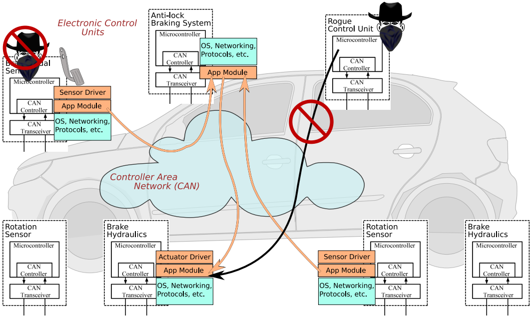
Techniques for developing and integrating secure software components

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

QA&Test, Bilbao, October 2017

My usual work: Trusted Computing for Embedded Control Systems



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

Developing and integrating secure software components

Today:

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
3. How to protect perfect software at runtime
   . . . because having no vulnerabilities in your code may not be enough

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 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:

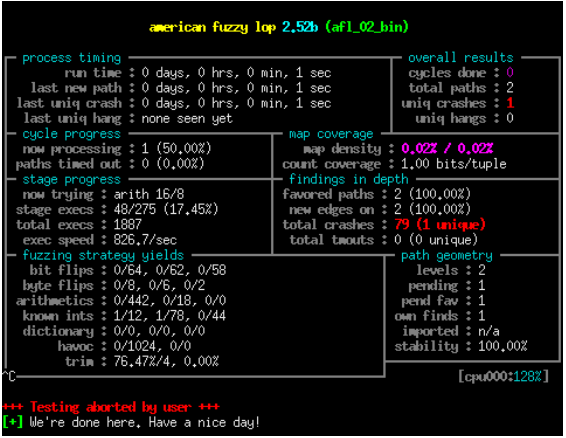
```
int main(int c, char* v[]) {
  struct rrec r; struct SSL3 s3;
  struct SSL  s;
  if (c >= 2)
    read_in(v[1], &r);
  s.s3 = &s3; s3.rrec = r;
  return tls1_process_heartbeat(&s);
}
```

- 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

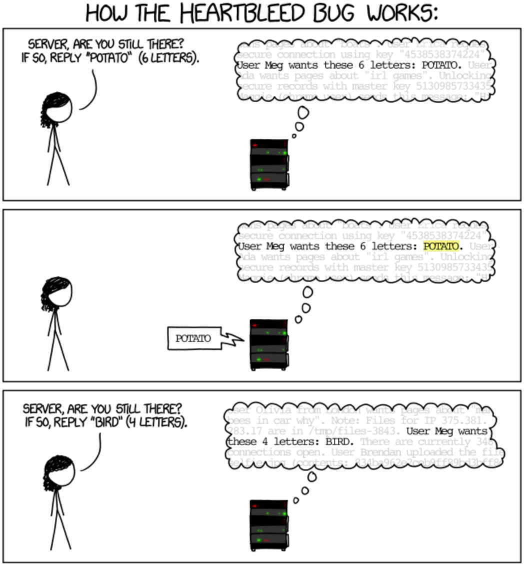


```
                 american fuzzy lop 2.52b (afl_02_bin)

┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 0 min, 1 sec         │  cycles done : 0     │
│   last new path : 0 days, 0 hrs, 0 min, 1 sec         │  total paths : 2     │
│ 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 : 1 (50.00%)       │    map density : 0.02% / 0.02%        │
│ paths timed out : 0 (0.00%)        │ count coverage : 1.00 bits/tuple      │
├─ stage progress ──────────────────┼─ findings in depth ────────────────────┤
│  now trying : arith 16/8           │ favored paths : 2 (100.00%)            │
│ stage execs : 48/275 (17.45%)      │  new edges on : 2 (100.00%)            │
│ total execs : 1887                 │ total crashes : 79 (1 unique)          │
│  exec speed : 826.7/sec            │  total tmouts : 0 (0 unique)           │
├─ fuzzing strategy yields ─────────┴──────────────┬─ path geometry ────────┤
│   bit flips : 0/64, 0/62, 0/58                     │    levels : 2          │
│  byte flips : 0/8, 0/6, 0/2                        │   pending : 1          │
│ arithmetics : 0/442, 0/18, 0/0                     │  pend fav : 1          │
│  known ints : 1/12, 1/78, 0/44                     │ own finds : 1          │
│  dictionary : 0/0, 0/0, 0/0                        │  imported : n/a        │
│       havoc : 0/1024, 0/0                          │ stability : 100.00%    │
│        trim : 76.47%/4, 0.00%                      ├────────────────────────┘
^C────────────────────────────────────────────────────┘          [cpu000:128%]

+++ Testing aborted by user +++
[+] We're done here. Have a nice day!
```

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.

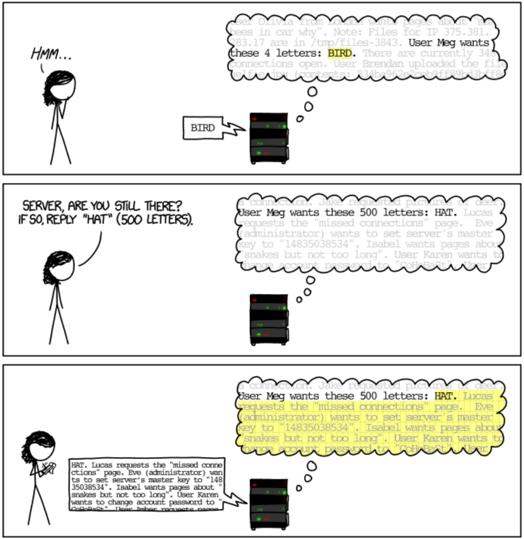
Option 2: Fuzzing, automated search



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



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

```c
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);
    // ... } ... }
```

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

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`

PRACTICE

Software Mode… heckin… akes Off

By Steven P. Miller, Michael W. Wha…
Communications of the ACM, Vol. …
10.1145/1646353.1646372
Comments

Although formal met… have been … in the development of safety- … ecur… tems for years, they have not achiev… spre… ind… rial use in software or systems engine… . Ho… ever… o important trends are making the indus… al u… of formal methods practical. The first … the gro… g ac… tance of model-based development fo… dded systems. Tools such as MATL… erel Technologies SCADE Suite[2] a… ead use in the design of avionics and … The graphical models produced b… a formal, or nearly formal, spe… amenable to formal analysis.

