UPPSALA
UNIVERSITET

IT 14 044
Examensarbete 15 hp
Augusti 2014

Finding vulnerabilities using automatic test generation

Jordi Bueno Domínguez

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Finding vulnerabilities using automatic test generation

Jordi Bueno Domínguez

Software bugs are still present in modern software, and they are a major concern for every user, specially security related bugs. Classical approaches for bug detection fall short to uncover some of them, as it has been proved on several occasions when a hidden bug has been used to compromise the security of many systems.
In this report an approach for automatic bug detection is presented and analysed. Using KLEE, a tool that can explore all the possible paths in a piece of code, bugs can be discovered. As an example for bug detection in a security software, the Heartbleed bug that affected the OpenSSL library is analysed. The behaviour of this bug is explained here, and KLEE is used to expose this bug. If this worked, it would be useful for developers in order to prevent dangerous bugs from staying undetected.
The results show that the tool is not ready to be used in real software due to its limitations. However, despite the difficulties these limitations pose, KLEE proves to be useful in a controlled scenario. As long as the software is kept simple, the tool can be used to effectively execute all the code. With some improvements, it could be a major step for a future without bugs.

Handledare: Magnus Johansson
Ämnesgranskare: Björn Victor
Examinator: Olle Gällmo
IT 14 044
Tryckt av: Reprocentralen ITC

Contents

1. Introduction

Computers, in all their forms, are spread around the world being part of our day-to-day life. We rely on them for almost everything, and we expect them to be reliable. However, there are errors in them which, when they are not fixed on time, can be used by the wrong people to do the wrong thing. Some of these errors are in the software and are known as software bugs. The usual problem with bugs is that they are hidden, which prevent the developers and maintainers to get rid of them.

This report analyses the viability of a new approach in bug detection. Due to the difficulty inherent on locating software bugs, a lot of them may lie unnoticed for years. Some of these bugs can be exploited by an attacker with severe consequences. An example of such a bug is Heartbleed [Codb], a bug in OpenSSL [The] that affected around half a million widely trusted websites [Paua] according to Netcraft [Net], and a lot more unaccounted for.

A tool that allows developers and testers to execute every branch in their code can be used to trace the execution of the program under any possible input. Using logs and other crafted outputs, the tool can be used to execute all the paths and can reveal unexpected behaviour and bugs. KLEE [Cri] is such a tool, although not the only one, and it is been chosen for this analysis.

1.1 Motivation

An international security company located in Uppsala requested the viability of using KLEE on their software. The intention is to determine if its possible to find hidden bugs in their code by using this tool. Nobody at the company has ever used such a tool, so an analysis of its capabilities is required. According to the initial information provided by the KLEE developers, the tools has been used on dummy examples and some of the core utilities of Unix, like ls, echo, cat, etc.

The company, FOXt. (FOX Technologies AB) [FOX], needs to ensure that their security product is reliable. Their software, BoKS ServerControl [BoK], unifies access management and governance for servers, applications, databases and desktops. This means that all their clients information and resources are accessed using the authentication mechanisms provided by BoKS. It is very important to ensure that there are no vulnerabilities an attacker could use to compromise the client's infrastructure and information.

1.2 Objectives and limitations

The objective of the project is to try and understand the capabilities of KLEE. The purpose is to determine if the tool is mature enough to be useful in real software. The code that the company wants to examine is part of their software product, FoxT BoKS ServerControl. The ultimate goal is to run KLEE on this software, however the task requires far more time and deep knowledge of the company's product than available. Because of this, a different security software may be used.

During the time of this project, the Heartbleed bug was disclosed in OpenSSL. As this library is a real security software with few dependencies and a significant bug is already found, it is the perfect test for this project. The Heartbleed bug will be explained, and KLEE will be used to locate the bug in an unpatched version of OpenSSL.

However KLEE failed to execute OpenSSL, the reasons are explained later on. This failure forced the project to reproduce the bug in a controlled environment where KLEE could work. The results must prove if KLEE is a viable option to invest in in the future. Apart from its capabilities for bug detection, other uses for these kind of tools shall be discussed, as well as the related works around them.

2. Background

The two main parts of this report are the tool in use, KLEE, and the Heartbleed bug. Both things are explained here.

2.1 KLEE

The main project on which KLEE depends is LLVM [Chrb] (formerly Low Level Virtual Machine). The LLVM Project is a compiler infrastructure. Despite its former name, it is not related to traditional virtual machines. The name is not used as an acronym anymore and LLVM is the full name of the project.

