Between Testing and Formal Verification

Jan Tobias Mühlberg

jantobias.muehlberg@cs.kuleuven.be
imec-DistriNet, KU Leuven, Celestijnenlaan 200A, B-3001 Belgium

SecAppDev, Leuven, March 2017

How much testing do we have to do? When are we done?

```
int foo (bool a, bool b, bool c)
{
    int ret = 0;
    if ((a || b) && c)
    {
        ret = 1;
    }
    return ret;
}
```

How much testing do we have to do? When are we done?

- Function Coverage

  `foo(F, F, F);`

```
int foo (bool a, bool b, bool c)
{
    int ret = 0;
    if ((a || b) && c)
    {
        ret = 1;
    }
    return ret;
}
```

How much testing do we have to do? When are we done?

- Function Coverage

  `foo(F, F, F);`
- Statement Coverage

  `foo(T, T, T);`

```
int foo (bool a, bool b, bool c)
{
    int ret = 0;
    if ((a || b) && c)
    {
        ret = 1;
    }
    return ret;
}
```

How much testing do we have to do? When are we done?

- Function Coverage
  `foo(F, F, F);`
- Statement Coverage
  `foo(T, T, T);`
- Branch/Decision Coverage
  `foo(T, T, T);`
  `foo(T, T, F);`

```
int foo (bool a, bool b, bool c)
{
    int ret = 0;
    if ((a || b) && c)
    {
        ret = 1;
    }
    return ret;
}
```

How much testing do we have to do? When are we done?

- Function Coverage
  `foo(F, F, F);`
- Statement Coverage
  `foo(T, T, T);`
- Branch/Decision Coverage
  `foo(T, T, T);`
  `foo(T, T, F);`
- Condition Coverage
  `foo(F, F, T);`
  `foo(T, T, F);`

```
int foo (bool a, bool b, bool c)
{
    int ret = 0;
    if ((a || b) && c)
    {
        ret = 1;
    }
    return ret;
}
```

How much testing do we have to do? When are we done?

- Function Coverage

  `foo(F, F, F);`
- Statement Coverage

  `foo(T, T, T);`
- Branch/Decision Coverage

  `foo(T, T, T);`

  `foo(T, T, F);`
- Condition Coverage

  `foo(F, F, T);`

  `foo(T, T, F);`
- MC/DC

  `foo(F, T, F);`

  `foo(F, T, T);`

  `foo(F, F, T);`

  `foo(T, F, T);`

```c
int foo (bool a, bool b, bool c)
{
    int ret = 0;
    if ((a || b) && c)
    {
        ret = 1;
    }
    return ret;
}
```

How much testing do we have to do? When are we done?

- Function Coverage

  `foo(F, F, F);`
- Statement Coverage

  `foo(T, T, T);`
- Branch/Decision Coverage

  `foo(T, T, T);`

  `foo(T, T, F);`
- Condition Coverage

  `foo(F, F, T);`

  `foo(T, T, F);`
- MC/DC

  `foo(F, T, F);`

  `foo(F, T, T);`

  `foo(F, F, T);`

  `foo(T, F, T);`
- Multiple condition coverage, Parameter value coverage, ...

```c
int foo (bool a, bool b, bool c)
{
    int ret = 0;
    if ((a || b) && c)
    {
        ret = 1;
    }
    return ret;
}
```

How much testing do we have to do? When are we done?

```c
int bar (SSL *s)
{
  // ...
  unsigned char *buffer, *bp;
  int r;
  buffer = OPENSSL_malloc(1 +
    2 + payload + padding);
  bp = buffer;

  *bp++ = TLS1_HB_RESPONSE;
  s2n(payload, bp);
  memcpy(bp, pl, payload);

  r = ssl3_write_bytes(s,
    TLS1_RT_HEARTBEAT, buffer,
    3 + payload + padding);
  // ...
}
```

How much testing do we have to do? When are we done?

