализа невозможно добиться максимальной точности и полноты, например, из-за неразрешимости проблемы остановки.

1.4 Динамический анализ

Динамический анализ программ, в отличие от статического, предполагает выполнение кода. Динамический анализ может выражаться в виде обнаружения ошибок работы с памятью (и других) во время выполнения программы с помощью инструментации кода, динамического символьного выполнения программы, динамического анализа потока данных, фаззинг-тестирования.

1.5 Концепция безопасного компилятора

В статье [2] 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.

В отличие от статического и динамического анализа, безопасный компилятор может не только обнаруживать неопределённое поведение программы, но и сделать его определённым. Кроме того, инструменты анализа в большинстве случаев не смогут обнаружить, что уязвимости появились из-за оптимизаций кода компилятором. Отключение всех оптимизаций приводит к сильному замедлению программ, поэтому необходим безопасный компилятор, не создающий уязвимостей своими оптимизациями, но при этом не наносящий большого ущерба производительности.

Детализация требований к безопасному компилятору приведена в стандарте [3]. Выделяется три класса требований, каждый из которых характеризуется строгостью механизмов предотвращения уязвимостей и скоростью работы генерируемых программ.

1.6 Соответствие возможностей Clang требованиям к безопасному компилятору

Безопасный компилятор 3 класса должен выполнять только безопасные преобразования исходного и машинного кода. Помимо этого, компилятор третьего класса безопасности должен включать механизмы повышенной защищённости. Также одной из задач третьего класса безопасности является выдача предупреждений в ходе сборки программы в некоторых случаях неопределённого поведения. В Clang присутствует большая часть требуемых опций.

Основными задачами безопасного компилятора 2 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. В Clang реализована лишь часть опций.

Одной из основных задач безопасного компилятора 1 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 UndefinedBehaviorSanitizer (UBSan). Он выполняет большую часть видов проверок, которые должен осуществлять безопасный компилятор.

Другой задачей безопасного компилятора является управление распределением автоматической и статической памяти. В рамках этой задачи компилятор 1 класса должен поддерживать возможность динамической компоновки программы, при которой при каждом запуске программы функции будут расположены в памяти в случайном порядке. Clang не поддерживает статическое (во время компиляции) случайное распределение памяти, но может поддерживать динамическое распределение при использовании компоновщика и динамического загрузчика, в которых реализована эта возможность.

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 при разработке безопасного компилятора на его основе.

2 Практическая часть

Безопасный компилятор Safelang на основе Clang 16.0.6 разрабатывается в рамках сотрудничества с ИСП РАН. Программа для ЭВМ была зарегистрирована в государственном реестре [6]. По результатам работы была опубликована статья в журнале «Труды ИСП РАН» [7].

2.1 Принудительное включение опций

Были добавлены опции `-Safe3`, `-Safe2`, `-Safe1`, включающие доступные в Clang и созданные в этой работе опции соответствующих классов безопасности. Для этого был разработан механизм принудительной установки опций. Как его часть, для управления функциональностью опций был разработан предметно-ориентированный язык на основе TableGen, позволяющий легко описать принудительное или опциональное включение опций.

2.2 Тестирование для разных архитектур

Безопасный компилятор должен поддерживать не только архитектуру x86-64, но и AArch64, а потенциально и другие. Поэтому была реализована возможность запускать тесты времени выполнения для всех архитектур на x86-64-системе.

2.3 Опции аварийного завершения работы

Механизмы безопасности при срабатывании могут выполнять дополнительный код, например, выводить информацию об ошибке перед аварийным завершением программы, что увеличивает поверхность атаки. С другой стороны, эта информация полезна для отладки программы. Поэтому в безопасном компиляторе реализованы опции для выбора одного из нескольких способов аварийного завершения.

2.4 Опции 3 класса

Для 3 класса были реализованы несколько опций.

2.4.1 `-fkeep-oversized-shifts` и `-fkeep-div-by-zero`

Опция `-fkeep-oversized-shifts` предотвращает оптимизацию побитовых сдвигов в случаях, когда второй аргумент оператора сдвига меньше нуля или больше или равен ширине типа, а `-fkeep-div-by-zero` — оптимизацию деления и взятия остатка в случаях, когда делитель может быть равен 0. Эти

опции реализованы заменой соответствующих инструкций LLVM IR вызовами новых intrinsic-функций. Проходы `InstCombine` и `SCCP` дополнены оптимизациями этих вызовов, осуществляемыми только в тех случаях, когда компилятор способен доказать, что аргументы корректны. В противном случае вызовы intrinsic-функций раскрываются после всех оптимизационных проходов, избегая, таким образом, оптимизаций.

