УНИВЕРСИТЕТ ИТМО
НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
март–апрель 2020 Том 20 № 2 ISSN 2226-1494 http://ntv.itmo.ru/
SCIENTIFIC AND TECHNICAL JOURNAL OF INFORMATION TECHNOLOGIES, MECHANICS AND OPTICS
March–April 2020 Vol. 20 No 2 ISSN 2226-1494 http://ntv.itmo.ru/en/
НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК УНИВЕРСИТЕТ ИТМО
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

УДК 004.052.42 doi: 10.17586/2226-1494-2020-20-2-243-248

ТЕСТИРОВАНИЕ МНОГОПОТОЧНЫХ ПРИЛОЖЕНИЙ С БЛОКИРОВКАМИ НА НЕАТОМАРНЫХ ПЕРЕМЕННЫХ

О.В. Доронин[a], К.И. Дергун[a], А.М. Дергачев[a], А.Г. Ильина[a], С.П. Горлач[b]

[a] Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация
[b] Университет Мюнстера, Мюнстер, 48149, Германия
Адрес для переписки: dam600@gmail.com

Информация о статье
Поступила в редакцию 14.01.20, принята к печати 27.02.20
Язык статьи — русский

Ссылка для цитирования: Доронин О.В., Дергун К.И., Дергачев А.М., Ильина А.Г., Горлач С.П. Тестирование многопоточных приложений с блокировками на неатомарных переменных // Научно-технический вестник информационных технологий, механики и оптики. 2020. Т. 20. № 2. С. 243–248. doi: 10.17586/2226-1494-2020-20-2-243-248

Аннотация

Предмет исследования. Исследован существующий алгоритм фаззинг-тестирования для поиска ошибок типа гонки данных в многопоточных программных приложениях, реализованный в инструменте Google TSan. Недостатком исследованного алгоритма является отсутствие возможности тестирования программных приложений, использующих неатомарные переменные. Это исключает возможность применения подобного инструмента для тестирования современных приложений, реализующих совместный доступ программных потоков к данным. Предложен новый алгоритм фаззинг-тестирования многопоточных приложений и способ внедрения его в модуль фаззинг-тестирования инструмента Google TSan. **Метод.** При фаззинг-тестировании многопоточных приложений входными данными являются различные комбинации исполнения программных потоков. Предложенный метод фаззинг-тестирования многопоточных программных приложений строится на предположении, что ошибки в многопоточных приложениях проявляются только в точках переключения потоков — точках синхронизации. Планировщик потоков реализован максимально просто. Каждому потоку назначается маркер состояния, позволяющий отслеживать его активность в процессе работы программы. Поток может находиться в неизвестном состоянии (до первой точки синхронизации), в состоянии исполнения, в ожидании — в очереди на исполнение, а также в состоянии, когда поток исчерпал свой квант исполнения, но не достиг точки синхронизации. Подобное состояние потока при достижении точки синхронизации автоматически меняется на состояние ожидания. Управление потоками осуществляется с помощью отдельного программного потока, следящего за состояниями всех потоков и выставляющего потокам, исчерпавшим квант исполнения, соответствующий маркер. Механизм поиска ошибок внедряется в программный продукт на этапе компиляции при указании соответствующих опций. **Основные результаты.** В инструмент Google TSan внедрен новый модуль фаззинг-тестирования, который позволяет находить ошибки типа гонки данных в любых многопоточных приложениях как с синхронизацией доступа к разделяемым данным, так и при совместном доступе программных потоков к данным. **Практическая значимость.** Верификация многопоточных приложений с совместным доступом к данным, в частности использующих неатомарные переменные, особенно актуальна для высоконагруженных масштабируемых программных систем.