- Which criterion is best?
- What about code that doesn't branch?
- What about code that is stimulated by I/O?
- ... in scenarios that you can't set up in the lab (SDI, Delta Works)?
- How do we know that we haven't missed critical interactions? Concurrency?
- Who writes all these tests?
- What about security properties?

```
int bar (SSL *s)
{
  // ...
  unsigned char *buffer, *bp;
  int r;
  buffer = OPENSSL_malloc(1 +
    2 + payload + padding);
  bp = buffer;

  *bp++ = TLS1_HB_RESPONSE;
  s2n(payload, bp);
  memcpy(bp, pl, payload);

  r = ssl3_write_bytes(s,
    TLS1_RT_HEARTBEAT, buffer,
    3 + payload + padding);
  // ...
}
```

QUANTA MAGAZINE illuminating science

PHYSICS MATHEMATICS BIOLOGY COMPUTER SCIENCE BLOG MORE ALL SUBSCRIBE

COMPUTER SECURITY

Hacker-Proof Code Confirmed

Computer scientists can prove certain programs to be error-free with the same certainty that mathematicians prove theorems. The advances are being used to secure everything from unmanned drones to the internet.

QUANTA MAGAZINE illuminating science

PHYSICS MATHEMATICS BIOLOGY COMPUTER SCIENCE BLOG MORE ALL SUBSCRIBE

COMPUTER SECURITY

Hacker-Proof Code Confirmed

Computer scientists can prove certain programs to be error-free with the same certainty that mathematicians prove theorems. The advances are being used to secure everything from unmanned drones to the internet.

Java Bug Fixed with Formal Methods CWI

Mon, 23/02/2015 - 13:16

Deze pagina in het Nederlands: Bug in Java gefixt met formele methoden CWI

Researchers from CWI fixed a bug in the widely used object-oriented programming language Java in February 2015. They found an error in a broadly applied sorting algorithm, TimSort, which could crash programs. The bug had already been known from 2013 but was never correctly resolved. When researcher Stijn de Gouw from the CWI research group Formal Methods attempted to prove the correctness of TimSort, he encountered the bug that could threaten the security. He filed a bug report with an improved version, which has now been accepted. This version of TimSort is used by Android.

Java is used for server software, Internet-based banking services and, for instance, in computer games like Minecraft. The programming language is broadly used because it provides a lot of support in the form of libraries. Developers don't have to invent a function to sort data, for instance, since they can simply get it from the library support. The sorting algorithm TimSort is part of the java.util.Arrays and java.util.Collections libraries. It is named after its creator, Tim Peters, who designed it in 2002 for the Python programming language, where it is now the default sorting algorithm. The sorting function is often used, for example in the analysis of data. De Gouw discovered that a previous fix of the error was wrong. The bug causes programs to crash when used on a large input that is sorted in a specific way.

QUANTA MAGAZINE illuminating science

PHYSICS MATHEMATICS BIOLOGY COMPUTER SCIENCE BLOG MORE ALL SUBSCRIBE

COMPUTER SECURITY

Hacker-Proof Code Confirmed

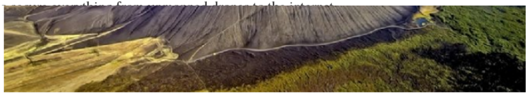
Computer scientists can prove certain programs to be error-free with the same certainty that mathematicians prove theorems. The advances are being used t

Java Bug Fixed with Form

Mon, 23/02/2015 - 13:16

Deze

Java is used for server software, Internet-
Minecraft. The programming language is
libraries. Developers don't have to invent
the library support. The sorting algorithm
It is named after its creator, Tim Peters, w
it is now the default sorting algorithm. The
De Gouw discovered that a previous fix of
on a large input that is sorted in a specific



http://www.geologyin.com/2016/09/what-is-difference-between-active-and.html

A 12 year Dormant Error found in just 1.474 seconds!!

Published on February 10, 2017

Yogananda Jeppu Follow
Principal Systems Engineer at Honeywell Technology Solutio...
101 12 25

When I wrote this article in IEEE saying that errors can remain dormant for a long time waiting for an opportunity to surface I did not have this example. I wrote this blog in 2014 describing the error. I repeat it here again.

In 2014 I got a call from a friend – they had found an error in the integrator reset which caused a channel failure in a safety critical control system. This has been working for the last 12 years wit

QUANTA MAGAZINE illuminating science

PHYSICS MATHEMATICS BIOLOGY COMPUTER SCIENCE BLOG MORE ALL SUBSCRIBE

COMPUTER SECURITY

Hacker-Proof Code Confirmed

Computer scientists can prove certain programs to be error-free with the same certainty that mathematicians prove theorems. The advances are being used

Java Bug Fixed with Form

Mon, 23/02/2015 - 13:16

http://www.geologyin.com/2016/09/what-is-difference-between-active-and.html

COMMUNICATIONS OF THE ACM

HOME CURRENT ISSUE NEWS BLOGS OPINION RESEARCH PRACTICE

Home / Magazine Archive / April 2015 (Vol. 58, No. 4) / How Amazon Web Services Uses Formal Methods / Full Text

