Формальные модели программ

Вводное занятие

Обзор курса. Предпосылки к задаче формальной верификации, актуальность. Современные методы верификации программ (рецензирование, тестирование, статический анализ, формальная верификация) и их интеграция в процессы разработки

Я — **Максим Александрович Кривчиков**, н.с. НИИ механики МГУ

maxim.krivchikov@gmail.com

Материалы курса: https://maxxk.github.io/formal-models-2015/

Обзор курса

В этом семестре — более теоретические вопросы:

- каким образом в принципе можно что-то доказывать про программы
- какие математические модели используются для описания вычислений
- теория для одного из подходов к верификации — программированию с зависимыми типами
  этот подход тесно связан с активно развивающимся направлением современной математики — теорией типов

В следующем семестре(предварительно) — практические вопросы:

- как описывать семантику программ и языков программирования (монады)
- как описывать отдельные аспекты исполнения программ
- как выполнять доказательства свойств программ
- что делать, чтобы такие доказательства было реально получить на практике (предметно-ориентированные языки и вывод типов)
- разбор существующих результатов (компилятор C CompCert, ядро ОС SeL4)

Похожие курсы в других университетах мира

Первый семестр:

- CMU (US) 15-819 Homotopy Type Theory — достаточно похоже на программу этого семестра
- Университет Стокгольма (Швеция) MM8036 ht15 Type Theory
- Университет Неймегена (Нидерланды) IMC010: Type Theory and Coq
- Университет Ноттингема (Великобритания) G52IFR Introduction to Formal Reasoning

Второй семестр:

- Cornell University (US) CS4110–CS6110
- University of Edinburgh INFR11114 Types and Semantics for Programming Languages
- University of Texas in Dallas CS 6371: Advanced Programming Languages

Общая информация

Основные материалы для подготовки — по презентациям; по основным темам будут ссылки на литературу для расширенного изучения (в том числе — обновляемый список литературы на сайте)

Курс читается впервые, поэтому принимаются замечания и предложения по содержанию в целом и по отдельным аспектам изложения.

Любые возникающие вопросы лучше задавать прямо по ходу занятия.

Зачёт/экзамен

Курс — обязательная кафедральная дисциплина; насколько я понимаю, все остальные могут сдавать его как спецкурс.

В этом семестре — зачёт (для 510 группы) или экзамен (для остальных).

В следующем семестре курс продолжится под названием «Формальные модели и верификация свойств программ», по этому курсу будет экзамен для всех.

В рамках курса будут выполняться задания для самостоятельного выполнения «со звёздочками». Одна «*» — плюс пол-балла на экзамене. Принимаются предложения по заданиям, не включённым в список, но позволяющим лучше иллюстрировать темы курса.

Введение

Программы окружают нас

Например, в следующих сферах жизни общества:

- развлечения
- рабочие места
- коммуникации
- транспорт
- медицина
- наука
- производство
- энергетика
- вооружение

Программы пишут на разных языках программирования

С JavaScript:

- 5.9% (74) без ЯП,
- 40.5% (509) используют один ЯП,
- 26.8% (337) используют два ЯП,
- 26.8% (336) используют три и более ЯП.

Без JavaScript и репозиториев без языков

- 51.6% (239) используют один ЯП,
- 25.5% (118) используют два ЯП,
- 22.8% (106) используют три и более ЯП.

Данные получены по 1256 репозиториям GitHub, наиболее популярных по количеству голосов пользователей и по количеству производных репозиториев в 2013 году.