Ключевые слова
многопоточность, гонки данных, взаимоблокировки, инструменты поиска ошибок, фаззинг-тестирование

doi: 10.17586/2226-1494-2020-20-2-243-248

TESTING OF MULTITHREADED APPLICATIONS WITH LOCKS ON NON-ATOMIC VARIABLES

O.V. Doronin[a], K.I. Dergun[a], A.M. Dergachev[a], A.G. Ilina[a], S.P. Gorlatch[b]

[a] ITMO University, Saint Petersburg, 197101, Russian Federation
[b] University of Muenster, Muenster, 48149, Germany
Corresponding author: dam600@gmail.com

Article info
Received 14.01.20, accepted 27.02.20
Article in Russian

For citation: Doronin O.V., Dergun K.I., Dergachev A.M., Ilina A.G., Gorlatch S.P. Testing of multithreaded applications with locks on non-atomic variables. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2020, vol. 20, no. 2, pp. 243–248 (in Russian). doi: 10.17586/2226-1494-2020-20-2-243-248

Abstract

Subject of Research. The paper presents the study of fuzz testing algorithm for "data race" type fault finding in multithreaded software. The algorithm is implemented in the Google TSan tool. The disadvantage of the studied algorithm is the inability to test software products which use non-atomic variables. At this, the use of such tool is excluded for testing of modern applications that implement joint access of program streams to data. We propose a new algorithm for fuzz testing of multithreaded applications and a method for its implementation in the fuzz testing module of the Google TSan tools. **Method.** Various combinations of the execution of program streams are the input data in the process of fuzz testing of multithreaded applications. The proposed method of fuzz testing for multithreaded software applications assumes that errors in multithreaded applications manifest themselves only at the threads switching points, also called synchronization points. The thread scheduler is implemented as simply as possible. Each thread is assigned with a status marker tracking its activity during the program running. A thread can be in an unknown state (until the first synchronization point), in a state of execution, in a state of waiting in a queue for execution, as well as in a state where the thread has spent its execution quantum but has not reached the synchronization point. Such thread state is automatically changed to the state of waiting at the synchronization point. Thread control is carried out using a separate program thread that monitors the status of all threads and exposes the corresponding marker to the threads that have spent the execution quantum. The error search mechanism is implemented in the software product at the compilation stage by specifying the appropriate options. **Main Results.** A new fuzz testing module has been integrated in the Google TSan tool, which finds "data race" errors in any multithreaded applications, both with synchronization of access to shared data and with shared access to data. **Practical Relevance.** Verification of multithreaded software with shared access to data, in particular case of applying non-atomic variables, is especially relevant for the heavily loaded scalable software systems.

Keywords

multithreading, data race, deadlock, error finding tools, fuzz testing

Введение

Для высоконагруженных программных систем важным показателем является масштабируемость, что достигается, в том числе за счет реализации принципа многопоточности и распределенности программных приложений. Однако при возрастании числа потоков синхронизированный доступ к данным в условиях большой нагрузки может приводить к ухудшению производительности. Если примитивами синхронизации являются объекты ядра операционной системы, то использование многопоточности приводит к росту накладных расходов. Для повышения производительности программных систем разработчики уходят от использования атомарных переменных — примитивов синхронизации, обеспечивающих потокобезопасность (thread-safety) на уровне ограничения доступа к данным, к параллельным структурам данных, обеспечивающим потокам совместный доступ к данным[1]. С одной стороны, это дает преимущество в производительности, с другой – искать ошибки и тестировать такие приложения становится на порядок сложнее [1].

При разработке программ, использующих многопоточность, могут встретиться такие типы ошибок, как инверсии приоритетов[2], взаимоблокировки (deadlock), зависания (hang or freeze) или зацикливания (livelock), проблема ABA [2] и другие [3, 4]. В программах, где потоки используют общую память, могут возникать гонки данных (data race) [5]. Состояния гонки данных возникают, когда разные процессы получают доступ к общим данным без явной синхронизации. Обнаружение данного типа ошибок представляет наибольшие трудности из-за непредсказуемости и неповторяемости. Общее тестовое покрытие отрабатывает их без сбоев, тогда как проявлять себя они могут единичными случаями неправильного поведения программного обеспечения. Для поиска таких ошибок используются специальные инструменты, такие как Valgrind[3], Google TSan[4] [6] и другие [7–10]. Google TSan — наиболее успешный инструмент, разработанный в 2009 году на основе Valgrind специально для поиска ошибок типа гонки данных, использует гибридный (hybrid mode) и чистый «выполняется прежде» (pure happens-before mode) режимы, а также выполняет подробное аннотирование сбоев, что в целом позволяет ему определять наибольшее число ошибок типа гонки данных среди прочих инструментов. Одним из последних усовершенствований инструмента TSan является внедрение в него модуля фаззинг-тестирования (fuzzing или fuzz testing — техника тестирования, заключающаяся в передаче приложению на вход неправильных, неожиданных или случайных данных) [11], алгоритм работы которого позволяет получать широкое тестовое покрытие за счет перебора различных комбинаций исполнения потоков [12]. Однако предложенный алгоритм не способен обрабатывать блокировки и ожидания на неатомарных переменных, что делает его неэффективным в ряде ситуаций с нарушением потокобезопасности при совместном использовании данных. Следующий пример демонстрирует такую ситуацию — алгоритм тестирования блокируется на неатомарной переменной barrier, так как не может найти следующую точку син-

