Techniques for developing and testing secure software components

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

SecAppDev, Leuven, February 2019

Lecturer: Jan Tobias Mühlberg, @jtmuehlberg

Short Bio:

- Research Manager at imec-DistriNet, KU Leuven
  https://distrinet.cs.kuleuven.be/people/muehlber
- Hardware & Software Co-Design for Security
- Embedded Systems Security
- Secure Processors & Trusted Computing
- Automated Software Testing and Formal Verification
- Safety-Critical Systems, Automotive Computing

Automated Detection and Prevention of Vulnerabilities

Frank Piessens: “New trends in system software security”

JT on Tuesday: Developing and testing SW

1. Software security for the bad guys
   Lazy ways of finding and exploiting software vulnerabilities
2. How to build “perfect software”
   Probably there is no such thing; but let’s rule out as many vulnerabilities as possible and affordable

JT on Thursday: Trusted Computing

3. How to protect perfect software at runtime
   . . . because not having vulnerabilities in your code may not be enough
4. Building security into distributed systems

Raoul Strackx: “Foreshadow – from oversight to a tech nightmare”

Software security for the bad guys

You want to “hack” an application!
Stand-alone or client software on a device you control, you have (at least) the compiled binary.

Goals: Hard-coded secrets? Application flags/ enable features? Disable adds? Access or modify application data? Understand remote communication? Find and weaponize a vulnerability?

What’s your approach?

Software security for the bad guys

Option 1: Reversing, search manually

- IDA, debugger, decompiler, experience, luck, brain cycles
- You’ll learn a lot about the program
- You may not find what you’re looking for
- Can be entertaining, can be a big waste of time

Option 2: Fuzzing, automated search

- Clever fuzzing software, little experience, CPU cycles
- You won’t learn that much but you’ll probably get crashes almost for free
- May be easily thwarted by anti-debugging techniques

Option 3: Combine manual reversing and fuzzing

- . . .

Option 1: Reversing, search manually

```
/* stack1.c; https://github.com/gerasdf/InsecureProgramming */

#include <stdio.h>

int main() {
        int cookie;
        char buf[80];

        printf("buf: %08x cookie: %08x\n", &buf, &cookie);
        gets(buf);

        if (cookie == 0x41424344) {
                printf("you win!\n");
        }
}
```

Task: Compile and exploit to get “you win!”. Manually!
src: `stack1.c`; **bin:** `stack1.gcc`

Option 1: Reversing, search manually

Only today: You have source code. It's ok to "instrument" the code a bit to get extra information about your progress. The value of "cookie" could be useful.

```
#include <stdio.h>

int main() {
        int cookie;
        char buf[80];

        printf("&buf: %08x &cookie: %08x\n", &buf, &cookie);
        gets(buf);

        if (cookie == 0x41424344) {
                printf("you win!\n");
        }

        printf("cookie: %08x\n", cookie);
}
```

Option 1: Reversing, search manually

Solution

```
$ perl -e 'print "A"x80 . "DCBA";' | ./stack1.gcc
&buf: 11ff71f0 &cookie: 11ff724c
cookie: 00000000
```

Hu?! 0x00000000?!

```
$ perl -e 'print "A"x100 . "DCBA";' | ./stack1.gcc
&buf: 6f65f350 &cookie: 6f65f3ac
cookie: 41414141
Segmentation fault
```

Ah! **But why?** Crash after last `printf()`? `&buf` and `&cookie` changed?

Option 1: Reversing, search manually

Solution (cont'd)

```
$ perl -e 'print "A"x90 . "DCBA";' | ./stack1.gcc
&buf: 10f732d0 &cookie: 10f7332c
cookie: 00004142
```

Ok, done.

```
$ perl -e 'print "A"x92 . "DCBA";' | ./stack1.gcc
&buf: 816fb9c0 &cookie: 816fba1c
you win!
cookie: 41424344
```

Now let's automate this: fuzzing the input with AFL.

Option 2: Fuzzing, automated search

Can we crash it automatically with AFL [Zal10]?

