CS4004/CS4504: FORMAL VERIFICATION

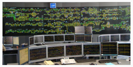
Lecture 1: Module Overview & Introduction

Vasileios Koutavas

School of Computer Science and Statistics
Trinity College Dublin

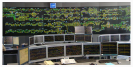
THE PERILS OF SOFTWARE

- Software is now controlling critical machines:
  - Transportation: cars ($>$ 100M LoC [IEEE]), airplains, trains, spacecraft, …
  - Medical: pacemakers, MRI machines, …
  - Utilities: power grids, telephone centres, …
  - Finance: online banking, stock prices, …
  - …

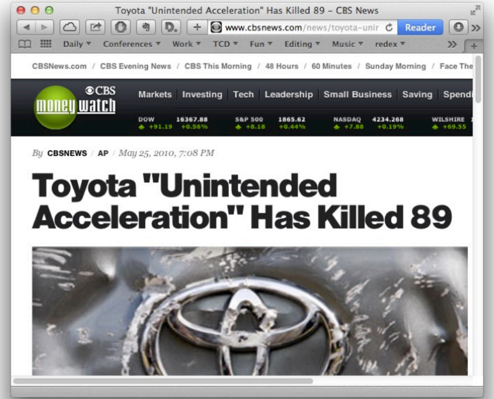


THE PERILS OF SOFTWARE

- Software is now controlling critical machines:
  - Transportation: cars ($> 100M$ LoC [IEEE]), airplains, trains, spacecraft, ...
  - Medical: pacemakers, MRI machines, ...
  - Utilities: power grids, telephone centres, ...
  - Finance: online banking, stock prices, ...
  - ...
- BUT Software is very unreliable
  - `http://en.wikipedia.org/wiki/List_of_software_bugs`
  - `http://www.cs.tau.ac.il/~nachumd/horror.html`



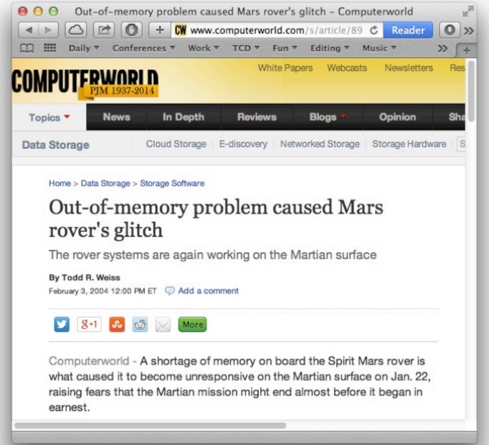
THE PERILS OF SOFTWARE


Toyota "Unintended Acceleration" Has Killed 89 – CBS News
www.cbsnews.com/news/toyota-unir
Reader
Daily
Conferences
Work
TCD
Fun
Editing
Music
redex
CBSNews.com / CBS Evening News / CBS This Morning / 48 Hours / 60 Minutes / Sunday Morning / Face The
CBS money watch
Markets | Investing | Tech | Leadership | Small Business | Saving | Spendi
DOW 16367.88 +91.19 +0.56%
S&P 500 1865.62 +8.18 +0.44%
NASDAQ 4234.268 +7.88 +0.19%
WILSHIRE +69.55

By CBSNEWS / AP / May 25, 2010, 7:08 PM

Toyota "Unintended Acceleration" Has Killed 89

THE PERILS OF SOFTWARE



Out-of-memory problem caused Mars rover's glitch

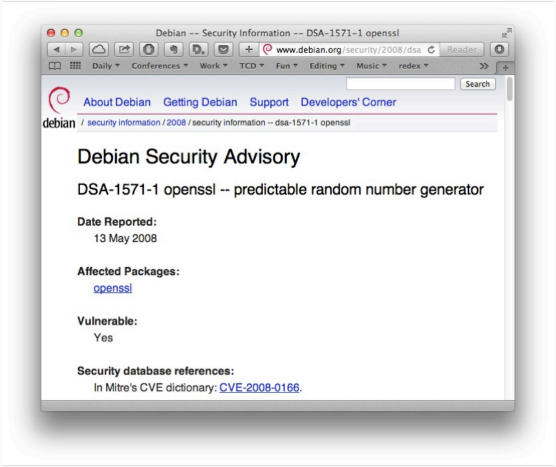
The rover systems are again working on the Martian surface