CONTRIBUTED ARTICLES

How Amazon Web Services Uses Formal Methods

By Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, Michael Deardeuff

Communications of the ACM, Vol. 58 No. 4, Pages 66-73

10.1145/2699417

Comments (1)

VIEW AS: SHARE:

Since 2011, engineers at Amazon Web Services (AWS) have used formal specification and model checking to help solve difficult design problems in critical systems. Here, we describe our motivation and experience, what has worked well in our problem domain, and what has not.

nd in just

101 12 25

rors can remain

ity to surface I did

describing the error.

nd an error in the

in a safety critical

ist 12 years wit

QUANTA MAGAZINE illuminating science

PHYSICS MATHEMATICS BIOLOGY COMPUTER SCIENCE BLOG MORE ALL SUBSCRIBE

COMPUTER SECURITY

Hacker-Proof Co

Computer scientists can prove certain p
theorems. The advances are being used

Java Bug Fixed with Form

Mon, 23/02/2015 - 13:16

COMMUNICATIONS OF THE ACM

HOME

Home / Magazine Archive / April 2015 (Vol. 58, I

CONTRIBUTED ARTICLES

How Amazon Web

By Chris Newcombe, Tim Rath, Fan Zhang, Bogda
Communications of the ACM, Vol. 58 No. 4, Pages
10.1145/2699417
Comments (1)

VIEW AS: S

COMMUNICATIONS OF THE ACM

HOME CURRENT ISSUE NEWS BLOGS OPINION RESEARCH PRACT

Home / Magazine Archive / February 2010 (Vol. 53, No. 2) / Software Model Checking Takes Off / Full Text

PRACTICE

Software Model Checking Takes Off

By Steven P. Miller, Michael W. Whalen, Darren D. Cofer
Communications of the ACM, Vol. 53 No. 2, Pages 58-64
10.1145/1646353.1646372
Comments

VIEW AS: SHARE:

Although formal methods have been used in the development of safety- and security-critical systems for years, they have not achieved widespread industrial use in software or systems engineering. However, two important trends are making the industrial use of formal methods practical. The first is the growing acceptance of model-based development for the design of embedded systems. Tools such as MATLAB Simulink[6] and Esterel Technologies SCADE Suite[2] are achieving widespread use in the design of avionics and automotive systems. The graphical models produced by these tools provide a formal, or nearly formal, specification that is often amenable to formal analysis.

The second is the growing power of formal verification tools, particularly model checkers. For many classes of models they provide a "push-button" means of determining if a model meets its requirements. Since these tools examine all possible combinations of inputs and state, they are much more likely to find design errors than testing.


QUANTA MAGAZINE
illuminating science
PHYSICS MATHEMATICS BIOLOGY COMPUTER SCIENCE BLOG MORE ALL SUBSCRIBE
COMPUTER SECURITY
Hacker-Proof Co
Computer scientists can prove cert
theorems. The advances are bei
Java Bug Fixed wit
Mon, 23/02/2015 - 13:16
COMMUNICATI
OF THE ACM
Home / Magazine Archive / April 2015 (Vol. 58
CONTRIBUTED ARTICLES
How Amazon Web
By Chris Newcombe, Tim Rath, Fan Zhang, Bog
Communications of the ACM, Vol. 58 No. 4, Pages
10.1145/2699417
Comments (1)
VIEW AS:
COMMUNICATIONS
ACM
BLOGS OPINION RESEARCH PRACT
Home / Magazine / Software / Full Text
PRACTICE
Software Model Checking Tak Off
By Steven P. Miller, Michael W. Whalen, Darren D. Cofer
Communications of the ACM, Vol. 53 No. 2, Pages 58-64
10.1145/1646353.1646372
Comments
SHARE:
methods have been used in the
of safety- and security-critical systems for
have not achieved widespread industrial use
are or systems engineering. However, two
rtant trends are making the industrial use of formal
ethods practical. The first is the growing acceptance of
model-based development for the design of embedded
systems. Tools such as MATLAB Simulink6 and Esterel
Technologies SCADE Suite2 are achieving widespread use
in the design of avionics and automotive systems. The
graphical models produced by these tools provide a
formal, or nearly formal, specification that is often
amenable to formal analysis.
The s cond is the growing powe f nal verification tools, particularly model checkers. For
man lasses of models they pr de a "push-button" means of determining if a model meets its
requirements. Since these tools amine all possible combinations of inputs and state, they are
much more likely to find design rors than testing.