…e… cond is the growing power of formal verification tools, particularly m… h… 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.[3] Less than a year later it had grown to two trillion objects and was regula… handling 1.1 million requests per second.[4]

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.”**

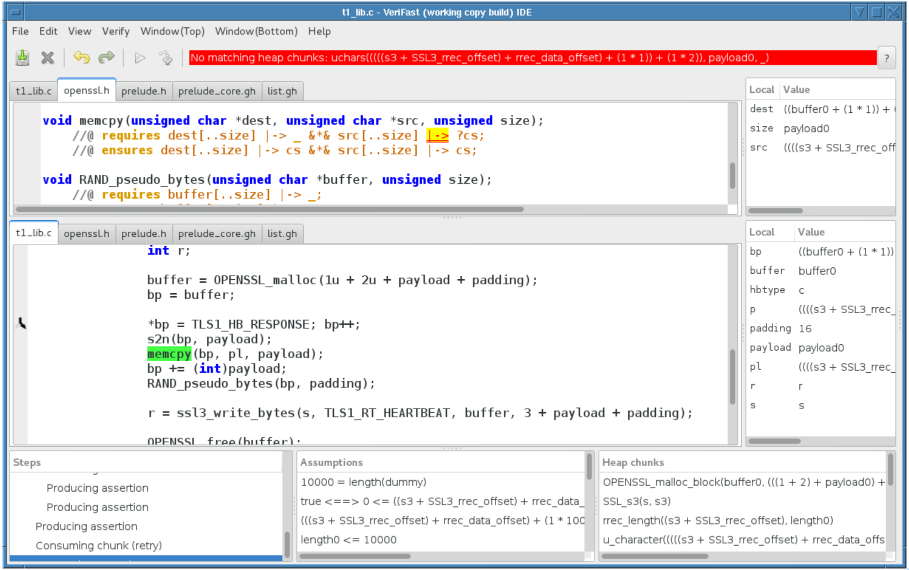
– Someone at JSConfAU16

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])



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

```
void memcpy(unsigned char *dest, unsigned char *src, unsigned size);
    //@ requires dest[..size] |-> _ &*& src[..size] |-> ?cs;
    //@ ensures dest[..size] |-> cs &*& src[..size] |-> cs;

void RAND_pseudo_bytes(unsigned char *buffer, unsigned size);
    //@ requires buffer[..size] |-> _;
```

Local	Value
dest	((buffer0 + (1 * 1)) + (
size	payload0
src	((((s3 + SSL3_rrec_off

```
                int r;

                buffer = OPENSSL_malloc(1u + 2u + payload + padding);
                bp = buffer;

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

                r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);

                OPENSSL_free(buffer);
```

Local	Value
bp	((buffer0 + (1 * 1))
buffer	buffer0
hbtype	c
p	((((s3 + SSL3_rrec_
padding	16
payload	payload0
pl	((((s3 + SSL3_rrec_
r	r
s	s

Steps

- Producing assertion
- Producing assertion
- Producing assertion
- Consuming chunk (retry)

Assumptions

10000 = length(dummy)
true <==> 0 <= ((s3 + SSL3_rrec_offset) + rrec_data_
(((s3 + SSL3_rrec_offset) + rrec_data_offset) + (1 * 100
length0 <= 10000

Heap chunks

OPENSSL_malloc_block(buffer0, (((1 + 2) + payload0) +
SSL_s3(s, s3)
rrec_length((s3 + SSL3_rrec_offset), length0)
u_character(((((s3 + SSL3_rrec_offset) + rrec_data_offs

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(_, _, _);`

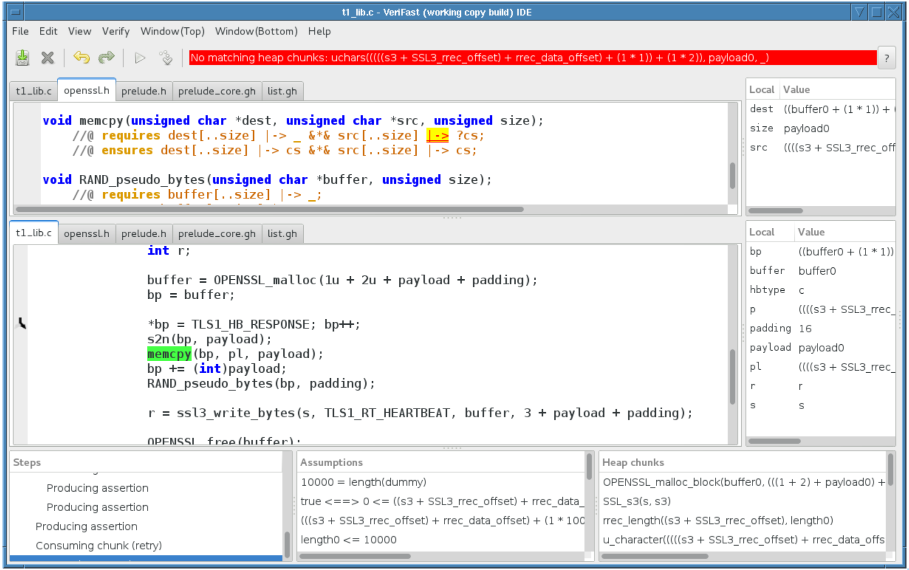
Assign symbolic inputs, use a “constraint solver” to find concrete inputs that satisfy a specific path.

```
(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])



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