Программы модифицируются и усложняются

GNU gzip (утилита для сжатия данных)

версии 1.2.4 – 1.6 (1993–2013)

× **6** файлов, × **7** строк

(34 ⟶ 216 файлов, 5.8 ⟶ 42 тыс. строк кода)

Ядро ОС FreeBSD для архитектуры i386

версии 2.0.5 – 8.4 (1995-2013)

× **8** файлов, × **12** строк

(1066 ⟶ 8860 файлов, 280 ⟶ 3380 тыс. строк кода)

Данные получены утилитой CLOC 1.60 для следующих типов файлов: C, C++, C/C++ Header

Программы содержат ошибки

Coverity Scan Open Source Report 2013:

Статический анализ кода 741 Open Source проекта от ≈ 10 тыс. строк до ≈ 8 млн строк, в среднем ≈ 340 тыс. строк на проект

Плотность обнаруженных ошибок: **0.59** на 1 тыс. строк кода. Для небольших проектов масштаба gzip (42 тыс. строк) — в среднем 24 ошибки *обнаружено*

В.В. Липаев отмечает [1]:

- высококачественное ПО имеет плотность порядка **0.20** ошибок на 1 тыс. строк кода
- существующими средствами достижима минимальная плотность порядка **0.05** ошибок на 1 тыс. строк кода

[1] Липаев В.В. Программная инженерия. Методологические основы: Учебник

Ошибки в программах могут приводить к катастрофическим последствиям

Уязвимости в ПО

- Stuxnet, Иранская ядерная программа (2009) — предположительно, повреждены центрифуги, используемые при обогащении урана

Некорректная работа ПО

- «Фобос-Грунт» (2011) — утрачена автоматическая межпланетная станция, ущерб ≈5 млрд руб.
- Панама, установки лучевой терапии (2000-2001) — пострадало 28 человек, погибло не менее 5
- Therac-25, установки лучевой терапии (1985-1987) — 6 пострадавших, 3 погибших
- FADEC (Chinook HC.2), отказ ПО управления двигателем вертолёта — одна из вероятных причин крушения в 1994 г. — 29 погибших

Или к менее катастрофическим, но всё равно серьёзным

Уязвимости в ПО

- OpenSSL Heartbleed (CVE-2014-0160) — значительная часть TLS-серверов в Интернет (популярные дистрибутивы Linux — Debian-based, RedHat-based; OpenBSD, FreeBSD, NetBSD, роутеры Cisco, Juniper, …) имели уязвимость в течение более 1 года
- (подставьте уязвимость, о которой вы читали за последние полгода, за исключением различных тайминг-атак)

Недокументированные возможности

- Volkswagen (2015) — занижались данные по выбросам вредных веществ дизельными двигателями, компанию ожидает штраф до 18 млрд. долл.

ADAM MANN SCIENCE 11.07.11 3:52 PM

RUSSIA RETURNING TO MARS AFTER 15-YEAR BREAK

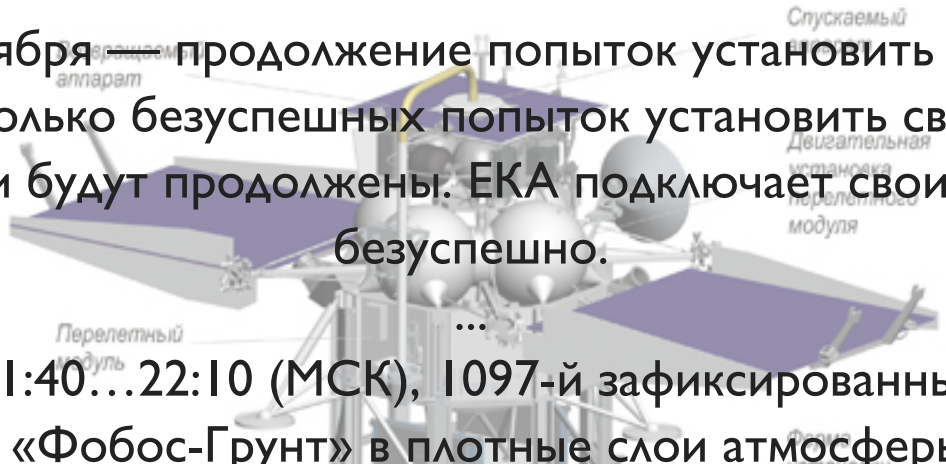
Фобос-Грунт

Фобос-Грунт (Wikipedia)