By Todd R. Weiss

February 3, 2004 12:00 PM ET Add a comment

Computerworld - A shortage of memory on board the Spirit Mars rover is what caused it to become unresponsive on the Martian surface on Jan. 22, raising fears that the Martian mission might end almost before it began in earnest.

THE PERILS OF SOFTWARE


Debian -- Security Information -- DSA-1571-1 openssl
www.debian.org/security/2008/dsa
Reader
Daily
Conferences
Work
TCD
Fun
Editing
Music
redex
Search
About Debian
Getting Debian
Support
Developers' Corner
debian
/ security information / 2008 / security information -- dsa-1571-1 openssl

Debian Security Advisory

DSA-1571-1 openssl -- predictable random number generator

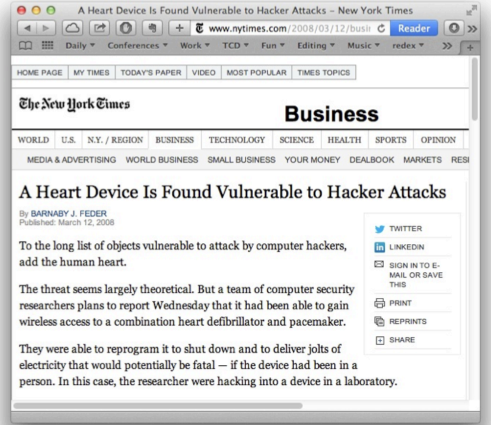
Date Reported:
13 May 2008

Affected Packages:
openssl

Vulnerable:
Yes

Security database references:
In Mitre's CVE dictionary: CVE-2008-0166.

THE PERILS OF SOFTWARE



A Heart Device Is Found Vulnerable to Hacker Attacks

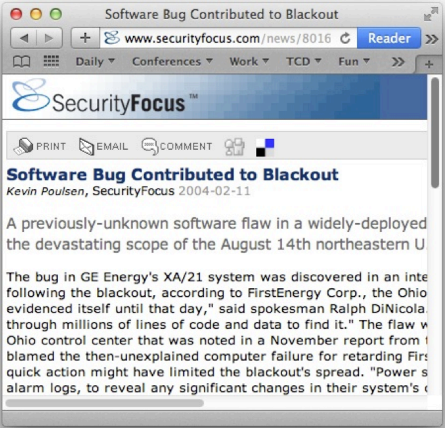
By BARNABY J. FEDER
Published: March 12, 2008

To the long list of objects vulnerable to attack by computer hackers, add the human heart.

The threat seems largely theoretical. But a team of computer security researchers plans to report Wednesday that it had been able to gain wireless access to a combination heart defibrillator and pacemaker.

They were able to reprogram it to shut down and to deliver jolts of electricity that would potentially be fatal — if the device had been in a person. In this case, the researcher were hacking into a device in a laboratory.

THE PERILS OF SOFTWARE



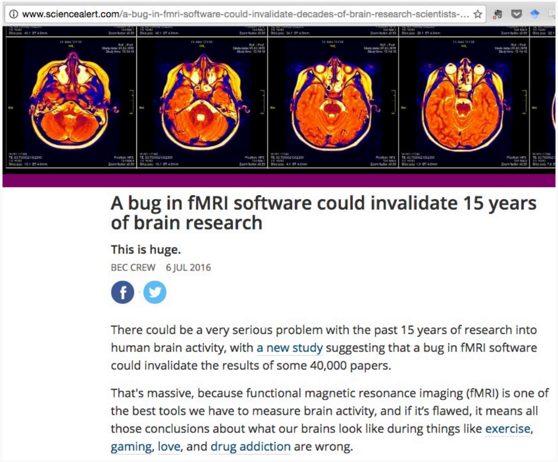
Software Bug Contributed to Blackout
www.securityfocus.com/news/8016
Reader
Daily
Conferences
Work
TCD
Fun
SecurityFocus
PRINT
EMAIL
COMMENT