Compile the Target

```
$ afl-2.52b/afl-gcc -std=c99 -ggdb stack1.c -o stack1.afl
$ ls -l
-rwxr-xr-x  1 muehlber muehlber 16888 Nov  3 10:24 stack1.afl
-rwxr-xr-x  1 muehlber muehlber 11232 Nov  1 16:11 stack1.gcc
```

`afl-gcc` instruments the target code to measure coverage, observe conditionals, and to improve detection of vulnerabilities.

Option 2: Fuzzing, automated search

Running the Fuzzer

```
# fuzzing programs that accept input on std-in
$ afl-2.52b/afl-fuzz -i testcase_dir -o findings_dir \
  /path/to/program [...params...]

# fuzzing programs that accept file name parameters
$ afl-2.52b/afl-fuzz -i testcase_dir -o findings_dir \
  /path/to/program [...params...] @@
```

You will often have to write a “test harness” to transform an input file into the right structured input (e.g. simulate a network packet, a sequence of packets, . . .) for your target.

Option 2: Fuzzing, automated search

Fuzzing `stack1.afl`

```
$ mkdir -p in
$ mkdir -p out
$ echo "test string" >in/seed001
$ AFL_SKIP_CPUFREQ=1 \
  afl-2.52b/afl-fuzz -i in -o out -- ./stack1.afl
```

Interrupt with `Ctrl+C`. You decide when.

Option 2: Fuzzing, automated search

```
                 american fuzzy lop 2.52b (stack1.afl)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 0 min, 2 sec         │  cycles done : 6       │
│   last new path : none yet (odd, check syntax!)       │  total paths : 1       │
│ last uniq crash : 0 days, 0 hrs, 0 min, 1 sec         │ uniq crashes : 1       │
│  last uniq hang : none seen yet                       │   uniq hangs : 0       │
├─ cycle progress ────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 0 (0.00%)      │    map density : 0.00% / 0.00%          │
│ paths timed out : 0 (0.00%)      │ count coverage : 1.00 bits/tuple        │
├─ stage progress ────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc              │ favored paths : 1 (100.00%)             │
│ stage execs : 160/256 (62.50%)   │  new edges on : 1 (100.00%)             │
│ total execs : 2951               │ total crashes : 12 (1 unique)           │
│  exec speed : 1032/sec           │  total tmouts : 0 (0 unique)            │
├─ fuzzing strategy yields ───────┴───────────────┬─ path geometry ────────┤
│   bit flips : 0/32, 0/31, 0/29                    │    levels : 1          │
│  byte flips : 0/4, 0/3, 0/1                       │   pending : 0          │
│ arithmetics : 0/224, 0/0, 0/0                     │  pend fav : 0          │
│  known ints : 0/24, 0/84, 0/44                    │ own finds : 0          │
│  dictionary : 0/0, 0/0, 0/0                       │  imported : n/a        │
│       havoc : 1/2304, 0/0                         │ stability : 100.00%    │
│        trim : 66.67%/2, 0.00%                     ├────────────────────────┘
^C────────────────────────────────────────────────────┘          [cpu000:156%]
```

Option 2: Fuzzing, automated search

Inspecting the results

```
$ ls out
crashes  fuzz_bitmap  fuzzer_stats  hangs  plot_data  queue
$ ls out/crashes/
id:000000,sig:11,src:000000,op:havoc,rep:128  README.txt
```

...and replay them!

```
$ ./stack1.afl < out/crashes/id\:000000*
&buf: 75586e80 &cookie: 75586e7c
Segmentation fault
$ ./stack1.gcc < out/crashes/id\:000000*
&buf: 59f43230 &cookie: 59f4328c
cookie: ff05eeee
Segmentation fault
```

Option 2: Fuzzing, automated search

But what about “You win?”