2011 г. 9 ноября — запуск и ожидаемое выведение на траекторию полёта к Марсу:
00:16:02.871 (МСК) — старт
00:27:31 (T+688 сек) — раскрытие солнечных батарей, приём телеметрии, ориентация, заряд батарей
02:55:47.981 — первое включение маршевой двигательной установки АМС не состоялось; блок баков сброшен не был; после длительных поисков с привлечением российских и американских систем контроля космического пространства станция была обнаружена; вероятно, произошёл сбой при переходе от ориентации по Солнцу к ориентации по звёздам; возможно, что не прошла команда датчиков на включение двигательной установки; по уточнённым параметрам орбиты и запасу энергии, имеется 2-недельный запас времени, чтобы заново передать в АМС программу полёта;

...

2011 г. 10 ноября — продолжение попыток установить связь с АМС:
Осуществлено несколько безуспешных попыток установить связь с межпланетной станцией; попытки будут продолжены. ЕКА подключает свои станции, но тоже безуспешно.



...

2012 г. 15 января, 21:40…22:10 (МСК), 1097-й зафиксированный виток орбиты — снижение АМС «Фобос-Грунт» в плотные слои атмосферы, прекращение

существования (аэродинамический перегрев, механическое разрушение и сгорание), возможное падение несгоревших фрагментов в южной части Тихого океана, Южной Америке и западной части Атлантического океана (заключение Роскосмоса на основе отсутствия аппарата на орбите по данным от средств контроля космического пространства).

Фобос-Грунт (GeekTimes, IEEE Spectrum)

The report blames the loss of the probe on memory chips that became fatally damaged by cosmic rays. The probe died so suddenly that it didn't even send an error message, but investigators concluded the only plausible failure mechanism was the simultaneous disabling of two identical chips in the dual-computer control system, causing both to restart simultaneously. This in turn led to the autopilot going into "safe mode" while maintaining the spacecraft's orientation to the sun. (That reorientation was observed in the ensuing days as thruster firings disturbed the probe's orbit.)

Phobos-Grunt was supposed to await further instructions from Earth, but it never received them; **in an incredible design oversight, the probe could receive emergency instructions only after a successful departure from parking orbit.**

Toyota

- However, on October 24, 2013, a jury ruled against Toyota and found that unintended acceleration could have been caused due to deficiencies in the drive-by-wire throttle system or Electronic Throttle Control System (ETCS). Michael Barr of the Barr Group testified that NASA had not been able to complete its examination of Toyota's ETCS and that Toyota did not follow best practices for real time life critical software, and that a single bit flip which can be caused by cosmic rays could cause unintended acceleration. As well, the run-time stack of the real-time operating system was not large enough and that it was possible for the stack to grow large enough to overwrite data that could cause unintended acceleration. (Wikipedia)

Toyota (EDN Network)

Barr's ultimate conclusions were that:

- Toyota's electronic throttle control system (ETCS) source code is of unreasonable quality.
- Toyota's source code is defective and contains bugs, including bugs that can cause unintended acceleration (UA).
- Code-quality metrics predict presence of additional bugs.
- Toyota's fail safes are defective and inadequate (referring to them as a "house of cards" safety architecture).
  The ECM software formed the core of the technical investigation. What follows is a list of the key findings.
- Mirroring (where key data is written to redundant variables) was not always done. This gains extra significance in light of …
- Stack overflow. Toyota claimed only 41% of the allocated stack space was being used. Barr's investigation showed that 94% was closer to the truth. On top of that, stack-killing, MISRA-C rule-violating recursion was found in the code, and the CPU doesn't incorporate memory protection to guard against stack overflow.

