Introduction to Software Verification

Vincent Penelle

<vpenelle@u-bordeaux.fr>

LaBRI, Université de Bordeaux

September 3, 2018

— *Introduction* —

Software Verification: Why?

Software have bugs !!!

Causes of bugs

- Usage errors.
- Programming errors.
- Design errors.
- Compiler errors.
- Hardware errors.

Effects of bugs

- Loss of reputation (Consumer electronics).
- Loss of efficiency (Software industry).
- Loss of money (Banking).
- Loss of devices (Spatial exploration).
- Death of people (Medical industry).

Example: OpenSSL Heartbleed

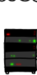
TLS Heartbeat Protocol

This is an extension of the of the TLS (Transport Layer Security) protocol that allow an host to ensure that the server is still alive.
The host may ask for an immediate answer from the server by giving a string and the size of the string. The server must give the string back as a proof it is still alive.

Timeline

- **21 March 2014**: Discovery of the bug by Neel Mehta at Google Security.
- **1 April 2014**: Google Security notify OpenSSL dev team about it.
- **3 April 2014**: Re-discovery of the bug by Codenomicon and re-notification.
- **7 April 2014**: Heartbleed bug becomes public.
- **7 April 2014**: OpenSSL 1.0.1g is released with a fix.
- **April 2014**: Several servers are attacked Worldwide and compromised.
- **April 2014**: Discussions on how static-analyzers can catch this kind of bug.

Explanation of the bug (XKCD)



- **Step 1**: Send a string and the string length to the server;
- **Step 2**: The server receive the message and reply by sending back the string;
- **Step 3**: The client get the string back.

- **Step 1**: Send the smallest string possible and the maximum string length to the server;
- **Step 2**: The server receive the message and reply by sending back the minimal string and part of the process memory;
- **Step 3**: The client get the string back plus extra-information.

The Bug

```
struct
{
  HeartbeatMessageType type;
  uint16 payload_length;
  opaque payload[HeartbeatMessage.payload_length];
  opaque padding[padding_length];
} HeartbeatMessage;
```

The problem was that the HeartbeatMessage arrives via an SSL3_RECORD structure, a basic building block of SSL/TLS communications. The key fields in SSL3_RECORD are given below; length is how many bytes are in the received HeartbeatMessage and data is a pointer to that HeartbeatMessage.

```
struct ssl3_record_st
{
  unsigned int length;    /* How many bytes available */
  [...]
  unsigned char *data;    /* pointer to the record data */
  [...]
} SSL3_RECORD;
```

So, just to be clear, the SSL3 record's data points to the start of the received HeartbeatMessage and length is the number of bytes in the received HeartbeatMessage. Meanwhile, inside the received HeartbeatMessage, payload_length is the number of bytes in the arbitrary payload that has to be sent back.
Whoever sends a HeartbeatMessage controls the payload_length but as we will see, this is never checked against the parent SSL3_RECORD's length field, allowing an attacker to overrun memory. F

Spotting Heartbleed bugs

Let's be clear: **it is trivial to create a static analyzer that runs fast and flags heartbleed**. I can accomplish this, for example, **by flagging a taint error** in every line of code that is analyzed. The task that is truly difficult is to create a static analysis tool that is performant and that has a high signal to noise ratio for a broad range of analyzed programs.

This is the design point that Coverity is aiming for, and while it is an excellent tool **there is obviously no general-purpose silver bullet**: halting problem arguments guarantee the non-existence of static analysis tools that can reliably and automatically detect even simple kinds of bugs such as divide by zero.

In practice, it's not halting problem stuff that stops analyzers but rather code that has a lot of indirection and a lot of data-dependent control flow. If you want to make a program that is robustly resistant to static analysis, implement some kind of interpreter.

John Regehr, 12 April 2014.