- AFL explored only one program path!
- Is the `true` branch of `if (cookie == 0x41424344)` even reachable?

```
$ perl -e 'print "A"x92 . "DCBA";' | ./stack1.afl
buf: dea0ed10 cookie: dea0ed0c
Segmentation fault
```

- Instrumentation make fuzzing fast but change execution semantics!
- Still: **You found the vulnerability.**
- Automatic exploits require different tools: **QEMU AFL**

Option 2: Fuzzing, automated search

- Can we crash it: AFL [Zal10]
- Find an input that reproducibly leads to SIGSEGV, SIGILL, SIGABRT
- This a library function, we can build our own "client" as a test harness:

```
int main(int c, char* v[]) {
  struct rrec r; struct SSL3 s3;
  struct SSL  s;
  if (c >= 2)
    read_in(v[1], &r);
  s.s3 = &s3; s3.rrec = r;
  return tls1_process_heartbeat(&s);
}
```

- Provide a seed test case "_ _ _ _"
- Compile with instrumentation, run in AFL

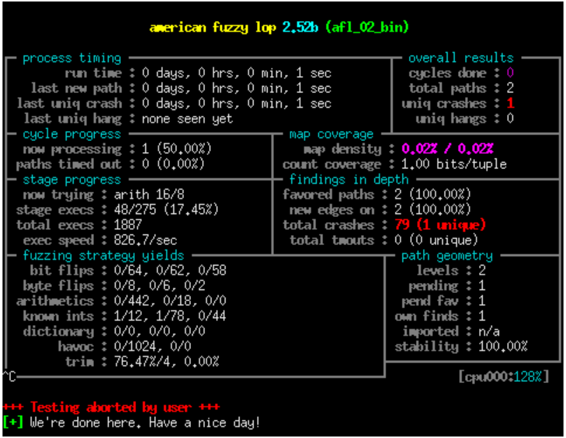
```
int tls1_process_heartbeat (SSL *s) {
  unsigned char *p = s->s3->rrec.data;
  // ...
  hbtype = *p; p++;
  n2s(p, payload); pl = p;
  if (hbtype == TLS1_HB_REQUEST) {
    unsigned char *buffer, *bp; int r;
    buffer = OPENSSL_malloc(1 + 2 +
    payload + padding);
    bp = buffer;

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

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

Option 2: Fuzzing, automated search

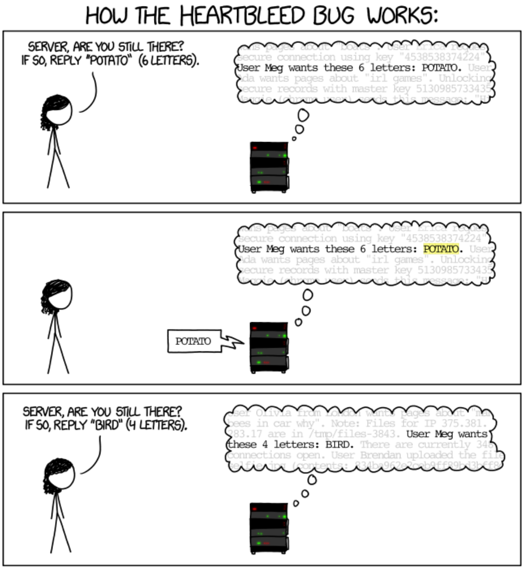
- Test case for a crash within one second: 0x20 0x64 0x20 0x20
- Severity as a vulnerability depends on executing context and skill of the attacker

But what happened?

1. Take next test case from queue
2. Trim the test case to the smallest size that does not alter testee’s behavior,
3. Repeatedly mutate the test case,
4. If any of the generated mutations results in a new state transition, add it to the queue,
5. Go to 1.



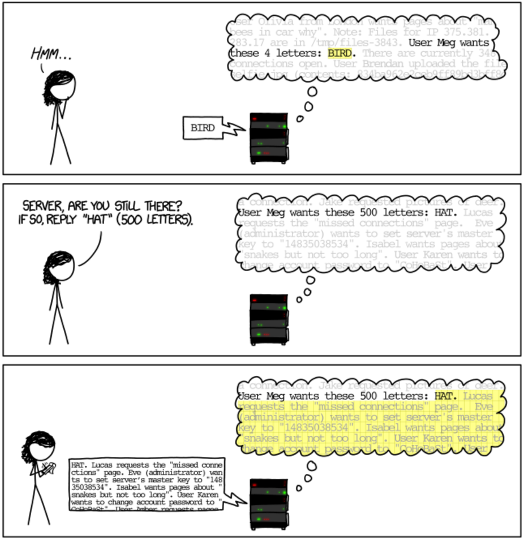
Option 2: Fuzzing, automated search



Source: `https://xkcd.com/1354/`