The project started as a research project at the University of Illinois [LA04]. Its objective was to provide a modern compilation strategy based on SSA [W.A98] (static single assignment). The use of SSA enables a lot of compiler optimizations that make the final applications faster. It supports both dynamic and static compilation for any programming language. LLVM is the base for several subprojects used in commercial products, open source projects and academic environment.

One of these sub-projects is KLEE. It was developed by Cristian Cadar, Daniel Dunbar and Dawson Engler, at Stanford University. The project implements a symbolic virtual machine that tries to evaluate all dynamic paths in an application using a theorem prover, [Vij] (Simple Theorem Prover), in an effort to locate errors and prove properties of functions. STP receives as input a formula, which can be expressed in C or other programing languages, and it outputs whether or not the formula is satisfied. KLEE is also able to produce a test case when it finds a bug.

It is basically a symbolic execution tool that can generate tests automatically for a diverse set of environmentally intensive and complex programs. KLEE runs a code using symbolic input allowed to take any possible value, instead of running it with random or manually constructed input. The program inputs are substituted by symbolic values and some program operations are changed for others that can operate with these symbolic values. When the execution flow finds a branch where the condition is a symbolic value, it follows both branches at the same time. For each branch, a set of constraints are stored with the values that produced it, and must be maintained during that branch. In the event of a bug or termination of the program, a test case with the concrete values of the branch is generated.

There are two main problems with symbolic execution tools such as KLEE. The first one is the exponential growth in the number of paths in real code, making it impractical to follow all of them in reasonable time. The other one is the interaction with the environment, such as the network or the operating system itself. This problem is usually irresolvable without modifying and limiting the scope of the program.

Furthermore, KLEE is not capable of handling floating point numbers, assembly code, threads, memory objects of variable size, some types of jumps (setjmp and longjmp) and sockets. KLEE does not model network calls either. All the analysed code must be modified to avoid these limitations.

As described below, KLEE can be used in some real world software. However most of the evidence suggests that the state of the project is not mature enough. According to Chris Lattner, main developer behind LLVM, KLEE is a great little project, but it is limited by not being practical to run on large-scale applications [Chra].

2.1.1 Previous and related works

KLEE has been used in the past to analyse the GNU Coreutils and Busybox in order to find bugs and inconsistencies between both [CDE08]. This was the case study of the main developers of KLEE. Their results are described as ”promising”. Although they are widely used programs, the core utils are very limited in functionality. They are simple and compact, focusing on one action, usually processing an input stream or arguments. Their complexity is not comparable to something like OpenSSL.

KLEE has also been used to identify inconsistent behaviour on the client side of a multiplayer videogame, allowing the developers to find cheats [Dar10]. The tested game was the open source game XPilot [XPi]. In order to use KLEE, the developers had to modify the code and log all the operations. Then they used the logs as a model of the network, this workaround is due to KLEE limitations in network analysis.