[1] Lock-free структуры данных. 1 – Начало [Электронный ресурс]. Режим доступа: https://habr.com/ru/post/195770/, свободный. Яз. англ. (дата обращения 18.01.2020).

[2] Priority Inversion [Электронный ресурс]. Режим доступа: https://www.sciencedirect.com/topics/computer-science/priority-inversion, свободный. Яз. англ. (дата обращения 17.01.2020).

[3] Valgrind [Электронный ресурс]. Режим доступа: https://valgrind.org/, свободный. Яз. англ. (дата обращения 17.01.2020).

[4] TSan — сокращение от англ. Thread Sanitizer

хронизации. В результате получаем взаимоблокировку на уровне инструмента тестирования.

```
volatile std::byte barrier = 0;
void thread1() {
   while (!barrier);
}
void thread2() {
  barrier = true;
  //...
}
```

При тестировании программных приложений с использованием аналогичных неатомарных переменных ранние реализации модуля фаззинг-тестирования инструмента TSan перестают реагировать на действия пользователя и зависают. Таким образом, существующий подход к верификации новейших программных продуктов требует доработки алгоритмов модуля фаззинг-тестирования инструмента TSan.

В настоящей работе представлен новый метод и алгоритм, позволяющий проводить фаззинг-тестирование потоков с корректной обработкой блокировок и ожиданий на неатомарных переменных. Работа является продолжением серии результатов, полученных авторами ранее:

— для проведения фаззинг-тестирования многопоточных приложений была разработана архитектура [12], реализация которой охватывает различные стратегии планирования исполнения потоков для поиска ошибок в многопоточном коде;
— добавлена поддержка работы с lock-free[1] алгоритмами и атомарными переменными [12];
— устранен один из недостатков — отсутствие поддержки мьютексов (mutex, от mutual exclusion) [11].

В настоящее время модуль фаззинг-тестирования многопоточных приложений, реализующий предложенный в работе алгоритм, находится на стадии внедрения в основную ветку инструмента TSan [6, 13].

Алгоритм тестирования на основе неатомарных переменных

Фаззинг[2] [14, 15] позволяет получать широкое тестовое покрытие за счет перебора различных вариантов исполнения программы. В самом простом случае — это перебор всех вариантов, что является неэффективным, так как требует больших вычислительных мощностей, поэтому при фаззинг-тестировании используются сложные алгоритмы, основанные на сборе и анализе статистики. Одним из вариантов является сбор и анализ тестового покрытия кода, во время которого строятся предположения, связанные с вероятностью встречаемости ошибок при конкретных входных параметрах.

При фаззинг-тестировании многопоточных приложений вместо перебора входных аргументов перебираются комбинации исполнения потоков. Основное предположение, на котором основывается разработанный алгоритм, заключается в том, что ошибки в многопоточных приложениях проявляются только в точках синхронизации. В связи с этим точки для переключения потоков выбираются среди операций: чтение/запись в атомарную переменную, захват/освобождение мьютекса, ожидание/нотификация на условной переменной и другие.