```
int tls1_process_heartbeat (SSL *s) {
  unsigned char *p = s->s3->rrec.data;
  // ...
  hbtype = *p; p++;
  n2s(p, payload); pl = p;
  if (hbtype == TLS1_HB_REQUEST) {
    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);
    // ... } ... }
```

Option 2: Fuzzing, automated search




```
int tls1_process_heartbeat (SSL *s) {
  unsigned char *p = s->s3->rrec.data;
  // ...
  hbtype = *p; p++;
  n2s(p, payload); pl = p;
  if (hbtype == TLS1_HB_REQUEST) {
    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);
    // ... } ... }
```

Source: `https://xkcd.com/1354/`

But . . .

But it's a known vulnerability, extracted, simplified, . . .
Yes, that's why it took only 1s.

But the input was really simple!
AFL pulls compressed multimedia files out of thin air. Also, there are specialised tools for network traffic, HW interactions, video streams. Problem: Crypto.

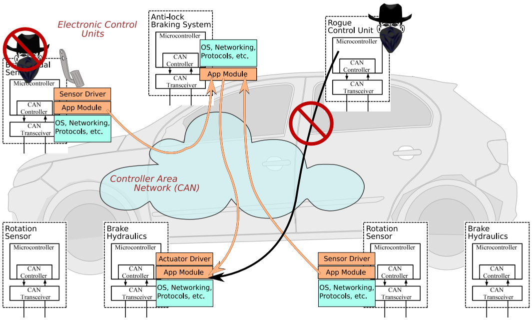
But you instrumented source code! We ship only binaries!
QEMU mode! What about your libraries?

But we also obfuscate them! And there's an obscure interpreter in there!
Does it still execute? Let's wait it out. Problem: Opaque predicate.

But we have anti-debugging! And the red stuff above!
Fuzzing coverage will reveal dead ends, which can be resolved manually.

Any vulnerability can be found. Understand your system, your assets, your attacker → Threat Modelling

My Personal Fuzzing Surprise



“VulCAN: Efficient Component Authentication and Software Isolation for Automotive Control Networks”, Van Bulck et al., ACSAC 2017. [VBMP17]

Software security for application developers

How can we defend applications against fuzzing?
How can we defend against people with reverse engineering skills?

Fuzz harder?

Fuzz more cleverly?

Hire a bad guy and ask him to do good stuff?

Testing?

Buy an insurance?

Penetration testing?

Formal verification?

Under what attacker model can we say that a thoroughly tested or formally verified application is secure?

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, . . .

```
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?