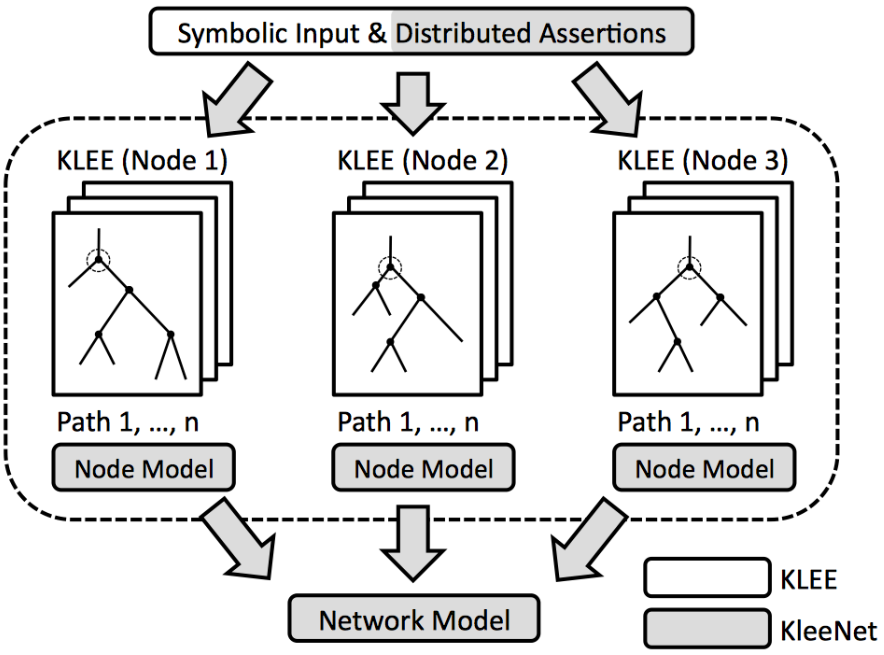

Figure 2.1: KleeNet overview [SLA+10]. KleeNet manages distributed program execution, communication, and user-defined distributed assertions. Extensions of KleeNet to KLEE are shaded.

Another use for KLEE has been for testing a network of wireless sensors, using distributed execution paths between the peers. The project is an extension of KLEE called KleeNet [SLA+10]. Because of the KLEE limitations when interacting with the environment, specially network, this project focuses on the distributed part. Using Node and Network models, KleeNet extends KLEE's capabilities for interacting between different processes. Its usage as an extension for networking in KLEE should be studied further in the future in order to improve the range of applications KLEE could be used on.

A modified version of KLEE is one of the tools used in RevNIC [Vit10], a project that automatically reverse engineers proprietary closed source drivers and generates new code to reproduce the same behaviour. The same people used their system to also test the drivers for bugs [KCC10].

Yet another interesting project uses one of the versions of KLEE to cross-check the execution between an OpenCL program and its simple sequential counterpart [CCK11]. The symbolic results of the OpenCL execution are compared against their plain C or C++ implementation in order to detect discrepancies between them.

Not every project uses symbolic execution or code coverage for bug detection. Some projects use it for other purposes like conformance check [LXC13] or location of data structures in binaries [SSB11]. However most of the projects are still bug related, as it is the most useful usage of code coverage tools like KLEE.

2.2 Heartbleed bug

Heartbleed was disclosed in April 7, 2014. The bug was named after the heartbeat extension of the Transport Layer Security (TLS) protocol where it was found. It is a security bug in the open source OpenSSL library, which is used for the TLS in many machines across the Internet. TLS provides cryptographic protocols used to make the communication across the Internet secure. Both servers and clients can be exploited using the bug.

It is a buffer over-read vulnerability, as it allows the attacker to read more data from the victim than it should. The cause of the bug is the use of non-validated data as input for a memcpy() call. Memcpy, as its name indicates, copies chunks of memory from one place to another. As a result, sensitive data in the program's memory can be extracted from the machine, including passwords, private keys and login cookies. HTTPS, VPN and other protocols relying on the TSL were compromised.

The affected versions of OpenSSL are from the 1.0.1 to 1.0.1f, included. This last version (1.0.1f) will be used in the project to test the capabilities of KLEE. The same day of the bug's disclosure, OpenSSL released the version 1.0.1g, free of the Heartbleed bug.

Heartbleed had a huge impact in the technological circles, internet and even mainstream media and governments [Kel]. According to Netcraft, a security internet company, around half a million trusted websites they had analysed were vulnerable to Heartbleed. They also released a extensions for several web browsers to detect if a server was compromised [Paub].

After the disclosure of the bug and all the received attention, there were several consequences. The team behind OpenBSD [Opea], an operating system focused on security, decided to fork OpenSSL and refactor it as LibreSSL, because OpenSSL was beyond repair [Jona]. OpenSSL has been used by a lot a companies and individuals, and almost none contributed to it. After the incident several tech giants like IBM, Intel, Microsoft, Facebook, Google and others, decided to finally contribute to its maintenance [Jonb]. Since then, up to seven new vulnerabilities, two of them critical, were found in OpenSSL and fixed the 5th of June [Opeb] [Lin].

Figure 2.2: Security company Codenomicon [Coda] gave Heartbleed both a name and a logo.

2.2.1 TLS heartbeat

The heartbeat extension is a keep-alive feature. It is used to maintain the connection open without having to renegotiate the connection each time. It was proposed as a standard in February 2012 by RFC 6520 [SToAS^{+}12], and was introduced in the 1.0.1 version of OppenSSL, which was released on March of the same year. The release came with the heartbeat feature enabled by default and included the Heartbleed bug since the beginning.