Between Testing and Formal Verification

Testing	Formal Verification
• Find as many defects as reasonably possible	Use mathematical methods to convincingly argue that a system is free of defects
• Gather evidence to show that a specification is correctly implemented	Prove that implementation is a refinement of the specification
• Relies on empirical evidence and intuition	Aims to be exhaustive and complete
• Expensive	Maybe more expensive

Iterative and Incremental Development

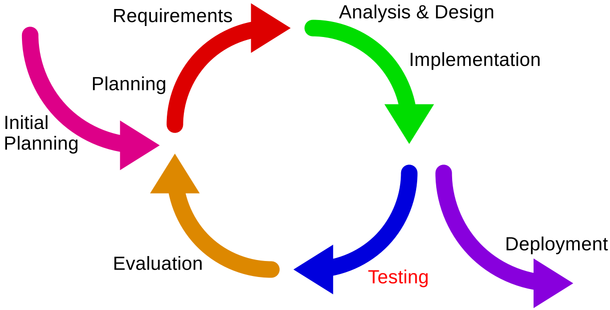
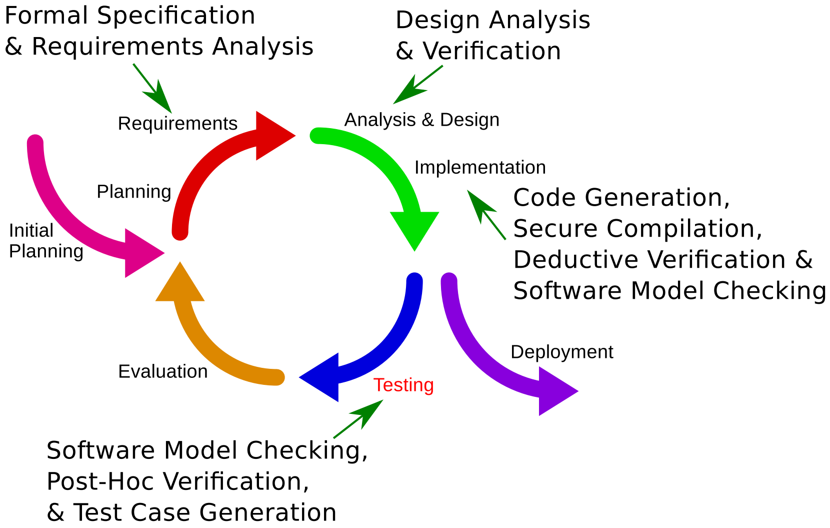
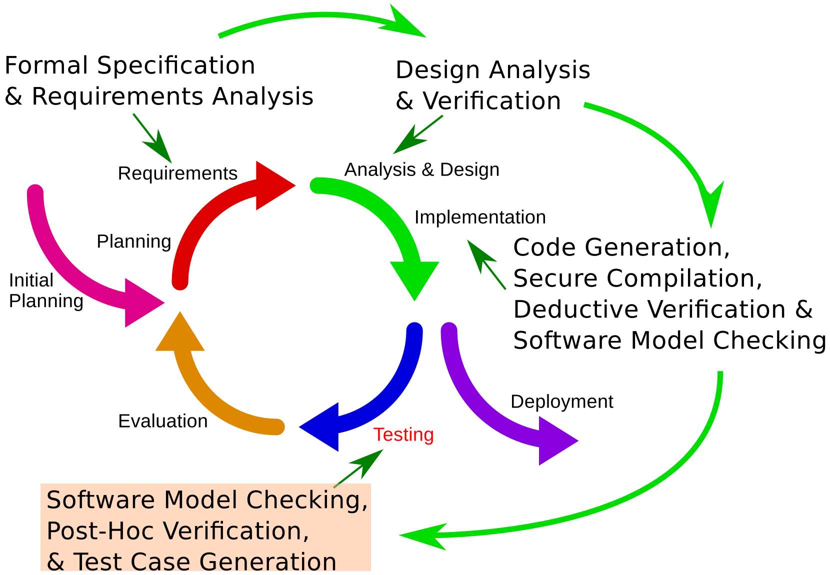