Software Bug Contributed to Blackout
Kevin Poulsen, SecurityFocus 2004-02-11

A previously-unknown software flaw in a widely-deployed
the devastating scope of the August 14th northeastern U

The bug in GE Energy's XA/21 system was discovered in an inte
following the blackout, according to FirstEnergy Corp., the Ohio
evidenced itself until that day," said spokesman Ralph DiNicola
through millions of lines of code and data to find it." The flaw w
Ohio control center that was noted in a November report from
blamed the then-unexplained computer failure for retarding Firs
quick action might have limited the blackout's spread. "Power s
alarm logs, to reveal any significant changes in their system's

THE PERILS OF SOFTWARE


www.sciencealert.com/a-bug-in-fmri-software-could-invalidate-decades-of-brain-research-scientists-...

A bug in fMRI software could invalidate 15 years of brain research

This is huge.

BEC CREW 6 JUL 2016

There could be a very serious problem with the past 15 years of research into human brain activity, with a new study suggesting that a bug in fMRI software could invalidate the results of some 40,000 papers.

That's massive, because functional magnetic resonance imaging (fMRI) is one of the best tools we have to measure brain activity, and if it's flawed, it means all those conclusions about what our brains look like during things like exercise, gaming, love, and drug addiction are wrong.

TIMSORT

- → Hybrid Sorting Algorithm of MergeSort + InsertionSort
- → Used in Android JDK, Sun JDK, OpenJDK, Python, GNU Octave

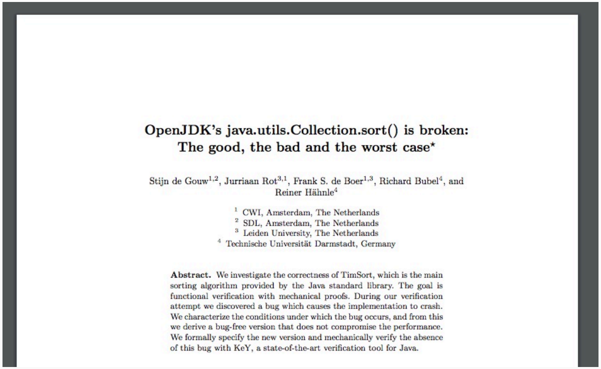
TIMSORT

- → Hybrid Sorting Algorithm of MergeSort + InsertionSort
- → Used in Android JDK, Sun JDK, OpenJDK, Python, GNU Octave

- → The implementation uses an intermediate array of fixed size
- → The (wrong) assumption is that the array will never fill up
- → But it **does** fill up for carefully selected inputs arrays of size > 562 **trillion**

http://www.envisage-project.eu/

proving-android-java-and-python-sorting-algorithm-is-broken-and-how-to-fix-it/



OpenJDK's java.utils.Collection.sort() is broken: The good, the bad and the worst case*

Stijn de Gouw[1,2], Jurriaan Rot[3,1], Frank S. de Boer[1,3], Richard Bubel[4], and Reiner Hähnle[4]

[1] CWI, Amsterdam, The Netherlands
[2] SDL, Amsterdam, The Netherlands
[3] Leiden University, The Netherlands
[4] Technische Universität Darmstadt, Germany

Abstract. We investigate the correctness of TimSort, which is the main sorting algorithm provided by the Java standard library. The goal is functional verification with mechanical proofs. During our verification attempt we discovered a bug which causes the implementation to crash. We characterize the conditions under which the bug occurs, and from this we derive a bug-free version that does not compromise the performance. We formally specify the new version and mechanically verify the absence of this bug with KeY, a state-of-the-art verification tool for Java.