Spotting Heartbleed bugs

```
2443
2444        /* Read type and payload length first */
2445        hbtype = *p++;
```

1. **byte_swapping:** Performing a byte swapping operation on `p[1]` implies that it came from an external source, and is therefore tainted.

2. **var_assign_var:** Assigning: `payload = ((unsigned int)p[0] << 8) | (unsigned int)p[1]` . Both are now tainted.

```
2446        n2s(p, payload);
2447        pl = p;
2448
```

3. Condition `s->msg_callback` , taking true branch

```
2449        if (s->msg_callback)
2450                s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
2451                        &s->s3->rrec.data[0], s->s3->rrec.length,
2452                        s, s->msg_callback_arg);
2453
```

4. Condition `hbtype == 1` , taking true branch

```
2454        if (hbtype == TLS1_HB_REQUEST)
2455                {
2456                unsigned char *buffer, *bp;
2457                int r;
2458
2459                /* Allocate memory for the response, size is 1 bytes
2460                 * message type, plus 2 bytes payload length, plus
2461                 * payload, plus padding
2462                 */
2463                buffer = OPENSSL_malloc(1 + 2 + payload + padding);
2464                bp = buffer;
2465
2466                /* Enter response type, length and copy payload */
2467                *bp++ = TLS1_HB_RESPONSE;
2468                s2n(payload, bp);
```

CID 22924 (#1 of 1): Untrusted value as argument (TAINTED_SCALAR)

5. **tainted_data:** Passing tainted variable `payload` to a tainted sink. [Note: The source code implementation of the function has been overridden by a builtin model.]

```
2469                memcpy(bp, pl, payload);
2470                bp += payload;
2471                /* Random padding */
2472                RAND_pseudo_bytes(bp, padding);
2473
```

References

- **Diagnosis of the OpenSSL Heartbleed Bug**, by Sean Cassidy, 7 April 2014.
  https://www.seancassidy.me/diagnosis-of-the-openssl-heartbleed-bug.html
- **Attack of the week: OpenSSL Heartbleed**, by Matthew Green, 8 April 2014.
  http://blog.cryptographyengineering.com/2014/04/attack-of-week-openssl-heartbleed.html
- **Anatomy of OpenSSL's Heartbleed: Just four bytes trigger horror bug**, by Chris William, 9 April 2014 (The Register).
  http://www.theregister.co.uk/2014/04/09/heartbleed_explained/
- **Heartbleed and Static Analysis**, by John Regehr, 10 April 2014.
  http://blog.regehr.org/archives/1125
- **A New Development for Coverity and Heartbleed**, by John Regehr, 12 April 2014.
  http://blog.regehr.org/archives/1128
- **On Detecting Heartbleed with Static Analysis**, by Andy Chou, 13 April 2014.
  http://security.coverity.com/blog/2014/Apr/on-detecting-heartbleed-with-static-analysis.html
- **Finding Heartbleed with CodeSonar**, by Paul Anderson, 1 May 2014.
  http://www.grammatech.com/blog/finding-heartbleed-with-codesonar
- **How did Heartbleed remain undiscovered, and what should we do about it?**, by Michael Hicks, 1 July 2014.
  http://www.pl-enthusiast.net/2014/07/01/

Spotting bugs: Verification Versus Validation

Software Verification

An attempt to **prove formally** that the **software is fulfilling a specification**.

$$\textbf{Software} \models \textbf{Specification}$$

Don't mix up "**Software Verification**" and "**Software Validation**" !