- 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 (Delta Works, SDI, Space)?
- How do we know that we haven't missed critical interactions? Concurrency?
- Who writes all these tests?
- What about security properties?

```
int tls1_process_heartbeat (SSL *s) {
  unsigned char *p = s->s3->rrec.data;
  // ...
  hbtype = *p; p++;
  n2s(p, payload); pl = p;
  if (hbtype == TLS1_HB_REQUEST) {
    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?

Life-critical, Safety-critical, Ultra-reliable

- 10^{-9} probability of failure for a 1 hour mission
  - $\rightarrow$ life-test for $> 114,000$ years (**safety!**)

Not Just Space Tech!

Image: NASA, STS-132; **FM @ NASA:** `https://shemesh.larc.nasa.gov/fm/fm-why.html`


COMMUNICATIONS
HOME
CURRENT ISSUE
NEWS
BLOGS
PRACT
(Vol. 53, No. 2) / Software Model Checking Takes
PRACTICE
Software Mod
akes Off
By Steven P. Miller, Michael W. Wha
Communications of the ACM, Vol.
10.1145/1646353.1646372
Comments
VIEW AS:
SHARE:
Although formal met
have been
in the
development of safety-
tems for
years, they have not achie
rial use
in software or systems engine
important trends are making the indus
f formal
methods practical. The first
model-based development fo
systems. Tools such as MATL
Technologies SCADE Suite
in the design of avionics and
graphical models produced
formal, or nearly formal, spe
amenable to formal analysis.
cond is the growing power of formal verification tools, particularly m
ckers. For
any classes of models they provide a "push-button" means of determining
del meets its
quirements. Since these tools examine all possible combinations of inputs a
tate, they are
uch more likely to find design errors than testing.
in 2006, A
one trillion ob
cts. Less than a year later it had grown to two trillion objects and was regula
handling 1.1 million requests per second.
COMPUTER SECUR
Hack
Computer s
theorems. T
A 12
1.474
Published on Fe
CONTRIBUTE
How
By Chris New
Communicat
Comments
VIEW AS:
Home
Mon, 23
the libra
It is nam
it is now
De Gou
processo
web

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

“We’re building self-driving cars and planning Mars missions – but we haven’t figured out how to make sure people’s vacuum cleaners don’t join botnets.”

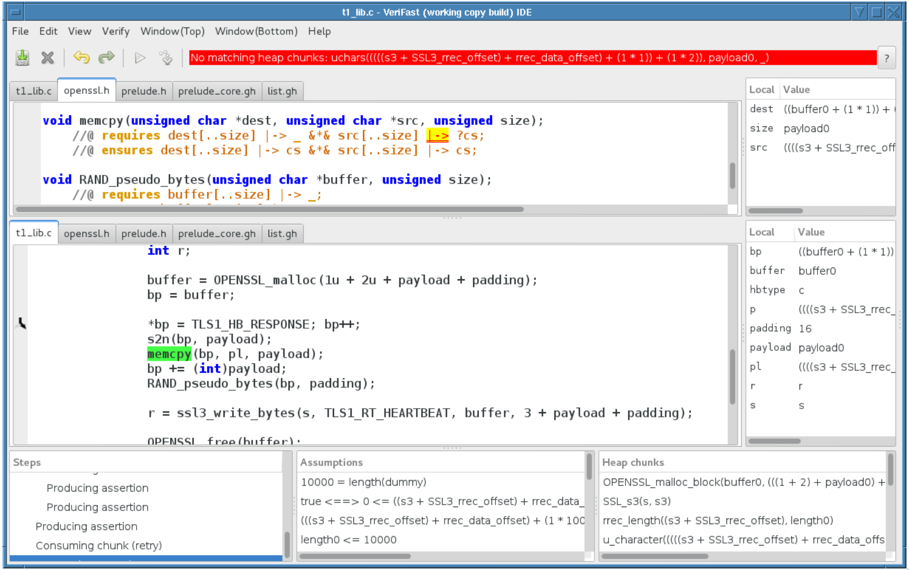
– Someone at JSConfAU16

Source: `https://twitter.com/MelissaKaulfuss/status/804209991510937600?s=09`

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	Expensive

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



Normal Execution vs. Symbolic Execution

Normal "Concrete" Execution: `foo(F, F, F);`

Assignment of concrete inputs, one execution, one output (unit tests, etc.)

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

Symbolic Execution (with Microsoft Z3)

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`

Symbolic Execution (with Microsoft Z3)

Symbolic Execution: `foo(_, _, _);`

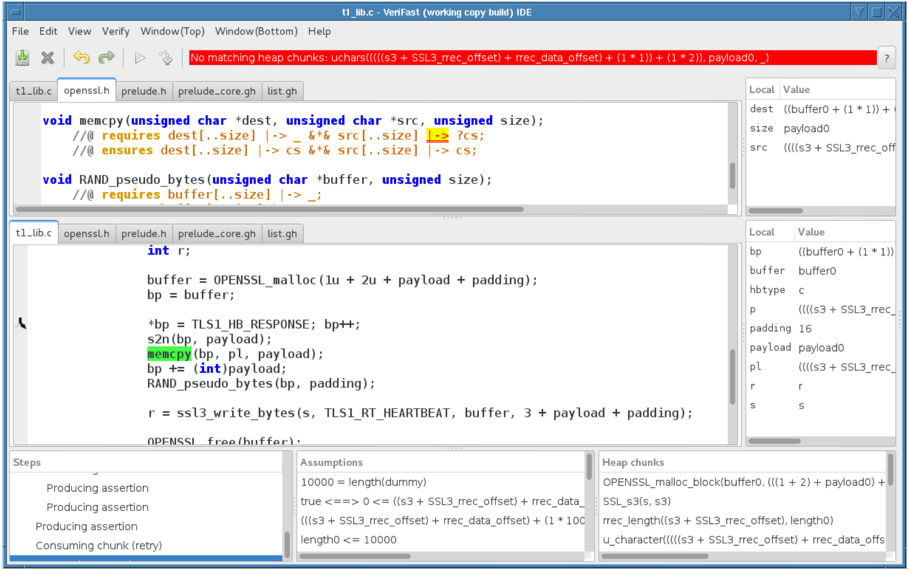
Assign symbolic inputs, use a "constraint solver" to find concrete inputs that satisfy a specific path.