FORMAL VERIFICATION

FORMAL VERIFICATION

→ Ultimate goal: prove the absence of software errors

FORMAL VERIFICATION

- Ultimate goal: prove the absence of software errors
- Easy? NO!
  - Finite code often has an infinite set of behaviours:

    ```
    quicksort (a: array int)
    ```

    Infinite number of `a`-inputs and outputs

FORMAL VERIFICATION

- Ultimate goal: prove the absence of software errors
- Easy? NO!
  - Finite code often has an infinite set of behaviours:

    ```
    quicksort (a: array int)
    ```

    Infinite number of `a`-inputs and outputs
  - What does it even mean for `quicksort` to be correct?

FORMAL VERIFICATION

- Ultimate goal: prove the absence of software errors
- Easy? NO!
  - Finite code often has an infinite set of behaviours:

    ```
    quicksort (a: array int)
    ```

    Infinite number of `a`-inputs and outputs
  - What does it even mean for `quicksort` to be correct?
- A number of ways to approach the problem.

METHODS FOR FORMAL VERIFICATION

→ Can we use **testing** to prove correctness/incorrectness?

METHODS FOR FORMAL VERIFICATION

→ Can we use **testing** to prove correctness/incorrectness?

We can use testing to prove **in**correctness: provide a failing test.

METHODS FOR FORMAL VERIFICATION

→ Can we use testing to prove correctness/incorrectness?

We can use testing to prove incorrectness: provide a failing test.

To prove correctness we neet to test all possible input-output pairs (infinite testsuite).

METHODS FOR FORMAL VERIFICATION

→ Algorithmic Verification: Can we come up with an **algorithm** to do prove correctness/incorrectness automatically?

  That is, create a program `av` that inputs another program `p` and after finite time outputs `false` if `p` has a bug for some input, or `true` otherwise.

METHODS FOR FORMAL VERIFICATION

→ Algorithmic Verification: Can we come up with an **algorithm** to do prove correctness/incorrectness automatically?

That is, create a program `av` that inputs another program `p` and after finite time outputs `false` if `p` has a bug for some input, or `true` otherwise.

NO! If we could create `av` we could solve the **halting problem** [Alan Turing 1936].

METHODS FOR FORMAL VERIFICATION

- Deductive Verification: Can we have a **mathematical proof system** to prove correctness/incorrectness for all programs?

  Create a system **L** of logical axioms and rules, such that for any program **p** we can write a **finite proof** that shows either
  - **p** has a bug for some input
  - **p** has no bug for any input

METHODS FOR FORMAL VERIFICATION

- Deductive Verification: Can we have a **mathematical proof system** to prove correctness/incorrectness for all programs?

  Create a system **L** of logical axioms and rules, such that for any program **p** we can write a **finite proof** that shows either

  - **p** has a bug for some input
  - **p** has no bug for any input

  NO! Kurt Gödel proved in 1931 that no such logical system exists.

METHODS FOR FORMAL VERIFICATION

- Deductive Verification: Can we have a **mathematical proof system** to prove correctness/incorrectness for all programs?

  Create a system **L** of logical axioms and rules, such that for any program **p** we can write a **finite proof** that shows either

  - **p** has a bug for some input
  - **p** has no bug for any input

  NO! Kurt Gödel proved in 1931 that no such logical system exists.

  If such a system **L** exists then we can create a fully automatic verification algorithm (simply systematically explore all logical derivations and eventually, in finite time, derive "**p** has a bug" or "**p** has no bug".)

HOLY GRAIL OF FORMAL VERIFICATION

A software verification method that is:

- → Sound: If the verification method returns **yes**, then indeed the program under examination **has no bug**
- → Complete: If the verification method returns **no**, then indeed the program under examination **has a bug**
- → Terminating: The verification method always terminates, returning **yes** or **no**.