2.4.2 `_FORTIFY_SOURCE`

Заголовочные файлы, содержащие функции стандартной библиотеки C с дополнительными проверками (например, на выход за границы массива), 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`, 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`.

2.4.3 `-fforce-volatile-before-setjmp`

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`.

2.5 Опции 2 класса

Для 2 класса были реализованы несколько опций.

2.5.1 -fpreserve-memory-writes

Для сохранения побочных эффектов записи в память была создана опция -fpreserve-memory-writes, предотвращающая DSE (Dead Store Elimination) в нескольких проходах оптимизационного конвейера. Благодаря этой опции 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.

2.5.2 -fassume-unaligned

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 выровненную память, генерируются инструкции для невыровненной памяти и предотвращаются аварийные завершения программ в случаях, когда используемый участок памяти имел некорректное выравнивание.

2.5.3 -finbounds-aliasing

Опция -finbounds-aliasing предотвращает оптимизации, при которых компилятор считает, что указатель, выходящий за границы объекта, на который он указывает, не может указывать на другие объекты. -finbounds-aliasing дополняет опцию -fno-strict-aliasing, включенную с 3 класса безопасности, в отключении оптимизаций, полагающихся на отсутствие алиасинга — ситуации существования разных указателей, указывающих на одну и ту же область памяти. Опция делает более консервативной обработку инструкций GetElementPtr, вычисляющих адрес элемента структуры данных, в BasicAliasAnalysis, также отключает оптимизацию GEP, всегда

выходящих за границы объекта, в `InstCombineLoadStoreAlloca`, и предотвращает оптимизации в `SelectionDAGAddressAnalysis` и `ScheduleDAGInstrs`.

2.6 Опции 1 класса

Для 1 класса были реализованы несколько опций.

2.6.1 `-fsanitize=float-to-float-cast-overflow`

Так как проверка `float-cast-overflow` в UBSan 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.

2.6.2 `-fsanitize=null-call`

Этот санитайзер проверяет, что при непрямом вызове функции не используется нулевой указатель. Хотя на большинстве платформ эта операция приводит к ошибке сегментации, поведение вызова функции через `NULL`-указатель не определено. Такая проверка отсутствует в `-fsanitize=null` из UBSan.

2.6.3 `-fsanitize=function`

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), доступной только для C++.

2.6.4 `-fsanitize=return-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++.

2.6.5 `-fsanitize=variadic`

В C и C++ существуют функции с переменным количеством аргументов — variadic-функции. Эти функции не типобезопасны — никак не проверяется, что вызывающая сторона передала аргумент того типа, который ожидала функция. Кроме того, функция может попытаться прочитать больше аргументов, чем ей было передано. Был разработан опциональный санитайзер, проверяющий количество и соответствие типов аргументов во время выполнения.

2.6.6 Случайное распределение статической памяти

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 `RandomizeLayout`, который задаёт случайный порядок функций и глобальных переменных на основе содержимого модуля и числа, задаваемого опцией `-mllvm -rng-seed`.

2.6.7 Случайное распределение автоматической памяти

Для рандомизации автоматической памяти создана опция `-floc-var-per`, перемешивающая локальные переменные в случайном порядке. Эта функциональность реализована в том же проходе, который используется для рандомизации статической памяти. 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.

2.7 Результаты

2.7.1 Корректность

Для безопасного Clang были разработаны регрессионные тесты и юнит-тест, использующие инструменты Lit и Google Test соответственно. Это позволяет выполнять тесты корректности безопасных функций вместе с остальными тестами Clang и LLVM при разработке. Также разработанный безопасный компилятор на основе Clang успешно проходит тесты из набора, созданного для проверки корректности безопасного компилятора SAFEC 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. Из-за того, что вместо предупреждения `-Wclobbered` реализована опция `-fforce-volatile-before-setjmp`, соответствующие тесты были отключены для Clang и были добавлены новые.

2.7.2 Исследование производительности

Производительность программ, скомпилированных безопасным компилятором, оценивалась с помощью тех же 5 тестов, что и для SAFEC. Во всех случаях, кроме одного (x264 с `-Safe1`), производительность соответствует стандарту (табл. 1).

Таблица 1 – Результаты измерения производительности

Тест	Без `-Safe`	`-Safe3`	`-Safe3` замедл.	`-Safe2`	`-Safe2` замедл.	`-Safe1`	`-Safe1` замедл.
GNU Go	3.93 с	4.06 с	3.31%	4.27 с	8.65%	6.83 с	73.79%
LAME	5.14 с	4.91 с	-4.47%	4.95 с	-3.76%	13.26 с	158.08%
fannkuch	2.23 с	2.10 с	-5.83%	2.13 с	-4.48%	2.68 с	20.18%
x264	1.44 с	1.55 с	7.37%	1.58 с	9.71%	6.18 с	329.00%
zlib	1.51 с	1.61 с	6.53%	1.64 с	8.19%	2.38 с	57.86%

2.7.3 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 дистрибутив Alpine Linux 3.21.2.