Toyota

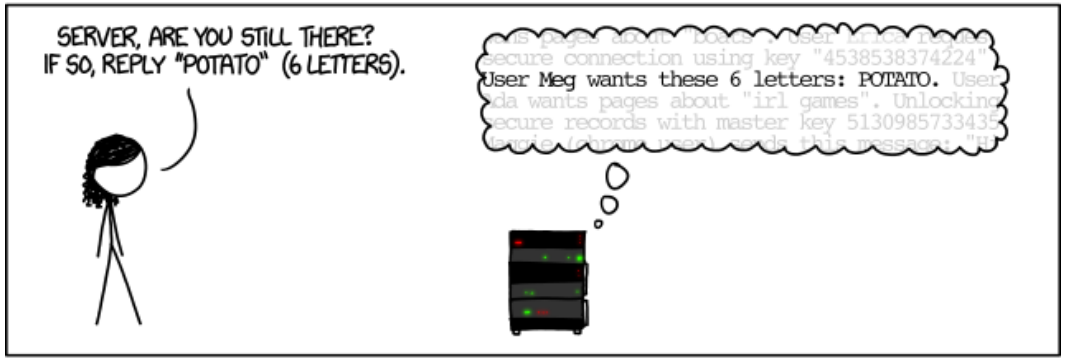
- Two key items were not mirrored: The RTOS' critical internal data structures; and—the most important bytes of all, the final result of all this firmware—the TargetThrottleAngle global variable.
- Although Toyota had performed a stack analysis, Barr concluded the automaker had completely botched it. Toyota missed some of the calls made via pointer, missed stack usage by library and assembly functions (about 350 in total), and missed RTOS use during task switching. They also failed to perform run-time stack monitoring.
- Unintentional RTOS task shutdown was heavily investigated as a potential source of the UA. As single bits in memory control each task, corruption due to HW or SW faults will suspend needed tasks or start unwanted ones. Vehicle tests confirmed that one particular dead task would result in loss of throttle control, and that the driver might have to fully remove their foot from the brake during an unintended acceleration event before being able to end the unwanted acceleration.
- A litany of other faults were found in the code, including buffer overflow, unsafe casting, and race conditions between tasks.

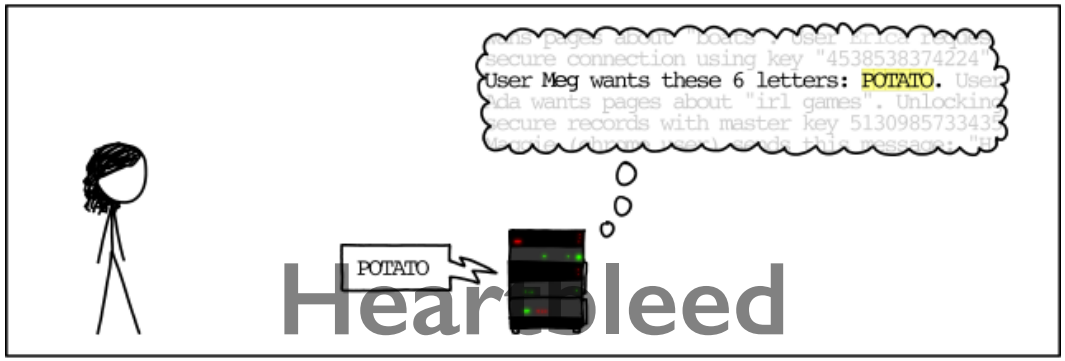
Airbus (ArsTechnica)