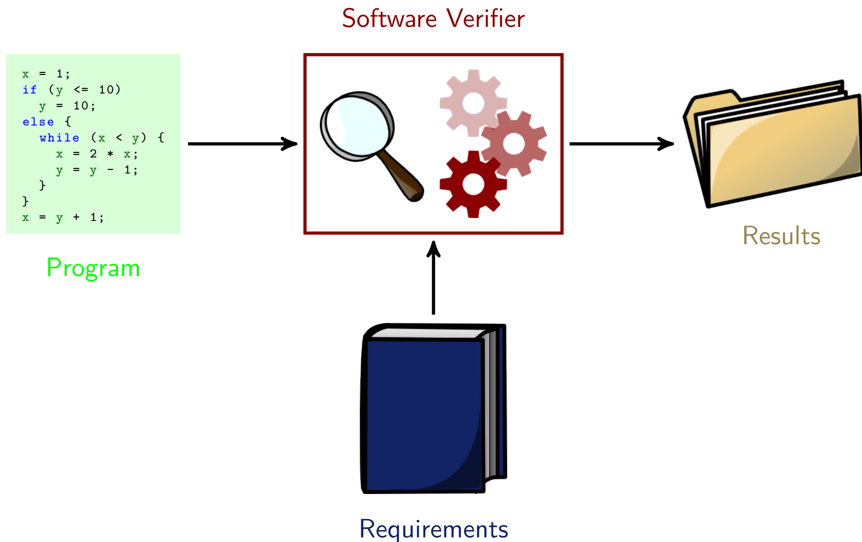
Software Verification

- Perform a **symbolic analysis** of the software through a **formal model** of the software.
- **Exhaustively** check the software.
- Verification is performed on **a model**, not on the actual software.

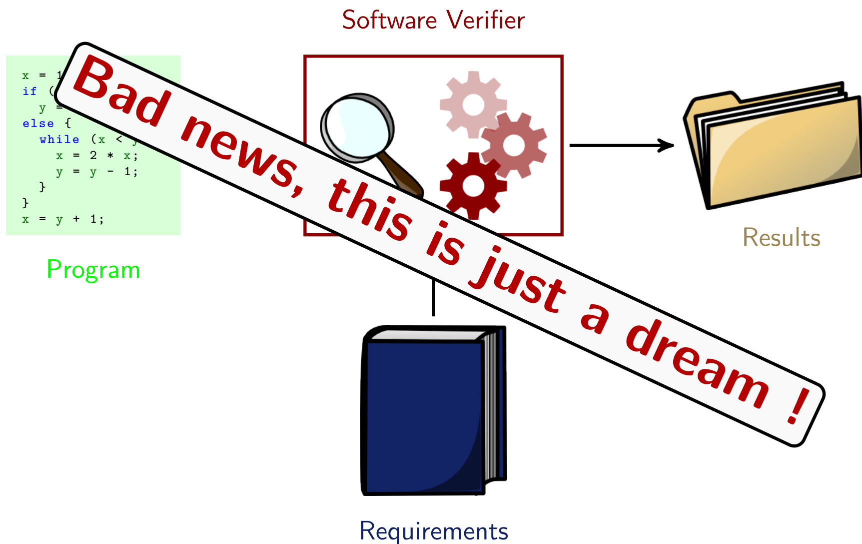
Software Validation

- Perform **multiple runs** of the software on **given inputs** and check results against **expected outputs**.
- Check **one input** at the time.
- Validation is performed on the **real software** in the **real context**.

Software Verification: How ?

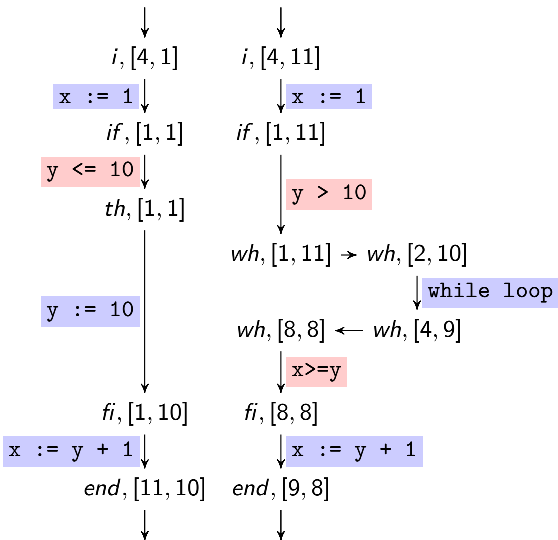


Software Verification: How ?