```
(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])



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

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!

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

VeriFast, specifically?

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

VeriFast is automatic, complete and sound, and supports concurrency: Pre- and post conditions must be satisfied for all executions

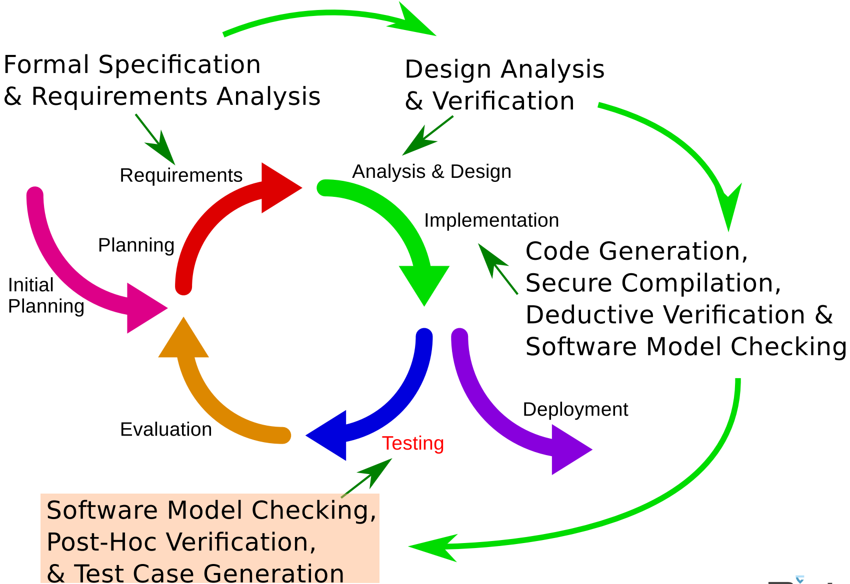
Static verification, no runtime overhead.

Writing pre- and post conditions 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?