```
void memcpy(unsigned char *dest, unsigned char *src, unsigned size);
    //@ requires dest[..size] |-> _ &*& src[..size] |-> ?cs;
    //@ ensures dest[..size] |-> cs &*& src[..size] |-> cs;

void RAND_pseudo_bytes(unsigned char *buffer, unsigned size);
    //@ requires buffer[..size] |-> _;
```

Local	Value
dest	((buffer0 + (1 * 1)) + (
size	payload0
src	(((s3 + SSL3_rrec_off

```
            int r;

            buffer = OPENSSL_malloc(1u + 2u + payload + padding);
            bp = buffer;

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

            r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);

            OPENSSL_free(buffer);
```

Local	Value
bp	((buffer0 + (1 * 1))
buffer	buffer0
hbtype	c
p	(((s3 + SSL3_rrec_
padding	16
payload	payload0
pl	(((s3 + SSL3_rrec_
r	r
s	s

Steps

- Producing assertion
- Producing assertion
- Producing assertion
- Consuming chunk (retry)

Assumptions

- 10000 = length(dummy)
- true <==> 0 <= ((s3 + SSL3_rrec_offset) + rrec_data_
- (((s3 + SSL3_rrec_offset) + rrec_data_offset) + (1 * 100
- length0 <= 10000

Heap chunks

- OPENSSL_malloc_block(buffer0, (((1 + 2) + payload0) +
- SSL_s3(s, s3)
- rrec_length((s3 + SSL3_rrec_offset), length0)
- u_character(((((s3 + SSL3_rrec_offset) + rrec_data_offs

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

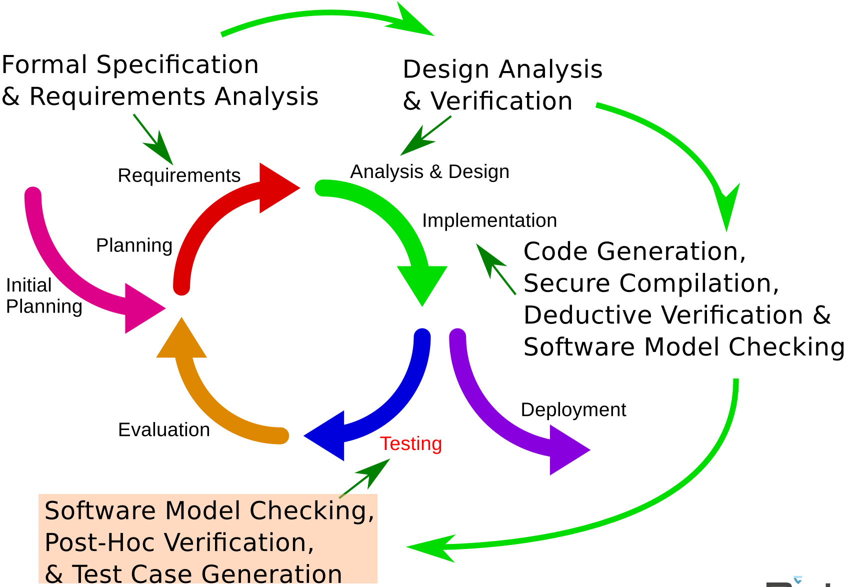
Static verification, no runtime overhead.

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



Formal Specification
& Requirements Analysis

Design Analysis
& Verification

Code Generation,
Secure Compilation,
Deductive Verification &
Software Model Checking

Software Model Checking,
Post-Hoc Verification,
& Test Case Generation

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].

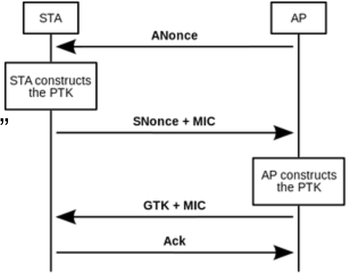
⊢ Facebook Infer is a static analysis tool - if you give Infer some Java or C/C++/Objective-C code it produces a list of potential bugs.
`http://fbinfer.com/`

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)

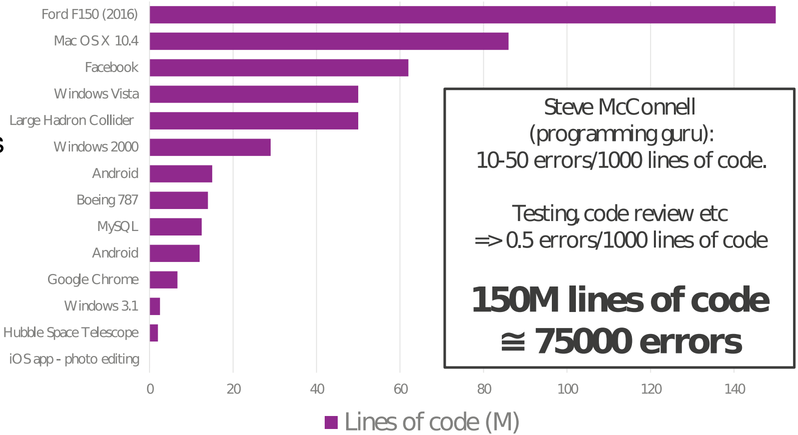
- That's expensive! As compared to what?

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!**



Steve McConnell
(programming guru):
10-50 errors/1000 lines of code.

Testing, code review etc
=> 0.5 errors/1000 lines of code

150M lines of code
≅ 75000 errors

Image: Thomas Kallstenius @ imec ITF, May 2017

Trusted Computing

According to the *Trusted Computing Group*

Protect computing infrastructure at end points;
Hardware extensions to enforce specific behaviour and to provide cryptographic capabilities, protecting against unauthorised change and attacks

- **Endorsement Key**, EK Certificate, Platform Certificate: Unique private key that never leaves the hardware, authenticate device identity