Основываясь на предположении, что ошибки в многопоточных приложениях проявляются только в точках синхронизации, было решено реализовать простейший интерфейс планировщика потоков с единственным методом SynchronizationPoint:

```
class IScheduler {
   virtual void SynchronizationPoint() = 0;
};
```

Для улучшения модуля фаззинг-тестирования потоков, необходимо решить две задачи:

1) реализовать алгоритмы планирования потоков для представленного интерфейса IScheduler (алгоритмы, которые задают последовательности исполнения потоков; простой алгоритм — случайный выбор потоков в точках синхронизации);
2) внедрить реализацию интерфейса IScheduler в архитектуру TSan.

Начнем с описания решения второй задачи. На этапе компиляции TSan встраивается в исходный код тестируемой программы. В результате в исходном коде функции для работы с мьютексами и условными переменными подменяются на специальные TSan-функции, операции чтения и записи в переменные инструментируются, а другие функции перехватываются инструментом TSan. Такой подход позволяет незаметно для пользователя подменять некоторую функциональность и на ее основе создавать алгоритмы поиска ошибок в коде. С точки зрения пользователя требуется взять исходный код программы и скомпилировать с особыми опциями компилятора, после чего алгоритмы поиска ошибок автоматически активируются. У данного подхода есть существенный недостаток — компиляция потребует дополнительного времени, либо при раздельной компиляции дополнительное время потребуется на сборку программного кода. Однако практика показывает, что в хорошо структурированных программах этот недостаток незначителен.

Теперь рассмотрим, как описанная архитектура выглядит с точки зрения программного кода. Допустим, произошел захват и затем освобождение мьютекса, как показано в следующем фрагменте кода:

```
pthread_mutex_t mutex;
//...
pthread_mutex_lock(&mutex);
// логика приложения
pthread_mutex_unlock(&mutex);
```

[1] Lock-free структуры данных. 1 — Начало [Электронный ресурс]. Режим доступа: https://habr.com/ru/post/195770/, свободный. Яз. англ. (дата обращения 18.01.2020).

[2] Фаззинг — важный этап безопасной разработки [Электронный ресурс]. Режим доступа: https://habr.com/ru/company/dsec/blog/450734/, свободный. Яз. англ. (дата обращения 18.01.2020).

После внедрения TSan этот код преобразуется следующим образом:

```
pthread_mutex_t mutex;
// ...
tsan_mutex_lock(&mutex);
// ...
tsan_mutex_unlock(&mutex);
```

Реализация функций tsan_mutex_lock/unlock лежит на плечах разработчиков алгоритмов поиска ошибок. Что касается метода SynchronizationPoint для tsan_mutex_lock, то его использование продемонстрировано в нижеследующем примере. Здесь создаются две точки синхронизации — до и после захвата мьютекса. Таким образом, происходит внедрение планировщика IScheduler в TSan.

```
int tsan_mutex_lock(void* mutex) {
  IScheduler::SyncronizationPoint();
  //…
  pthread_mutex_lock(mutex);
  //…
  IScheduler::SynchronizationPoint();
}
```

Теперь перейдем к описанию того, как построен алгоритм фаззинг-тестирования многопоточных приложений, учитывающий недостатки существующего модуля. Каждый поток может находиться в одном из следующих отслеживаемых планировщиком IScheduler состояний:

— UNKNOWN — поток находится в этом состоянии до тех пор, пока не встретил первую точку синхронизации;
— RUNNING — в текущий момент времени основным потоком исполнения является поток, который имеет данное состояние. Только один поток в программе может находиться в таком состоянии;
— WAIT — поток, имеющий данное состояние, ожидает своей очереди на исполнение;
— OUT_TIME — такое состояние бывает у потока, если он исчерпал свой квант исполнения и не достиг следующей точки синхронизации SynchronizationPoint.

Как показано в примере, приведенном в разделе «Введение» и демонстрирующем приостановку потока thread1, одной из причин состояния OUT_TIME является зависание потока на ожидании некоторого события, в данном примере — на ожидании события barrier. В этом случае поток остается на исполнении, и таких исполняющихся OUT_TIME потоков может быть несколько. Когда такие потоки доходят до точки синхронизации SynchronizationPoint, то они переходят в состояние WAIT.