Naive Idea: Graph of all configurations

```
x = 1;
if (y <= 10) {
  y = 10;
}
else {
  while (x < y) {
    x = 2 * x;
    y = y - 1;
  }
}
x = y + 1;
```



Practical Limit Combinatorial Explosion

The amount of available memory of computers is finite, therefore, we could theoretically systematically explore the whole graph for checking a property:

Software Verification is **decidable** for finite-state systems.

Practical Limit Combinatorial Explosion

The amount of available memory of computers is finite, therefore, we could theoretically systematically explore the whole graph for checking a property:

Software Verification is **decidable** for finite-state systems.

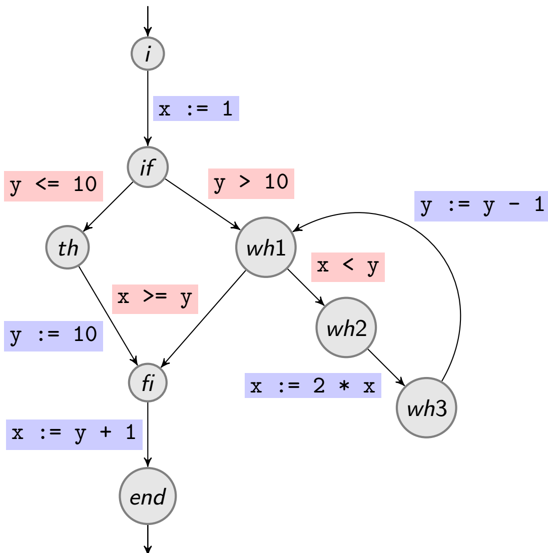
But, even with bounded memory, complexity in practice is too high for finite-state model-checking:

- 1 megabyte (1 000 000 bytes) of memory $\approx 10^{2\,400\,000}$ states
- 1000 variables $\times$ 64 bits $\approx 10^{19\,200}$ states
- optimistic limit for finite-state model checkers: 10^{100} states

And, the complexity of verification algorithms are, most of the time, beyond NP !

Other naive idea: consider infinite memory

```
x = 1;
if (y <= 10) {
  y = 10;
}
else {
  while (x < y) {
    x = 2 * x;
    y = y - 1;
  }
}
x = y + 1;
```



This is a finite presentation of an infinite graph!

Theoritical Limit: Undecidability

As long as we can **describe them finitely**, infinite graphs are not a problem (e.g., Turing Machines). But,

Rice's Theorem

Any non-trivial semantic property of programs is undecidable.

Classical Example: Termination

There exists no algorithm which can solve the **halting problem** on a **Turing-complete language**'s program:

- given a description of a program as input,
- decide whether the program terminates or loops forever.

Summary: Finite and Infinite Graphs

Finite Graphs:

- **Describes exhaustively a program (in theory)**
- **Every problem is decidable**
- In practice, way too big

Infinite Graphs:

- **Allow to represent unbounded values**
- **Shorter description of a program**
- Almost all problems are undecidable

$\Rightarrow$ We need a compromise between expressiveness and decidability

Possible Compromises

Less Expressive Logics

Using constrained theories help to build smaller proofs.

- Propositional logic (with finite number of propositions);
- Presburger arithmetic (only addition and multiplication by a constant);
- Quantifier-free arithmetic over reals (Tarski decidability theorem).

Less Expressive Models

Using constrained models to regain decidability of some properties.

- (Higher-order) Pushdown Automata
- Vector Addition Systems

Incomplete Methods

To ensure termination of the algorithms.

- Approximate Algorithms
  - **Always terminate**;
  - **Indefinite answer (yes/no/dontknow)**.
- Exact Semi-Algorithms
  - **Definite answer (yes/no)**;
  - **May not terminate**.