- **Memory curtaining:** provide isolation of sensitive areas of memory
- **Sealed storage:** Bind data to specific device or software
- **Remote attestation:** authenticate hardware and software configuration to a remote host
- **Trusted third party** as an intermediary to provide (ano|pseudo)nymity

In practice: different architectures, subset of the above features, additions such as “enclaved” execution, memory encryption or secure I/O capabilities

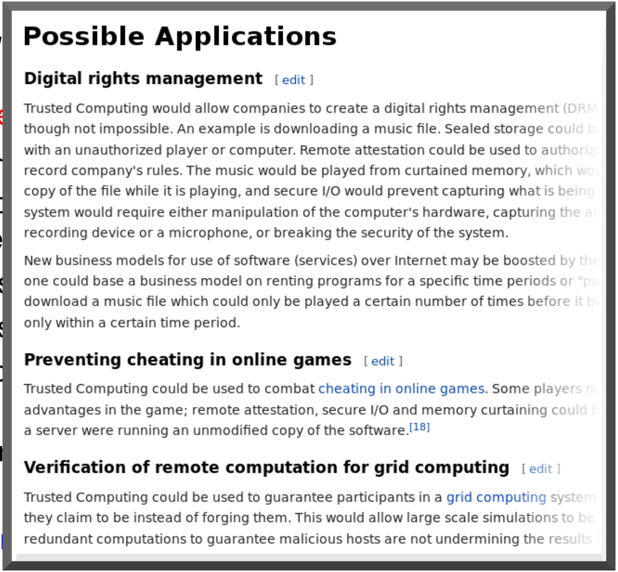
Source: `https://en.wikipedia.org/wiki/Trusted_Computing`

Trusted Computing

According to the *Trusted Compu*
Protect computing infrastructure at
Hardware extensions to enforce spe
capabilities, protecting against unau

- **Endorsement Key**, EK Certific that never leaves the hardware
- **Memory curtaining:** provide is
- **Sealed storage:** Bind data to s
- **Remote attestation:** authentic remote host
- **Trusted third party** as an inter

In practice: different architectures,
as “enclaved” execution, memory e



Possible Applications

Digital rights management [edit]

Trusted Computing would allow companies to create a digital rights management (DRM
though not impossible. An example is downloading a music file. Sealed storage could b
with an unauthorized player or computer. Remote attestation could be used to authorize
record company's rules. The music would be played from curtained memory, which wou
copy of the file while it is playing, and secure I/O would prevent capturing what is being
system would require either manipulation of the computer's hardware, capturing the an
recording device or a microphone, or breaking the security of the system.

New business models for use of software (services) over Internet may be boosted by the
one could base a business model on renting programs for a specific time periods or "pa
download a music file which could only be played a certain number of times before it be
only within a certain time period.

Preventing cheating in online games [edit]

Trusted Computing could be used to combat cheating in online games. Some players m
advantages in the game; remote attestation, secure I/O and memory curtaining could b
a server were running an unmodified copy of the software.[18]

Verification of remote computation for grid computing [edit]

Trusted Computing could be used to guarantee participants in a grid computing system
they claim to be instead of forging them. This would allow large scale simulations to be
redundant computations to guarantee malicious hosts are not undermining the results t

Source: `https://en.wikipedia.org/wiki/Trusted_Computing`

Trusted Computing