Проблема зависания потока на неатомарной переменной решается с помощью состояния OUT_TIME, когда реально физически могут исполняться несколько потоков. Такой алгоритм накладывает ограничения на структуры данных, которые используются в планировщике IScheduler — они должны быть потокобезопасными, так как доступ к IScheduler может осуществляться из нескольких потоков. При этом получены следующие преимущества: отсутствуют зависания при различных стратегиях планирования потоков; поддерживается корректная работа для любых готовых к промышленной эксплуатации (production-ready) приложений.

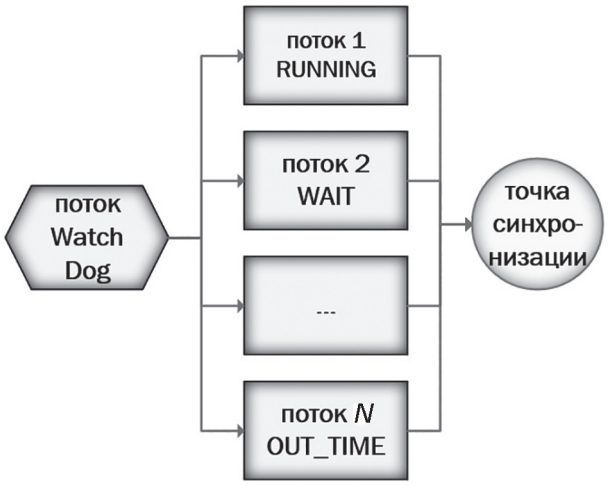

Рисунок. Состояния потоков многопоточного приложения

Поскольку ошибки в многопоточных приложениях возникают при заранее неизвестных комбинациях выполнения потоков, необходимо перебирать порядок исполнения потоков до появления ошибки, отслеживая при этом состояния потоков. Для отслеживания состояний, в которых могут находиться потоки приложения, используется специальный выделенных поток, написанный с использованием шаблона программирования Watch Dog. На рисунке схематично показано, как выглядит многопоточное приложение с интегрированным в нее потоком, отслеживающим состояния потоков приложения. Пусть в системе есть N потоков, количество которых может динамически увеличиваться или уменьшаться. Выбор потоков на исполнение происходит через точку синхронизации SynchronizationPoint за исключением обработки состояния OUT_TIME. Поток переводится в состояние OUT_TIME в том случае, если поток исполнялся дольше заданного промежутка времени. В зависимости от выбора длительности промежутка времени можно балансировать качество и время работы.

Ниже представлен псевдокод метода SynchronizationPoint, с помощью которого осуществляется планирование выполнения потоков в точке синхронизации.

```
SynchronizationPoint():
  tid = GetTid();
  oldState = state[tid];
  state[tid] = WAIT;
  if (oldState = RUNNING) {
    nextTid = GetNextTid();
    state[nextTid] = nextTid;
  }
  while (state[tid] == WAIT) Yield();
```

В приведенном выше псевдокоде маркер состояния state[tid] для текущего потока переводится в состояние

WAIT. Если этот поток не был в состоянии OUT_TIME, то выбирается следующий поток на исполнение. Таким образом, в каждый момент времени может быть только один поток в состоянии RUNNING, отслеживаемый управляющим потоком Watch Dog.

Заключение

В работе предложен новый метод тестирования многопоточных приложений, обеспечивающий корректную обработку зависаний программных потоков на неатомарных переменных. Полученные результаты позволяют проводить тестирование любых многопоточных приложений, использующих для синхронизации потоков не только примитивы, являющиеся объектами ядра операционной системы, но и реализующие совместный доступ к данным на основе неатомарных переменных. На основе предложенного метода разработан алгоритм и реализован новый программный модуль для фаззинг-тестирования, в котором отсутствуют недостатки предыдущих реализаций. В настоящее время ведется работа по интеграции результатов в основную ветку инструмента Google TSan[1].