HOLY GRAIL OF FORMAL VERIFICATION

A software verification method that is:

→ Sound: If the verification method returns **yes**, then indeed the program under examination **has no bug**

→ Complete: If the verification method returns **no**, then indeed the program under examination **has a bug**

→ Terminating: The verification method always terminates, returning **yes** or **no**.

Unfortunately we can only pick TWO. A verification method with all three properties is theoretically impossible.

Usually verification systems pick **soundness** and **termination**.

METHODS FOR FORMAL VERIFICATION

However not all is lost! Sound and terminating systems can prove the correctness of *virtually every program we would care about.*

The scientific community continuously pushes the limits of these systems.

- → **Algorithmic verification**: model checking, abstract interpretation, static analysis, type systems.
  - → create a model of the program in a **decidable framework** (finite state system, pushdown system)
  - → building the model may require some user input
  - → "push-button" verification of correctness
- → **Deductive verification**:
  - → create a correctness proof of the program in a **logic** (with axioms and logical rules)
  - → constructing the proof likely to require user input
  - → once the proof is constructed, checking its validity is automatic

METHODS FOR FORMAL VERIFICATION

However not all is lost! Sound and terminating systems can prove the correctness of *virtually every program we would care about*.

The scientific community continuously pushes the limits of these systems.

- → **Algorithmic verification**: model checking, abstract interpretation, static analysis, type systems.
  - → create a model of the program in a **decidable framework** (finite state system, pushdown system)
  - → building the model may require some user input
  - → "push-button" verification of correctness
- → **Deductive verification**:
  - → create a correctness proof of the program in a **logic** (with axioms and logical rules)
  - → constructing the proof likely to require user input
  - → once the proof is constructed, checking its validity is automatic

In this module: we will use the second approach.

- → We will manually prove programs correct using pen and paper proofs.
- → We will use software that provide *some* automation to make this easier.

SUCCESS STORIES

Success stories start appearing from the mid-1990s.

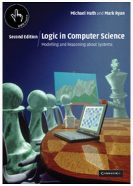
- → Paris metro line 14, (1998,refinement approach – combination of algorithmic and deductive approach)
  `http://www.methode-b.com/documentation_b/ClearSy-Industrial_Use_of_B.pdf`
- → Flight control software of A380: verified absence of run-time errors (2005, abstract interpretation) `http://www.astree.ens.fr/`
- → L4.verifiedmicro-kernel: a formally correct operating system kernel (2010, deductive verification) `http://www.ertos.nicta.com.au/research/l4.verified/`
- → SLAM: verifier for MS Windows drivers (2010, model checking)
  `http://research.microsoft.com/en-us/projects/slam/`
- → Facebook's Infer verifier: detects bugs in Android and iOS apps `http://fbinfer.com/`
- → And more...

THIS MODULE

CS4004/CS4504 THEORY

Book: Logic in Computer Science: Modelling and Reasoning about Systems, 2nd Edition, by M. Huth & M. Ryan

1. Symbolic logic
   - → Natural deduction
   - → Propositional logic
   - → First-order Predicate logic
2. Correctness of imperative programs
   - → Floyd-Hoare logic
   - → Weakest Precondition calculus
   - → Loop invariants
   - → functional abstractions, etc.



cs4004/cs45004: practice

Exercises, Tutorials and Assignments:

1. Pencil&paper proofs: propositional, inductive, correctness proofs.
2. Semi-automatic proofs using the **Dafny** verifier

   `http://research.microsoft.com/en-us/projects/dafny/` `http://rise4fun.com/Dafny`

cs4004/cs45004: other

Marks:

- → 2% marks from attendance (random sampling)
- → 33% marks from coursework
- → 65% marks from annual exam (before Christmas)
- → Second attempt supplementals: 100% exam

Final exam: pencil&paper proofs of simple propositional properties, and program correctness properties.

You will need to install Visual Studio + Dafny (in Windows) or VStudio Code + Dafny (in Linux/OS X).

Three lectures per week – mix of lecture and in-class tutorial.

More information here:
`www.scss.tcd.ie/Vasileios.Koutavas/teaching/cs4004-4504`