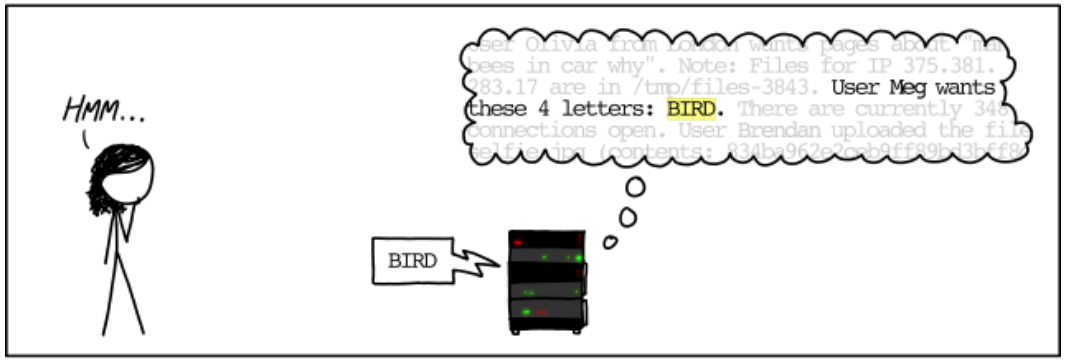
As Ars reported on May 19, Airbus had issued a warning to its military customers about a potential software problem in the engine control software for the A400M. The release of the exact cause of the crash, however, had been delayed because a Spanish magistrate placed the flight data recorders from the aircraft under seal. Airbus has since been able to obtain the flight data, which Lahoud said confirms that the engine control software had been improperly configured during the installation of the engines on the ill-fated aircraft.

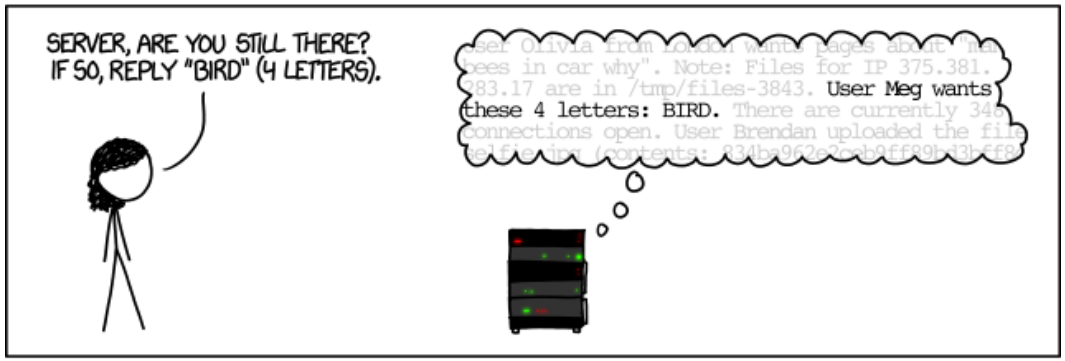
"The black boxes attest to that," Lahoud told Handelsblatt. "There are no structural defects, but we have a serious quality problem in the final assembly." The error was not in the code itself, but in configuration settings programmed into the electronic control unit (ECU) of the engines.

HOW THE HEARTBLEED BUG WORKS:

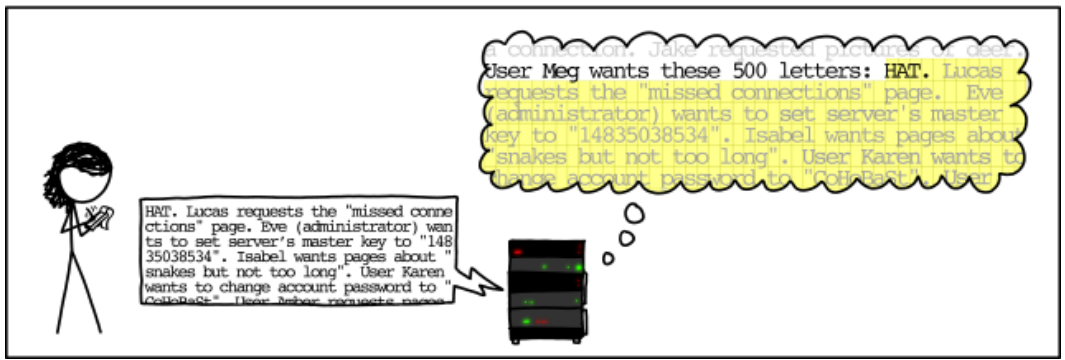








Are you still there, server? It's me, Margaret.