Литература

1. Гедич А.А., Зыков А.Г., Лаздин А.В., Поляков В.И. Поиск процедур по графу переходов функциональной программы при верификации вычислительных процессов // Известия высших учебных заведений. Приборостроение. 2014. Т. 57. № 4. С. 64–68.
2. Dechev D., Pirkelbauer P., Stroustrup B. Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs // Proc. 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC 2010). 2010. V. 1. P. 185–192. doi: 10.1109/ISORC.2010.10
3. Multi-Threaded Programming Terminology – 2018 [Электронный ресурс]. URL: https://www.bogotobogo.com/cplusplus/multithreaded.php, свободный. Яз. англ. (дата обращения: 18.01.2020).
4. Трудности многопоточного программирования [Электронный ресурс]. URL: https://poznayka.org/s9887t1.html, свободный. Яз. рус. (дата обращения: 18.01.2020).
5. Race Condition vs. Data Race 2018 [Электронный ресурс]. URL: https://blog.regehr.org/archives/490, свободный. Яз. англ. (дата обращения: 18.01.2020).
6. Serebryany K., Iskhodzhanov T. ThreadSanitizer - Data race detection in practice // ACM International Conference Proceeding Series. 2009. P. 62–71. doi: 10.1145/1791194.1791203
7. Лукин М.А. Верификация параллельных автоматных программ // Научно-технический вестник информационных технологий, механики и оптики. 2014. Т. 14. № 1(89). С. 60–66.
8. Никифоров В.В. Протокол предотвращения взаимного блокирования задач в системах реального времени // Известия высших учебных заведений. Приборостроение. 2014. Т. 57. № 12. С. 21–27.
9. Никифоров В.В., Тюгашев А.А. Доступ к разделяемым ресурсам в системах реального времени с переменными приоритетами задач // Известия высших учебных заведений. Приборостроение. 2016. Т. 59. № 11. С. 964–970. doi: 10.17586/0021-3454-2016-59-11-964-970
10. Triantafillou P. An approach to deadlock detection in multidatabases // Information Systems. 1997. V. 22. N 1. P. 39–55. doi: 10.1016/S0306-4379(97)00003-3
11. Дергун К.И., Доронин О.В. Фаззинг тестирование fine-grained алгоритмов // Сборник тезисов докладов Конгресса молодых ученых [Электронный ресурс]. URL: https://kmu.itmo.ru/digests/article/1224, свободный. Яз. рус. (дата обращения: 18.01.2020).
12. Doronin O., Dergun K., Dergachev A. Automatic fuzzy-scheduling of threads in Google Thread Sanitizer to detect errors in multithreaded code // CEUR Workshop Proceedings. 2019. V. 2344. P. 1–12.
13. ThreadSanitizer project: documentation, source code, dynamic annotations, unit tests [Электронный ресурс]. URL: http://code.google.com/p/data-race-test, свободный. Яз. англ. (дата обращения: 18.01.2020).
14. Саттон М., Грин А., Амини П. Fuzzing: исследование уязвимостей методом грубой силы. Москва: Символ-Плюс, 2009. 555 с.
15. Vallen A., Johansson V. Random testing with sanitizers to detect concurrency bugs in embedded avionics software [Электронный ресурс]. URL: http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1269941&dswid=8217, свободный. Яз. англ. (дата обращения: 18.01.2020).