Image: Wikipedia

Iterative and Incremental Development



Iterative and Incremental Development

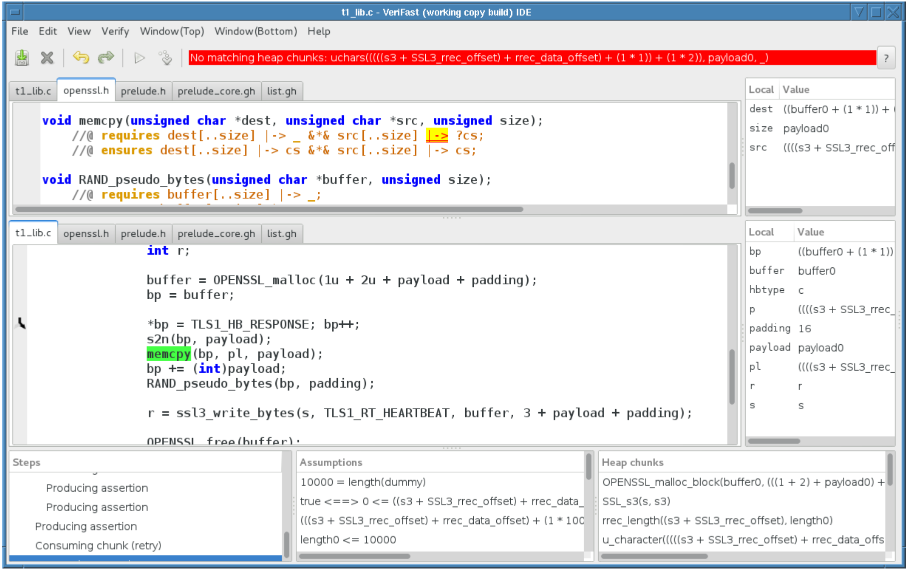


Between Testing and Formal Verification

**"Beware of bugs in the above code;
I have only proved it correct, not tried it."**

– Donald Knuth

VeriFast (imec-DistriNet, [JSP10], [PMP+14])




```
No matching heap chunks: uchars(((((s3 + SSL3_rrec_offset) +
rrec_data_offset) + (1 * 1)) + (1 * 2)), payload0, _)
```

Symbolic Execution (with Microsoft Z3)

```
int foo (bool a, bool b, bool c)
{
    int ret = 0;
    if ((a || b) && c)
    {
        ret = 1;
    }
    return ret;
}
```

Learn more: https://github.com/Z3Prover

Normal "Concrete" Execution: `foo(F, F, F);`
Assignment of concrete inputs, one execution, one output.

```
int foo (bool a, bool b, bool c)
{
    int ret = 0;
    if ((a || b) && c)
    {
        ret = 1;
    }
    return ret;
}
```

Learn more: `https://github.com/Z3Prover`

Symbolic Execution (with Microsoft Z3)

Normal "Concrete" Execution: `foo(F, F, F);`
Assignment of concrete inputs, one execution, one output.

Symbolic Execution: `foo(_, _, _);`
Assign symbolic inputs, use a constraint solver to find concrete inputs that satisfy a specific path.

```
int foo (bool a, bool b, bool c)
{
    int ret = 0;
    if ((a || b) && c)
    {
        ret = 1;
    }
    return ret;
}
```

Learn more: `https://github.com/Z3Prover`

Symbolic Execution (with Microsoft Z3)

Normal "Concrete" Execution: `foo(F, F, F);`
Assignment of concrete inputs, one execution, one output.

Symbolic Execution: `foo(_, _, _);`
Assign symbolic inputs, use a constraint solver to find concrete inputs that satisfy a specific path.

```
(declare-const a Bool)
(declare-const b Bool)
(declare-const c Bool)
```

```
int foo (bool a, bool b, bool c)
{
    int ret = 0;
    if ((a || b) && c)
    {
        ret = 1;
    }
    return ret;
}
```

Learn more: `https://github.com/Z3Prover`

Symbolic Execution (with Microsoft Z3)

Normal "Concrete" Execution: `foo(F, F, F);`
Assignment of concrete inputs, one execution, one output.

Symbolic Execution: `foo(_, _, _);`
Assign symbolic inputs, use a constraint solver to find concrete inputs that satisfy a specific path.

```
(declare-const a Bool)
(declare-const b Bool)
(declare-const c Bool)

(assert (and (or a b) c))
```

```
int foo (bool a, bool b, bool c)
{
    int ret = 0;
    if ((a || b) && c)
    {
        ret = 1;
    }
    return ret;
}
```

Learn more: `https://github.com/Z3Prover`

Symbolic Execution (with Microsoft Z3)

Normal "Concrete" Execution: `foo(F, F, F);`
Assignment of concrete inputs, one execution, one output.