Between Testing and Formal Verification



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;
}
```

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

Symbolic Execution in Attacks

Some techniques work on binary programs, in the absence of source code.
AFL [Zal10], SAGE [GLM08], SOCA [ML10], etc.

Automated Crash Generation
...search for paths where a well-chosen input leads to undefined behaviour or unhandled exceptions.
You have seen this for AFL.

Automated Exploit Generation
...as above, but find exploitable behaviour and derive a “crazy machine” to execute code:

- Patch-based exploit generation [BPSZ08]
- Crash analysis and exploit generation [HHH+14]
- End-to-end solutions to generate zero-days [ACR+14]

Other Tools

MS PEX …automatically generates test suites to achieve high code coverage in .NET in a short amount of time [TdH08].

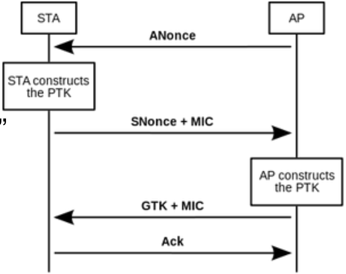
⊢ Facebook Infer is a static analysis tool - if you give Infer some Java or C/C++/Objective-C code it produces a list of potential bugs.
`http://fbinfer.com/`

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/`

Key Reinstallation Attacks

Breaking WPA2 by forcing nonce reuse: "The attack works against all modern protected Wi-Fi networks. [. . .] if your device supports Wi-Fi, it is most likely affected."



Analysis

- Problem in IEEE 802.11i (2004)
- Formal security properties by He et al. [HSD+05]
- Crypto in Wi-Fi are highly secure (iff secure nonces)

What went wrong?

- Two "unit proofs", no "integration proof"

→ Formal correctness of protocols in integrated scenarios!

→ Correct implementations (verified **and** tested)

- That's expensive! As compared to what?

Discovered by **Mathy Vanhoef** at imec-DistriNet, `https://www.krackattacks.com/`, paper at CCS (November 2017)
Discussion of verification efforts by **Matthew Green**, `https://blog.cryptographyengineering.com/`

Preventing Vulnerabilities Through Testing and Verification

Modern (embedded) software systems are huge!

- Interactions with safety-critical components not well defined
- There are bugs in established standards and well-tested code
- Formal analysis and verification reduces the chance for bugs to slip through
- **Don't forget to isolate critical code!**

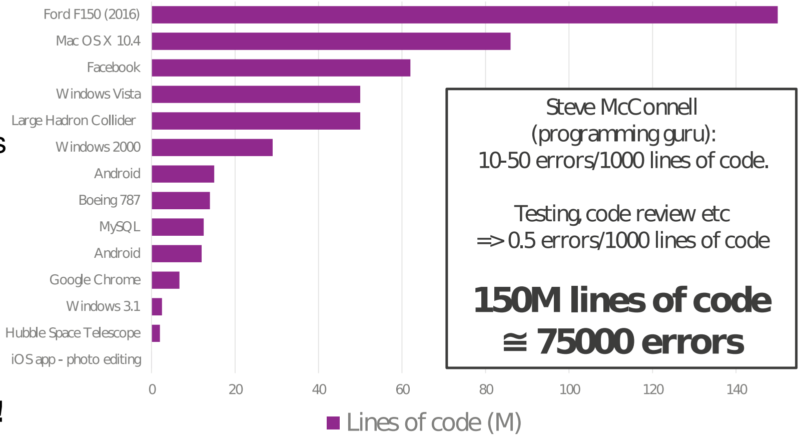


Image: Thomas Kallstenius @ imec ITF, May 2017

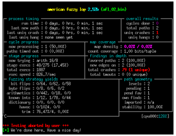
Summary

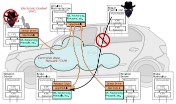
Fuzzing, Testing & Formal Verification

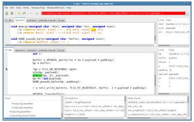
1. There are automated techniques to find vulnerabilities and to generate exploits
2. Securing application code requires dedicated testing and verification
3. Know your system, be selective
4. Correct code still needs protection against layer-below attacks!

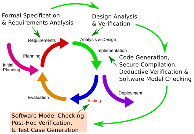
My next session: Trusted Computing & Sancus

1. Strong application isolation and attestation
2. Requires correct hardware and software









Thank you!

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

– Donald Knuth

Thank you! Questions?

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

References I

T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley.
Automatic exploit generation.
Commun. ACM, 57(2):74–84, 2014.

D. Brumley, P. Poosankam, D. Song, and J. Zheng.
Automatic patch-based exploit generation is possible: Techniques and implications.
In *2008 IEEE Symposium on Security and Privacy (S&P 2008)*, pp. 143–157, 2008.

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.

P. Godefroid, M. Y. Levin, and D. Molnar.
Automated whitebox fuzz testing.
In *NDSS '08*. Internet Society (ISOC), 2008.

S. K. Huang, M. H. Huang, P. Y. Huang, H. L. Lu, and C. W. Lai.
Software crash analysis for automatic exploit generation on binary programs.
IEEE Transactions on Reliability, 63(1):270–289, 2014.

C. He, M. Sundararajan, A. Datta, A. Derek, and J. C. Mitchell.
A modular correctness proof of ieee 802.11i and tls.
In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pp. 2–15, New York, NY, USA, 2005. ACM.

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

References II

J. T. Mühlberg and G. Lüttgen.
Symbolic object code analysis.
In *SPIN '10*, vol. 6349 of *LNCS*, pp. 4–21, Heidelberg, 2010. Springer.

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. Van Bulck, J. T. Mühlberg, and F. Piessens.
VulCAN: Efficient component authentication and software isolation for automotive control networks.
In *ACSAC '17*, pp. 225–237. ACM, 2017.

M. Zalewski.
American Fuzzy Lop: A security-oriented fuzzer, 2010.
http://lcamtuf.coredump.cx/afl/.