Heartbleed (OpenSSL Git)

```
diff --git a/ssl/d1_both.c b/ssl/d1_both.c
index 7a5596a..2e8cf68 100644 (file)
@@ -1459,26 +1459,36 @@ dtls1_process_heartbeat(SSL *s)
        unsigned int payload;
        unsigned int padding = 16; /* Use minimum padding */

-       /* Read type and payload length first */
-       hbtype = *p++;
-       n2s(p, payload);
-       pl = p;
-
// ...
+       /* Read type and payload length first */
+       if (1 + 2 + 16 > s->s3->rrec.length)
+               return 0; /* silently discard */
+       hbtype = *p++;
+       n2s(p, payload);
+       if (1 + 2 + payload + 16 > s->s3->rrec.length)
+               return 0; /* silently discard per RFC 6520 sec. 4 */
+       pl = p;
```

Необходим контроль качества программных продуктов

верификация

процесс, целью которого является показать соответствие продукта, сервиса или системы требованиям, спецификациям и другим условиям, которые накладываются на продукт

валидация

процесс, целью которого является определить адекватность продукта, сервиса или системы потребностям заказчика

Рецензирование (code review)

— экспертный контроль качества кода.

1. Формализованный процесс (например, Fagan inspection)
   - отдельная стадия разработки
   - большое количество участников и ролей
   - несколько этапов
   - критерии
2. Неформализованный процесс
   - рецензирование «из-за спины»
   - коммиты по email
   - парное программирование
   - средства рецензирования (например, Reviewable)
   - pull-requests

Рецензирование в Mozilla

Проект Servo (перспективный параллельный движок Web-браузера)

- разработка ведётся на GitHub
- все изменения в код вносятся в отдельных ветках, которые вливаются в основную после рецензирования

Пример: #7204 Very basic touch events and touch scrolling
Пример рецензии для того же PR: servo/servo #7204 @ Reviewable.io

Рецензирование в Chromium

Пример PR/рецензии: https://codereview.chromium.org/1234223005/#ps260001

Рецензирование в ядре ОС Linux

Пример PR/рецензии: https://lkml.org/lkml/2015/10/1/521

Средства рецензирования в GitHub

https://github.com/atom/atom/pull/6712#discussion_r33490479

(Динамическое) Тестирование

- процесс исследования, выполняемый с целью предоставления «заказчикам» информации о качестве продукта или сервиса.
- (отдельная большая тема)
- классификации:
  - ручное или **автоматизированное**
  - методы чёрного/белого/серого ящика
  - юнит-тестирование, интеграционное, интерфейсное, системное, приёмочное
  - регрессионное тестирование
  - smoke-тестирование, sanity-тестирование
  - функциональное или нефункциональное тестирование (производительность, безопасность, живучесть, удобство использования)

Автоматизированные юнит-тесты

```
describe("simple parsers", function() {
  it("regex", () => {
    const p = pr.regex(/h(i)?/);
    (() => p.run("no")).should.throw(/h\(i\)\?/);
    const rv = p.execute("hit");
    rv.state.pos.should.equal(2);
    rv.value[0].should.eql("hi");
    rv.value[1].should.eql("i");
  });
})
```

```
simple parsers
  ✓ reject
  ✓ succeed
  ✓ end
  ✓ literal string
  ✓ consumes the whole string
  36) regex
```

```
  36) simple parsers regex:

      AssertionError: expected 0 to be 2
      + expected - actual

      -0
      +2

      at Assertion.fail
(node_modules/should/lib/assertion.js:180:17)
      at Assertion.prop.value
(node_modules/should/lib/assertion.js:65:17)
      at Context.<anonymous>
(test/src/test_simple.js:43:25)
```

TDD / BDD

Test Driven Development / Behaviour Driven Development

Перед началом разработки модуля кода разработчик пишет примеры использования этого модуля в форме тестов. Разработка модуля условно завершается, когда все тесты корректно выполняются.