Symbolic Execution: `foo(_, _, _);`
Assign symbolic inputs, use a constraint solver to find concrete inputs that satisfy a specific path.

```
(declare-const a Bool)
(declare-const b Bool)
(declare-const c Bool)

(assert (and (or a b) c))
(check-sat)
-> sat
(get-model)
-> (model
 (define-fun c () Bool true)
 (define-fun a () Bool true))
```

```
int foo (bool a, bool b, bool c)
{
    int ret = 0;
    if ((a || b) && c)
    {
        ret = 1;
    }
    return ret;
}
```

Learn more: `https://github.com/Z3Prover`

Normal "Concrete" Execution: `foo(F, F, F);`
Assignment of concrete inputs, one execution, one output.

Symbolic Execution: `foo(_, _, _);`
Assign symbolic inputs, use a constraint solver to find concrete inputs that satisfy a specific path.

```
(declare-const a Bool)
(declare-const b Bool)
(declare-const c Bool)
(push)
(assert (and (or a b) c))
(check-sat)(get-model)
(pop)
(assert (not
  (and (or a b) c)))
(check-sat)(get-model)
```

```
int foo (bool a, bool b, bool c)
{
    int ret = 0;
    if ((a || b) && c)
    {
        ret = 1;
    }
    return ret;
}
```

Learn more: `https://github.com/Z3Prover`

Symbolic Execution (with Microsoft Z3)

Normal "Concrete" Execution: `foo(F, F, F);`
Assignment of concrete inputs, one execution, one output.

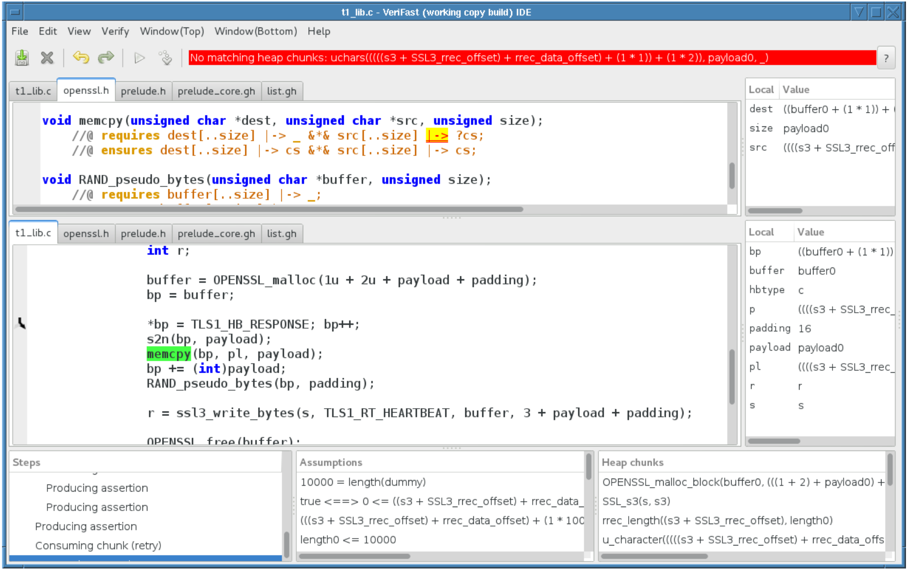
Symbolic Execution: `foo(_, _, _);`
Assign symbolic inputs, use a constraint solver to find concrete inputs that satisfy a specific path.

```
(declare-const a Bool)
(declare-const b Bool)
(declare-const c Bool)
(push)
(assert (and (or a b) c))
(check-sat)(get-model)
(pop)
(assert (not
  (and (or a b) c)))
(check-sat)(get-model)
-> sat
-> (model
 (define-fun c () Bool false))
```

```
int foo (bool a, bool b, bool c)
{
    int ret = 0;
    if ((a || b) && c)
    {
        ret = 1;
    }
    return ret;
}
```

Learn more: `https://github.com/Z3Prover`

VeriFast (imec-DistriNet, [JSP10], [PMP+14])




```
No matching heap chunks: uchars(((((s3 + SSL3_rrec_offset) +
rrec_data_offset) + (1 * 1)) + (1 * 2)), payload0, _)
```