Таблица 2 – Результаты сборки пакетов Alpine Linux

Класс	Успешно собрано	Ошибки сборки	Ошибки тестирования
Без `-Safe`	4096	385	
`-Safe3`	4070	22	4
`-Safe2`	4019	49	2
`-Safe1`	3537	482	

98.1% пакетов, собираемых небезопасным Clang, удалось собрать со 2 классом безопасности, а 86.3% — с 1 классом (табл. 2). Можно сделать вывод, что почти все программы могут работать при компиляции с 3 или 2 уровнем безопасности без изменения исходного кода. По результатам сборки были предложены исправления ошибок, найденных следующими опциями, для 17 пакетов:

— `-D_FORTIFY_SOURCE=3`: 1 — принято, 1 — нет ответа;
— `-ftrivial-auto-var-init=zero`: 1 — нет ответа;
— `-fsanitize=variadic`: 12 — приняты, 2 — нет ответа.

ЗАКЛЮЧЕНИЕ

В данной работе были рассмотрены такие методы предотвращения уязвимостей в программах, как использование стандартов безопасного программирования, инструментов статического и динамического анализа, безопасного компилятора. Было обосновано решение о разработке безопасного компилятора языков C и C++ на основе Clang, для чего было проверено соответствие его возможностей требованиям стандарта. Выяснилось, что Clang не соответствует ни одному классу безопасности. Реализованы опции, включающие доступные в Clang опции для 3, 2 и 1 классов безопасности. Были описаны и реализованы все компоненты, недостающие для соответствия всем классам безопасности. Соответствие безопасного Clang стандарту было проверено с помощью тестов корректности и производительности. Полученный компилятор был зарегистрирован в государственном реестре [6]. По результатам работы была опубликована статья [7].

В ходе написания работы были решены следующие задачи:

- изучены методы предотвращения уязвимостей в программах;
- проверено соответствие возможностей Clang стандарту безопасного компилятора;
- реализованы опции, включающие существующие функции для частичного соответствия классам безопасности;
- доработан Clang для соответствия 3, 2 и 1 классам безопасности;
- протестирована корректность работы и производительность безопасного Clang.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Undefined behavior: what happened to my code? / X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, M. F. Kaashoek // Proceedings of the Asia-Pacific Workshop on Systems. — New York, NY, USA : Association for Computing Machinery, 2012. — P. 1–7. — (APSYS '12). — ISBN 9781450316699. — DOI: 10.1145/2349896.2349905.
2. Предотвращение уязвимостей, возникающих в результате оптимизации кода с неопределенным поведением / Р. В. Баев, Л. В. Скворцов, Е. А. Кудряшов, Р. А. Бучацкий, Р. А. Жуйков // Труды Института системного программирования РАН. — 2021. — Т. 33, № 4. — С. 195—210. — ISSN 2220-6426. — DOI: 10.15514/ISPRAS-2021-33(4)-14.
3. ГОСТ Р 71206-2024 Защита информации. Разработка безопасного программного обеспечения. Безопасный компилятор C/C++. Общие требования. — Москва : Российский институт стандартизации, 2024.
4. ГОСТ Р 56939-2016 Защита информации. Разработка безопасного программного обеспечения. Общие требования. — Москва : Стандартинформ, 2016.
5. *Сикорд Р. С.* Безопасное программирование на C и C++, 2-е изд. : Пер. с англ. — М. : ООО «И.Д. Вильямс», 2015. — 496 с. — ISBN 978-5-8459-1908-3.
6. Свидетельство о государственной регистрации программы для ЭВМ №2024686547 / А. А. Синкевич, П. Д. Дунаев, А. М. Гранат, В. А. Иванишин, А. В. Монаков, Н. Ю. Шугалей. — Заявка №2024685551 от 28.10.2024, опубликовано 11.11.2024 в бюл. № 11.
7. Разработка безопасного компилятора на основе Clang / П. Д. Дунаев, А. А. Синкевич, А. М. Гранат, И. А. Батраева, С. В. Миронов, Н. Ю. Шугалей // Труды Института системного программирования РАН. — 2024. — Т. 36, № 4. — С. 27—40. — ISSN 2220-6426. — DOI: 10.15514/ISPRAS-2024-36(4)-3.