Пример: https://github.com/orfjackal/tdd-tetris-tutorial
— пишем Tetris на Java с использованием TDD (есть скринкасты)

1. FallingBlocksTest
2. RotatingPiecesOfBlocksTest
3. RotatingTetrominoesTest
4. FallingPiecesTest
5. MovingAFallingPieceTest
6. RotatingAFallingPieceTest
7. And beyond…

Регрессионные тесты

— тесты, задача которых — проверять, что при внесении изменений существующая функциональность осталась неизменной и что ранее устранённые ошибки не возвращаются.

Применимы в том числе и к нефункциональным требованиям (производительность).

Приблизительная схема работы:

1. Получаем сообщение об ошибке.
2. Воспроизводим ошибку.
3. Пишем тест, который автоматически воспроизводит ошибку.
4. ???
5. Всё работает.

Пример:

- багрепорт: https://github.com/meteor/meteor/issues/2691
- тест: https://github.com/meteor/meteor/pull/5038

Формальные методы в тестировании

model-based testing или specification-based testing: тесты генерируются автоматически на основе спецификации программы

- библиотека QuickCheck ЯП Haskell (и производные для большого количества других ЯП)
- теория конформности (И.Б. Бурдонов, ИСП РАН)

Fuzzing

— автоматическое или полуавтоматическое тестирование на большом наборе корректно (или некорректно) сгенерированных входных данных.

тестирование на основе спецификации входных данных:

- Csmith, генератор случайных корректных C-программ для тестирования компиляторов
- http://www.cs.utah.edu/~regehr/papers/pldi13.pdf

Статический анализ

- автоматизированная проверка исходного кода (статический = без запуска программы) алгоритмом, который предназначен для поиска того или иного класса дефектов.
- фундаментальное ограничение: автоматическое доказательство свойств произвольных программ неразрешимо.

Одно из основных свойств — достоверность (soundness) или недостоверность.

Достоверные методы статического анализа не дают ложных негативных результатов (но дают ложные позитивные).

Примеры недостоверных средств

Коммерческие средства:

- Coverity — использовалась для статистики на первом слайде
- PVS-Studio (российская разработка)
  Пример обнаруженной ошибки: использование функции memset перед тем, как буфер выходит из области видимости, может быть проигнорировано компилятором.

```
static int crypt_iv_tcw_whitening(....) {
  ....
  u8 buf[TCW_WHITENING_SIZE];
  ....
  out:
  memset(buf, 0, sizeof(buf));
  return r;
```

Средства с открытым исходным кодом:

- Clang

Статический анализ — достоверные средства

- Frama-C, OpenSource, пример

```
int max_array(int* a, int length) {
  int m = a[0];
  /*@
  loop invariant 0<=i<=length; loop invariant
  \forall integer j; 0<=j<i ==> m >= a[j]; loop invariant
  \exists integer j; 0<=j<i && m == a[j]; loop assigns i,m;
  */
  for (int i = 1; i<length; i++) {
    if (a[i] > m) m = a[i]; }
  return m;
```

- Polyspace, коммерческое средство от разработчиков MATLAB, на сайте есть примеры

Динамический анализ

- автоматизированная проверка кода и состояния программы в процессе исполнения алгоритмом, который предназначен для поиска того или иного класса дефектов.
- вносит изменения в процесс исполнения программы.
- в целом, имеет те же фундаментальные ограничения, что и статический анализ.
- пример: valgrind

Распространённые подходы не гарантируют отсутствия ошибок

- Визуальный контроль
  - зависит от экспертного мнения;
  - не автоматизирован;
  - на больших объёмах кода сложен в применении.
- (динамическое автоматизированное) тестирование
  - большой объём кода тестов по сравнению с кодом продукта (SQLite: 80 тыс. строк кода / 90 млн. строк тестов);
  - тесты необходимо поддерживать в актуальном состоянии;
  - тесты не гарантируют отсутствия дефектов/ошибок.
- Статический и динамический анализ
  - нетривиальные свойства неразрешимы согласно теореме Райса;
  - большое количество ложных срабатываний;
  - на практике встречаются дефекты, необнаружимые средствами статического анализа (OpenSSL Heartbleed bug: информация раскрыта 7 апреля 2014 г., адаптированные средства статического анализа появились 18 апреля 2014 г.).