VeriFast

Annotations precisely state pre- and post conditions of functions and loops

VeriFast then checks that these conditions are satisfied for all executions of the program

Somewhat equivalent to putting `assert()` statements before and after every call, then having a very diligent tester exhaustively trying to make each assertion fail

VeriFast

Annotations precisely state pre- and post conditions of functions and loops

VeriFast then checks that these conditions are satisfied for all executions of the program

Somewhat equivalent to putting `assert()` statements before and after every call, then having a very diligent tester exhaustively trying to make each assertion fail

...but VeriFast is automatic, complete and sound. It doesn't forget to check a single assertion and error reports translate to concrete inputs or program paths that trigger error conditions.

...supports concurrency – VeriFast finds the odd synchronisation issue that only pops up if 15 threads are scheduled in a very specific way.

But how good is that?

But how good is that?

Could we have found heartbleed with testing?

But how good is that?

Could we have found heartbleed with testing?

Yes, easily!

```
assert("size of pl >= payload");
memcpy(bp, pl, payload);
```

Plus a test case…

But how good is that?

Could we have found heartbleed with testing?

Yes, easily!

```
assert("size of pl >= payload");
memcpy(bp, pl, payload);
```

Plus a test case…

Why didn't we find heartbleed earlier? With formal methods or testing?

But how good is that?

Could we have found heartbleed with testing?

Yes, easily!

```
assert("size of pl >= payload");
memcpy(bp, pl, payload);
```

Plus a test case…

Why didn't we find heartbleed earlier? With formal methods or testing?

No one thought of it.

But how good is that?

Could we have found heartbleed with testing?

Yes, easily!

```
assert("size of pl >= payload");
memcpy(bp, pl, payload);
```

Plus a test case…

Why didn't we find heartbleed earlier? With formal methods or testing?

No one thought of it.

But: It's easy to "find" a bug in retrospective.

But how good is that?

Could we have found heartbleed with testing?

Yes, easily!

```
assert("size of pl >= payload");
memcpy(bp, pl, payload);
```

Plus a test case. . .

Why didn't we find heartbleed earlier? With formal methods or testing?

No one thought of it.

But: It's easy to "find" a bug in retrospective.

But: You wouldn't know of bugs that got fixed before they could be exploited!

But how good is that?

VeriFast, specifically?

But how good is that?

VeriFast, specifically?

VeriFast finds the bug. Without a tester thinking about a specific test case.

Static verification, no runtime overhead.

VeriFast

But how good is that?

VeriFast, specifically?

VeriFast finds the bug. Without a tester thinking about a specific test case.

Static verification, no runtime overhead.

Writing annotations isn't easy. You may need a lot of annotations – depending on program complexity and verification properties.

But how good is that?

VeriFast, specifically?

VeriFast finds the bug. Without a tester thinking about a specific test case.

Static verification, no runtime overhead.

Writing annotations isn't easy. You may need a lot of annotations – depending on program complexity and verification properties.

You are verifying one part of an application at the level of abstraction provided by C or Java.

Layer-below attacks? Compilation errors?

Buggy or malicious libraries (not behaving to spec)?

Buggy OS? Kernel-level malware?

KLEE (Stanford, [CDE+08])

KLEE is a symbolic virtual machine built on top of LLVM

```
int foo (bool a, bool b, bool c)
{
    int ret = 0;
    if ((a || b) && c)
    {
        ret = 1;
    }
    return ret;
}
```

KLEE (Stanford, [CDE+08])

KLEE is a symbolic virtual machine built on top of LLVM

- No annotations but symbolic test cases
- Support for symbolic arguments, files and streams
- Exploration can be bounded wrt. input sizes, memory and CPU consumption

```
int foo (bool a, bool b, bool c)
{
    int ret = 0;
    if ((a || b) && c)
    {
        ret = 1;
    }
    return ret;
}
```

KLEE (Stanford, [CDE+08])

KLEE is a symbolic virtual machine built on top of LLVM

- No annotations but symbolic test cases
- Support for symbolic arguments, files and streams
- Exploration can be bounded wrt. input sizes, memory and CPU consumption

```
int main(void) {
  bool a, b, c;
  klee_make_symbolic(
    &a, sizeof(a), "a");
  // same for b and c
  return (foo(a, b, c));
}
```

```
int foo (bool a, bool b, bool c)
{
    int ret = 0;
    if ((a || b) && c)
    {
        ret = 1;
    }
    return ret;
}
```