According to *Richard Stallman*

Treacherous Computing: “The technical idea underlying treacherous computing is that the computer includes a digital encryption and signature device, and the keys are kept secret from you. Proprietary programs will use this device to control which other programs you can run, which documents or data you can access, and what programs you can pass them to. These programs will continually download new authorisation rules through the Internet, and impose those rules automatically on your work.”

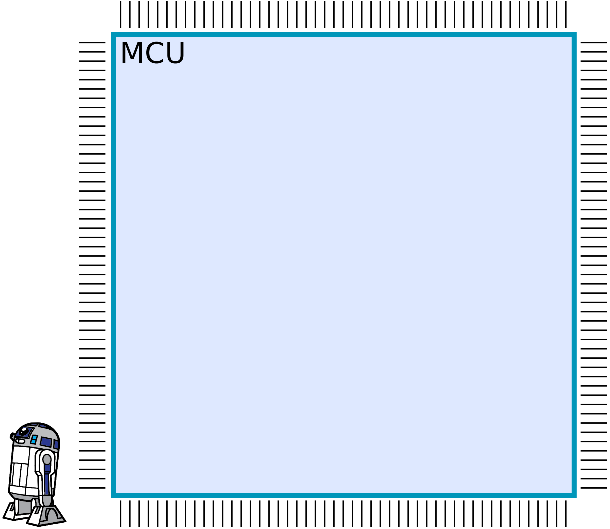
In the light of recent incidents. . .

- **Buggy software:** think of OpenSSL’s Heartbleed in an enclave
- **Side channels:** timing, caching, speculative execution, etc.
- **Buggy system:** CPUs, peripherals, firmware (Broadpwn, Intel ME, Meltdown)
- **Malicious intent:** Backdoors, ransomware, etc.

Source: `https://www.gnu.org/philosophy/can-you-trust.html`

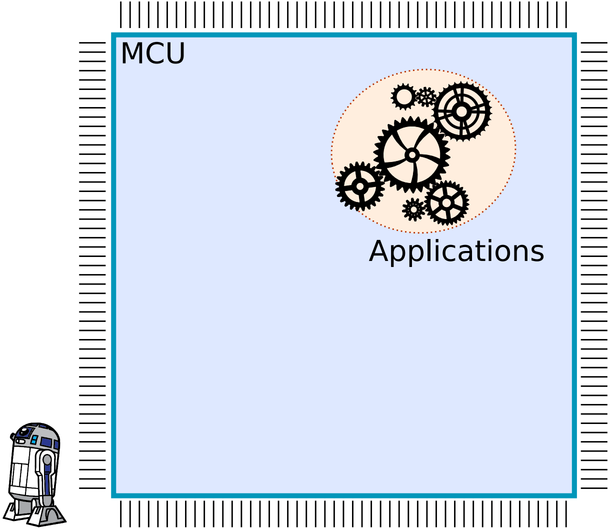
Isolation and Attestation on Light-Weight MCUs

Many microcontrollers feature little security functionality



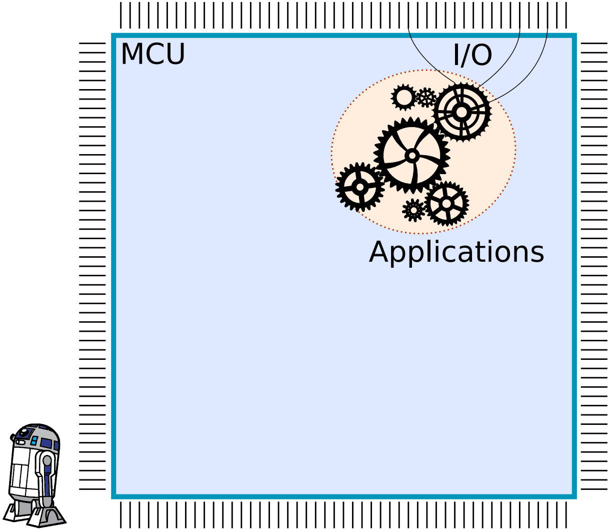
Isolation and Attestation on Light-Weight MCUs

Many microcontrollers feature little security functionality



Isolation and Attestation on Light-Weight MCUs

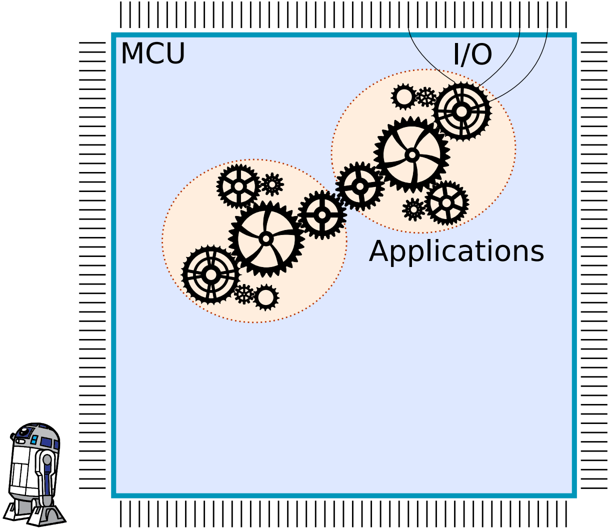
Many microcontrollers feature little security functionality



Isolation and Attestation on Light-Weight MCUs

Many microcontrollers feature little security functionality

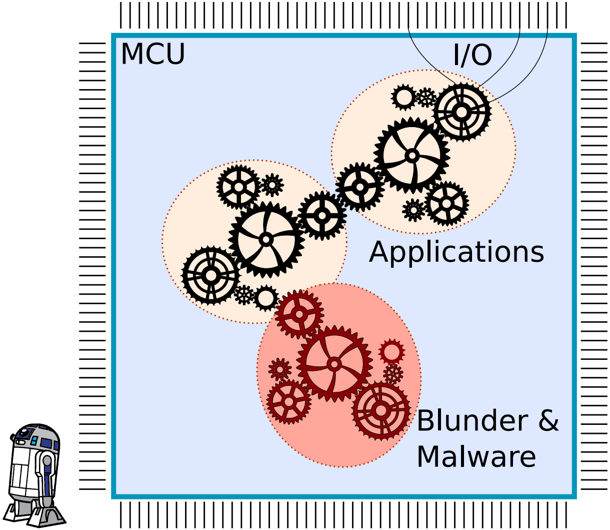
- Applications share address space



Isolation and Attestation on Light-Weight MCUs

Many microcontrollers feature little security functionality

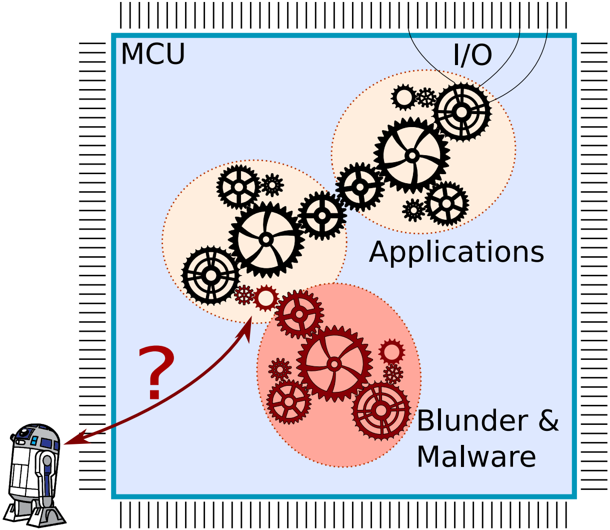
- Applications share address space
- Boundaries between applications are not enforced



Isolation and Attestation on Light-Weight MCUs

Many microcontrollers feature little security functionality

- Applications share address space
- Boundaries between applications are not enforced
- Integrity? Confidentiality? Authenticity?



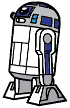
Isolation and Attestation on Light-Weight MCUs

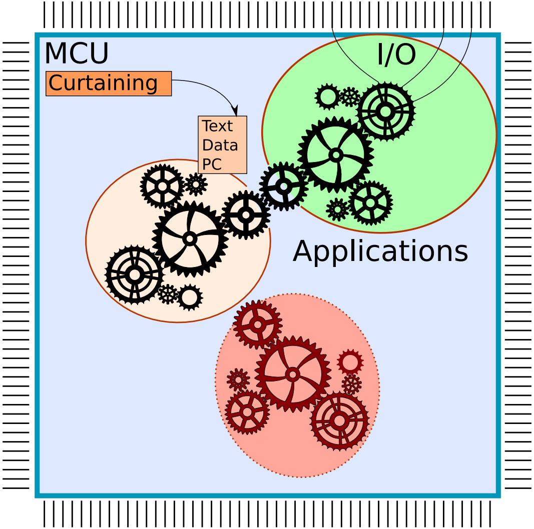
Many microcontrollers feature little security functionality

- Applications share address space
- Boundaries between applications are not enforced
- Integrity? Confidentiality? Authenticity?

Trusted Computing aims to fix that:

- Strong isolation, restrictive interfaces, exclusive I/O





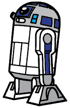
Isolation and Attestation on Light-Weight MCUs

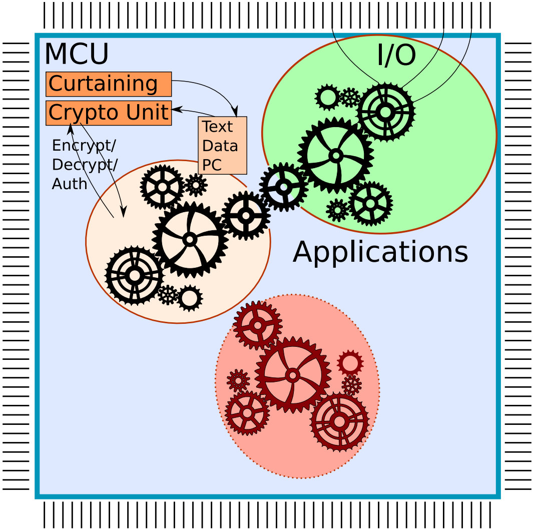
Many microcontrollers feature little security functionality

- Applications share address space