Формальная верификация

- процесс, результатом которого является получение строгого математического доказательства соответствия программы требуемой спецификации, наличия у программы требуемых свойств.

Недостатки:

- является трудоёмким и наукоёмким процессом
- не имеет широкого распространения (в отличие от формальной верификации аппаратного обеспечения)

Область применимости формальной верификации:

- риски, превышающие затраты на верификацию
- сложность системы

Разработку программы необходимо вести с учётом требований по её формальной верификации.

SeL4 — 3 тыс. строк формальных спецификаций, 10 тыс. строк кода на C, 100 тыс. строк формальных доказательств.

Проблематика формальной верификации

- программа — это нестрого определённый объект реального мира (исполнимый код? исходный код? алгоритмы?)
- требования — это нестрого определённые представления заказчика о том, как должна выглядеть программа
- если достаточно детализировать спецификацию, не станет ли она сложнее, чем программа?
- не определено, чему можно доверять — и ОС, и стандартная библиотека, и компиляторы содержат ошибки и могут вносить ошибки в процесс работы программы

Системы типов и верификация

- в языке программирования со статической типизацией компилятор на этапе проверки типов выполняет работу статического анализатора — если компилятор принимает программу, значит она удовлетворяет спецификации с позиций безопасности типов
- «выразительные способности» систем типов разных языков программирования существенно различаются:
  - в динамических языках есть только один тип
  - на языке Go не получится создать пользовательский тип обобщённых списков, сохраняя строгую типизацию, тогда как в C, например, получится 1, 2
  - на языках Java, C#, C++ можно задавать сложные пользовательские обобщённые типы (`std::map`, `Dictionary<TKey, TValue>`)
  - язык Haskell поддерживают полноценный полиморфизм с широкими возможностями по выводу типов

```haskell
instance (Eq a, Ord a)  => Eq (Set a) where
  (Set xs) == (Set ys) = (sort xs) == (sort ys)
```

План на следующее занятие

Фундаментальные модели вычислений, понятие об их эквивалентности

- как вышло, что модели вычислений предшествовали появлению программируемых вычислительных устройств?
- основные модели — автоматы и машина Тьюринга, λ-исчисление, частично-рекурсивные функции, нормальные алгорифмы Маркова
- менее известные модели вычислений
- (?) интерактивные демонстрации
- понятие эквивалентности фундаментальных моделей вычислений

Задачи «со звёздочкой»

Задача 1.1 **

Статистика по использованию языков программирования по GitHub (современная или более полная), BitBucket, Launchpad, каким-то другим популярным OpenSource-репозиториям.

Задача 1.2а *

Рецензия на фрагмент кода 50-150 строк. Код должен представлять законченный фрагмент функциональности. Оформить рецензию в форме комментариев к коду.

Задача 1.2б **

Аналогичное задание, но в парах — обмениваетесь с партнёром фрагментом самостоятельно написанного кода и пишете рецензию.

Задача 1.2в ***

Отправить pull-request в проект с открытым исходным кодом, в котором используют рецензирование (принимается ретроспектива).

Задачи «со звёздочкой»

Задача 1.3а *

Написать набор тестов к своему коду (принимается ретроспектива).

Задача 1.3б *

Бонус к задаче **1.2в**, если в pull-request есть тесты.

Задача 1.3в **

Выполнить fuzzing-тестирование своего кода.

Задача 1.3г ***

Выполнить fuzzing-тестирование для проекта с открытым исходным кодом.

Задача 1.4а **

Выполнить проверку своего кода средствами статического анализа.

Задача 1.4б ***

Выполнить проверку средствами статического анализа проекта с открытым исходным кодом.