KLEE is a symbolic virtual machine built on top of LLVM

- No annotations but symbolic test cases
- Support for symbolic arguments, files and streams
- Exploration can be bounded wrt. input sizes, memory and CPU consumption

```
int main(void) {
  bool a, b, c;
  klee_make_symbolic(
    &a, sizeof(a), "a");
  // same for b and c
  return (foo(a, b, c));
}
```

```
int foo (bool a, bool b, bool c)
{
    int ret = 0;
    if ((a || b) && c)
    {
        ret = 1;
    }
    return ret;
}
```

- Combines concrete with symbolic execution!
- Bug reports or crashes reported with real program inputs
- Achieve $\geq$ 90% coverage

Pex (Microsoft, [TdH08])

Pex: white-box test generation for .NET

- Pex automatically generates test suites to achieve high code coverage in a short amount of time
- Code must have branches for Pex to be effective
- Combination of concrete and symbolic execution

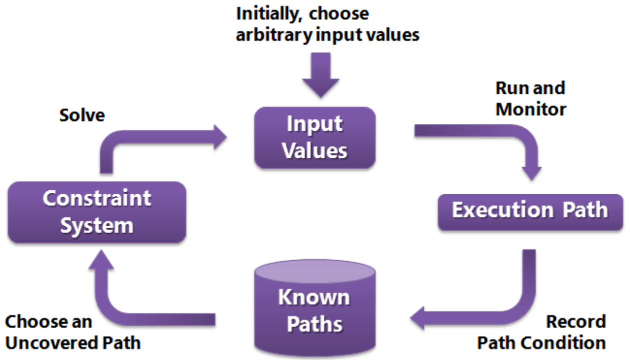


Image & more: `http://www.pexforfun.com/`

Other Tools

Facebook Infer is a static analysis tool - if you give Infer some Java or C/C++/Objective-C code it produces a list of potential bugs.
`http://fbinfer.com/`

CBMC … is a Bounded Model Checker for C and C++ programs. CBMC verifies array bounds (buffer overflows), pointer safety, exceptions and user-specified assertions.
`http://www.cprover.org/cbmc/`

SATABS … is a verification tool for ANSI-C and C++ programs. SATABS transforms a C/C++ program into a Boolean program, which is an abstraction of the original program in order to handle large amounts of code.
`http://www.cprover.org/satabs/`

Conclusions

What is formal software verification?

What (semi-) formal tools and techniques integrate well with testing?

Is formal verification orthogonal to testing? Do we need both?

How do formal verification and testing interact?

When and how can formal verification replace testing, or testing replace formal verification?

Conclusions

What is formal software verification?

What (semi-) formal tools and techniques integrate well with testing?

Is formal verification orthogonal to testing? Do we need both?

How do formal verification and testing interact?

When and how can formal verification replace testing, or testing replace formal verification?

Further Reading:

- Practice and experience with FM: [WLBF09], [TWC01]
- Industrial case studies with VeriFast: [PMP+14]
- Symbolic execution for testing: [CGK+11]

Thank you!

Thank you! Questions?

https://distrinet.cs.kuleuven.be/

References I

C. Cadar, D. Dunbar, D. R. Engler, et al.
Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.
In *OSDI*, vol. 8, pp. 209–224, 2008.

C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser.
Symbolic execution for software testing in practice: Preliminary assessment.
In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pp. 1066–1071, New York, NY, USA, 2011. ACM.

B. Jacobs, J. Smans, and F. Piessens.
VeriFast: Imperative programs as proofs.
In *VSTTE 2010 workshop proceedings*, pp. 63–72, 2010.

P. Philippaerts, J. T. Mühlberg, W. Penninckx, J. Smans, B. Jacobs, and F. Piessens.
Software verification with VeriFast: Industrial case studies.
Science of Computer Programming (SCP), 82:77–97, 2014.

N. Tillmann and J. de Halleux.
Pex – White Box Test Generation for .NET, pp. 134–153.
Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

J. Tretmans, K. Wijbrans, and M. Chaudron.
Software engineering with formal methods: The development of a storm surge barrier control system revisiting seven myths of formal methods.
Formal Methods in System Design, 19(2):195–215, 2001.

References II

J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald.
Formal methods: Practice and experience.
ACM Comput. Surv., 41(4):1–36, 2009.