- Boundaries between applications are not enforced
- Integrity? Confidentiality? Authenticity?

Trusted Computing aims to fix that:

- Strong isolation, restrictive interfaces, exclusive I/O





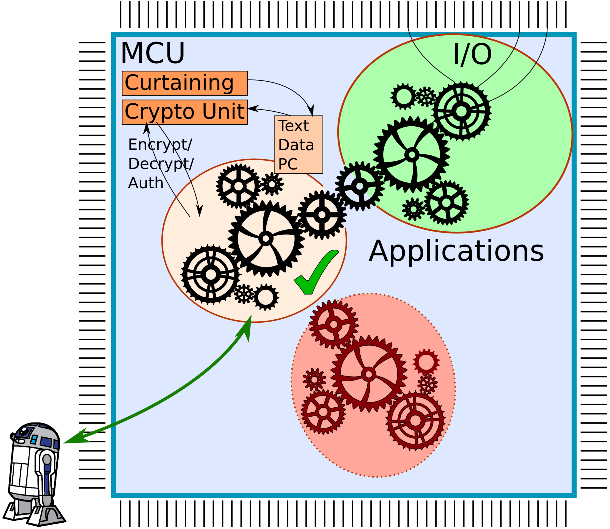
Isolation and Attestation on Light-Weight MCUs

Many microcontrollers feature little security functionality

- Applications share address space
- Boundaries between applications are not enforced
- Integrity? Confidentiality? Authenticity?

Trusted Computing aims to fix that:

- Strong isolation, restrictive interfaces, exclusive I/O
- Built-in cryptography and (remote) attestation



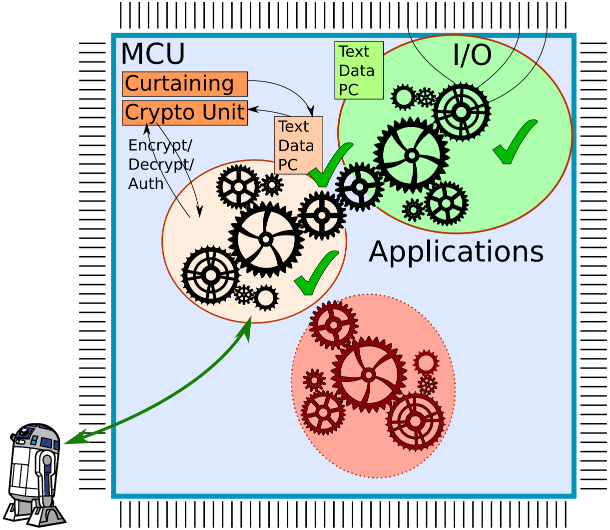
Isolation and Attestation on Light-Weight MCUs

Many microcontrollers feature little security functionality

- Applications share address space
- Boundaries between applications are not enforced
- Integrity? Confidentiality? Authenticity?

Trusted Computing aims to fix that:

- Strong isolation, restrictive interfaces, exclusive I/O
- Built-in cryptography and (remote) attestation



Comparing Hardware-Based Trusted Computing Architectures

	Isolation	Attestation	Sealing	Dynamic RoT	Code Confidentiality	Side-Channel Resistance	Memory Protection	Lightweight	Coprocessor	HW-Only TCB	Preemption	Dynamic Layout	Upgradeable TCB	Backwards Compatibility	Open-Source	Academic	Target ISA
AEGIS	●	●	●	●	●	○	●	○	○	●	●	●	○	●	○	●	–
TPM	○	●	●	○	●	–	◓	○	●	●	–	–	○	●	○	○	–
TXT	●	●	●	●	●	●	◓	○	●	●	○	●	○	●	○	○	x86_64
TrustZone	●	○	○	●	○	○	○	○	○	○	●	●	○	●	○	○	ARM
Bastion	●	○	●	●	●	○	●	○	○	○	●	●	●	●	○	●	UltraSPARC
SMART	○	●	○	●	○	–	○	●	○	○	–	–	○	●	○	●	AVR/MSP430
Sancus 1.0	●	●	○	●	○	●	○	●	○	●	○	○	○	●	●	●	MSP430
Soteria	●	●	○	●	●	●	○	●	○	●	○	○	○	●	●	●	MSP430
Sancus 2.0	●	●	○	●	●	●	○	●	○	●	◓	○	○	●	●	●	MSP430
SecureBlue++	●	○	●	●	●	○	●	○	○	●	●	●	○	●	○	○	POWER
SGX	●	●	●	●	●	○	●	○	○	○	●	●	●	●	○	○	x86_64
Iso-X	●	●	○	●	○	○	●	○	○	○	●	●	●	●	○	●	OpenRISC
TrustLite	●	●	○	○	○	●	○	●	○	○	●	●	●	●	○	●	Siskiyou Peak
TyTAN	●	●	●	●	○	●	○	●	○	○	●	●	●	●	○	●	Siskiyou Peak
Sanctum	●	●	●	●	●	●	○	○	○	○	●	●	●	●	◓	●	RISC-V

● = Yes; ◓ = Partial; ○ = No; – = Not Applicable

Adapted from "Hardware-Based Trusted Computing Architectures for Isolation and Attestation", Maene et al., IEEE Transactions on Computers, 2017. [MGdC^{+}17]

Sancus: Strong and Light-Weight Embedded Security [NVBM$^+$17]

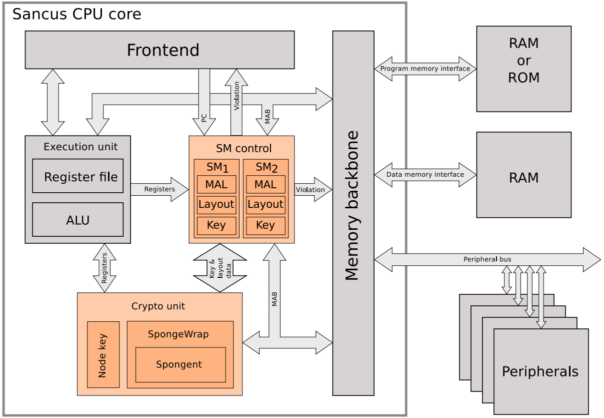
Extends openMSP430 with strong security primitives

- Software Component Isolation
- Cryptography & Attestation
- Secure I/O through isolation of MMIO ranges

Efficient

- Modular, $\leq$ 2 kLUTs
- Authentication in μs
- $+$ 6% power consumption

Cryptographic key hierarchy for software attestation