References

1. Gedich A.A., Zykov A.G., Lazdin A.V., Polyakov V.I. Search for procedure with the use of functional program transition graph to verify computational processes. *Journal of Instrument Engineering*, 2014, vol. 57, no. 4, pp. 64–68. (in Russian)
2. Dechev D., Pirkelbauer P., Stroustrup B. Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. *Proc. 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC 2010)*, 2010, vol. 1, pp. 185–192. doi: 10.1109/ISORC.2010.10
3. *Multi-Threaded Programming Terminology — 2018*. Available at: https://www.bogotobogo.com/cplusplus/multithreaded.php (accessed: 18.01.2020).
4. *Difficulties in concurrent programming*. Available at: https://poznayka.org/s9887t1.html (accessed: 18.01.2020). (in Russian)
5. *Race Condition vs. Data Race 2018*. Available at: https://blog.regehr.org/archives/490 (accessed: 18.01.2020).
6. Serebryany K., Iskhodzhanov T. ThreadSanitizer — Data race detection in practice. *ACM International Conference Proceeding Series*, 2009, pp. 62–71. doi: 10.1145/1791194.1791203
7. Lukin M. Verification of parallel automata-based programs. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2014, vol. 14, no. 1(89), pp. 60–66. (in Russian)
8. Nikiforov V.V. Protocol for prevention of task deadlocks in real-time systems. *Journal of Instrument Engineering*, 2014, vol. 57, no 12, pp. 21–27. (in Russian)
9. Nikiforov V.V., Tyugashev A.A. Access to shared resources in real-time systems with varying tasks priorities. *Journal of Instrument Engineering*, 2016, vol. 59, no. 11, pp. 964–970. (in Russian). doi: 10.17586/0021-3454-2016-59-11-964-970
10. Triantafillou P. An approach to deadlock detection in multidatabases. *Information Systems*, 1997, vol. 22, no. 1, pp. 39–55. doi: 10.1016/S0306-4379(97)00003-3
11. Dergun K.I., Doronin O.V. Fuzzing testing of fine-grained algorithms. *Proc. Conference of Young Scientists*. Available at: https://kmu.itmo.ru/digests/article/1224 (accessed: 18.01.2020). (in Russian)
12. Doronin O., Dergun K., Dergachev A. Automatic fuzzy-scheduling of threads in Google Thread Sanitizer to detect errors in multithreaded code. *CEUR Workshop Proceedings*, 2019, vol. 2344, pp. 1–12.
13. *ThreadSanitizer project: documentation, source code, dynamic annotations, unit tests*. Available at: http://code.google.com/p/data-race-test (accessed: 18.01.2020).
14. Sutton M., Greene A., Amini P. *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education, 2007, 576 p.
15. Vallen A., Johansson V. *Random testing with sanitizers to detect concurrency bugs in embedded avionics software*. Available at: http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1269941&dswid=8217 (accessed: 18.01.2020).

[1] fuzzing scheduler with use watch dog [Электронный ресурс]. Режим доступа: https://reviews.llvm.org/D66235 свободный. Яз. англ. (дата обращения 18.01.2020).

Авторы

Доронин Олег Владимирович — аспирант, Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация, Scopus ID: 57208322052, ORCID ID: 0000-0003-4209-8440, dorooleg@yandex.ru
Дергун Карина Ильдаровна —аспирант, Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация, Scopus ID: 57208315877, ORCID ID: 0000-0001-7350-6780, dergun_karina@mail.ru
Дергачев Андрей Михайлович — кандидат технических наук, доцент, Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация, Scopus ID: 57205166082, ORCID ID: 0000-0002-1754-7120, dam600@gmail.com
Ильина Аглая Геннадьевна — кандидат технических наук, доцент, Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация, ORCID ID: 0000-0003-1866-7914, agilina@itmo.ru
Горлач Сергей Петрович — PhD, профессор, Университет Мюнстера, Мюнстер, 48149, Германия, Scopus ID: 8971522600, ORCID ID: 0000-0001-6511-7574, gorlatch@uni-muenster.de

Authors

Oleg V. Doronin — Postgraduate, ITMO University, Saint Petersburg, 197101, Russian Federation, Scopus ID: 57208322052, ORCID ID: 0000-0003-4209-8440, dorooleg@yandex.ru
Karina I. Dergun — Postgraduate, ITMO University, Saint Petersburg, 197101, Russian Federation, Scopus ID: 57208315877, ORCID ID: 0000-0001-7350-6780, dergun_karina@mail.ru
Andrey M. Dergachev — PhD, Associate Professor, ITMO University, Saint Petersburg, 197101, Russian Federation, Scopus ID: 57205166082, ORCID ID: 0000-0002-1754-7120, dam600@gmail.com

Aglaia G. Ilina — PhD, Associate Professor, ITMO University, Saint Petersburg, 197101, Russian Federation, ORCID ID: 0000-0003-1866-7914, agilina@itmo.ru
Sergei P. Gorlatch — PhD, Professor, University of Muenster, Muenster, 48149, Germany, Scopus ID: 8971522600, ORCID ID: 0000-0001-6511-7574, gorlatch@uni-muenster.de