Isolated components are typically very small ($<$ 1kLOC)

Sancus is Open Source: https://distrinet.cs.kuleuven.be/software/sancus/

Sancus: Strong and Light-Weight Embedded Security [NVBM$^+$17]

Extends openMSP430 with strong security primitives

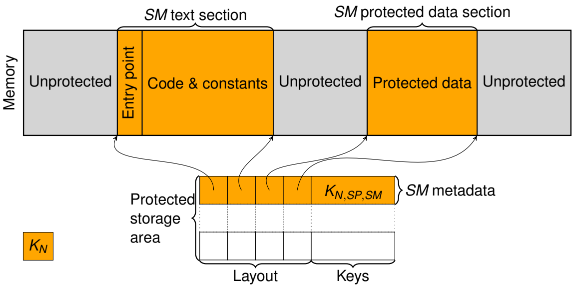
- Software Component Isolation
- Cryptography & Attestation
- Secure I/O through isolation of MMIO ranges

Efficient

- Modular, $\leq$ 2 kLUTs
- Authentication in μs
- $+$ 6% power consumption

Cryptographic key hierarchy for software attestation

N = Node; SP = Software Provider / Deployer
SM = protected Software Module



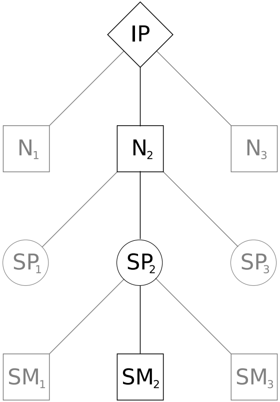
Isolated components are typically very small ($<$ 1kLOC)

Sancus is Open Source: https://distrinet.cs.kuleuven.be/software/sancus/

Attestation and Communication with Sancus

Ability to use $K_{N,SP,SM}$ proves the integrity and isolation of SM deployed by SP on N

- Only N and SP can compute $K_{N,SP,SM}$
  N knows K_N and SP knows K_{SP}
- $K_{N,SP,SM}$ on N is computed after enabling isolation
  No isolation, no key; no integrity, wrong key
- Only SM on N is allowed to use $K_{N,SP,SM}$
  Through special instructions



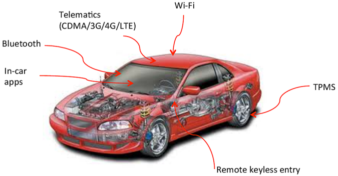
Remote attestation and secure communication by Authenticated Encryption with Associated Data

- Confidentiality, integrity and authenticity
- Encrypt and decrypt instructions use $K_{N,SP,SM}$ of the calling SM
- Associated Data can be used for nonces to get freshness

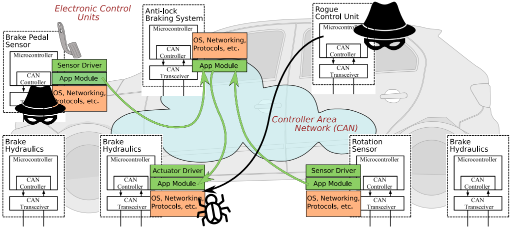
Secure Automotive Computing with Sancus [VBMP17]

Modern cars can be hacked!

- Network of more than 50 ECUs
- Multiple communication networks
- Remote entry points
- Limited built-in security mechanisms



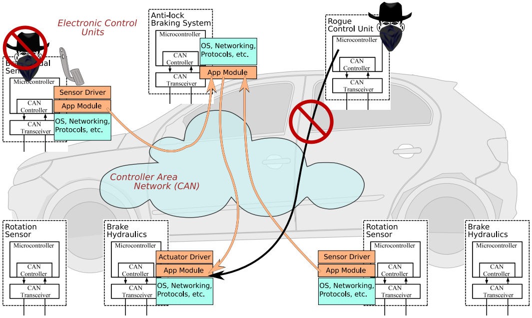
Miller & Valasek, “Remote exploitation of an unaltered passenger vehicle”, 2015



Sancus brings strong security for embedded control systems:

- Message authentication
- Trusted Computing: software component isolation and cryptography
- Strong software security
- Applicable in automotive, ICS, IoT, . . .

My usual work: Trusted Computing for Embedded Control Systems



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

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

Trusted Computing & Sancus

① Strong application isolation and attestation

② Requires correct hardware and software

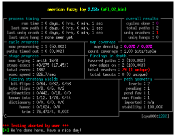
Security

① Understand the system

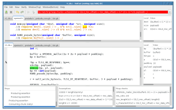
② Understand the security requirements

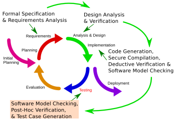
③ Understand the attacker

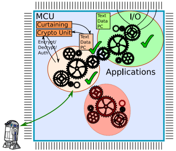
④ Understand and embrace change

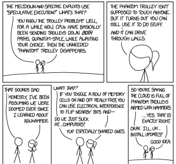


```
american fuzzy lop 2.52b (afl_02_bin)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 0 min, 1 sec        │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 1 sec        │  total paths : 2      │
│ 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 : 1 (50.00%)    │    map density : 0.02% / 0.02%             │
│ paths timed out : 0 (0.00%)     │ count coverage : 1.00 bits/tuple           │
├─ stage progress ────────────────┼─ findings in depth ────────────────────────┤
│  now trying : arith 16/8        │ favored paths : 2 (100.00%)                │
│ stage execs : 48/275 (17.45%)   │  new edges on : 2 (100.00%)                │
│ total execs : 1887              │ total crashes : 79 (1 unique)              │
│  exec speed : 826.7/sec         │  total tmouts : 0 (0 unique)               │
├─ fuzzing strategy yields ───────┴───────────────┬─ path geometry ────────────┤
│   bit flips : 0/64, 0/62, 0/58                  │    levels : 2              │
│  byte flips : 0/8, 0/6, 0/2                     │   pending : 1              │
│ arithmetics : 0/442, 0/18, 0/0                  │  pend fav : 1              │
│  known ints : 1/12, 1/78, 0/44                  │ own finds : 1              │
│  dictionary : 0/0, 0/0, 0/0                     │  imported : n/a            │
│       havoc : 0/1024, 0/0                       │ stability : 100.00%        │
│        trim : 76.47%/4, 0.00%                   ├────────────────────────────┘
^C────────────────────────────────────────────────┘          [cpu000:128%]

+++ Testing aborted by user +++
[+] We're done here. Have a nice day!
```









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

P. Maene, J. Gotzfried, R. de Clercq, T. Muller, F. Freiling, and I. Verbauwhede.
Hardware-based trusted computing architectures for isolation and attestation.
IEEE Transactions on Computers, PP(99):1–1, 2017.

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

C. Miller and C. Valasek.
Remote exploitation of an unaltered passenger vehicle.
Black Hat USA, 2015.

J. Noorman, J. Van Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling.
Sancus 2.0: A low-cost security architecture for IoT devices.
ACM Transactions on Privacy and Security (TOPS), 20:7:1–7:33, 2017.

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.

References III

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