The basic functioning is as follows. One end of the connection sends some arbitrary data, a payload. On the other end, the data is received and sent back to the source to prove that the connection is still working properly. Along with the payload, the length of it is included in a 16 bit integer. The data structure according to the RFC 6520 is as follows.

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

2.2.2 The error

The communications in the TLS, as well as its predecessor SSL, are built with SSL3_RECORD. The structure code in the OpenSSL implementation looks like this.

```
typedef struct ssl3_record_st
       {
/*r */ int type;            /* type of record */
/*rw*/ unsigned int length; /* How many bytes available */
/*r */ unsigned int off;    /* read/write offset into `buf` */
/*rw*/ unsigned char *data; /* pointer to the record data */
/*rw*/ unsigned char *input; /* where the decode bytes are */
/*r */ unsigned char *comp; /* only used with decompression -
    malloc()ed */
/*r */ unsigned long epoch; /* epoch number, needed by DTLS1 */
/*r */ unsigned char seq_num[8]; /* sequence number, needed by
    DTLS1 */
       } SSL3_RECORD;
```

The fields of the data structure that concern us are length and data. The data pointer indicates the beginning of the message, and the length field indicates how many bytes are available.

This structure is used to send the heartbeat messages, in which another length (payload_length) is included. The length in SSL3_RECORD must be 1 byte for the type, plus 2 bytes for the length, plus payload_length, plus 16 bytes of padding. The sender specified how long was his message and the receiver knows the size of the message he received. However, in the affected versions of OpenSSL, this is never checked. Even more, the length used when sending the message back is the one indicated by payload_length.

When an attacker sends a heartbeat message indicating a larger length than the real one, the victim responds with a new message. It starts at the original data and continues, over-reading from his own memory, until it fills the payload_length specified by the attacker. The maximum length is 64 Kbyte, but the attacker can repeat the attack indefinitely as the bug leaves no trace or record of what happened.

2.2.3 The code

The error can be found in the function tls1_process_heartbeat(SSL *s) [openssl-1.0.1f/ssl/t1_lib.c]. The macros s2n and n2s are located in [openssl-1.0.1f/ssl/ssl_locl.h].

The following variables are used in the function, note how p points to the beginning of the heartbeat message.

```
    unsigned char *p = &s->s3->rrec.data[0], *pl;
    unsigned short hbtype;
    unsigned int payload;
    unsigned int padding = 16; /* Use minimum padding */
```

The main error is here, where the payload length is read from the message but not compared with the length in the SSL3 structure.

```
    /* Read type and payload length first */
    hbtype = *p++;
    n2s(p, payload);
    pl = p;
```

The n2s macro reads into payload the data pointed by p, which is the length, and increments the pointer. The s2n macro, which will be used later, does the opposite thing. It writes the length into the specified position and increments the pointer.

```
#define n2s(c,s)    ((s=(((unsigned int)(c[0]))<< 8)| \
                       (((unsigned int)(c[1])) )),c+=2)
#define s2n(s,c)    ((c[0]=(unsigned char)(((s)>> 8)&0xff), \
                     c[1]=(unsigned char)(((s) )&0xff)),c+=2)
```

After that, the program allocates memory for the response message.

```
            /* Allocate memory for the response, size is 1 bytes
             * message type, plus 2 bytes payload length, plus
             * payload, plus padding
             */
            buffer = OPENSSL_malloc(1 + 2 + payload + padding);
            bp = buffer;
```

The response message is composed with the TLS1_HB_RESPONSE type, the same payload length that was specified by the sender and a direct memcpy() of the received payload. The amount of data to be copied is indicated by the untrusted payload length. This is were the damage is made. The function memcpy(bp, pl, payload), copies payload bytes of memory from pl to bp. The way it is supposed to work is by copying the message from the request into the response. When the amount to be copied is greater than the length of the request, not only the original message will be copied, but also part of the program's memory. In the program's memory is stored all the recent activity, including keys from other clients.

```
            /* Enter response type, length and copy payload */
            *bp++ = TLS1_HB_RESPONSE;
            s2n(payload, bp);
            memcpy(bp, pl, payload);
            bp += payload;
            /* Random padding */
            RAND_pseudo_bytes(bp, padding);

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

2.2.4 The fix

The correction of the code was introduced in the version 1.0.1g of OpenSSL. The patch is a simple check with the real length of the message specified in the SSL3 structure. The snippet of code with the new addition looks like this.

```
hbtype = *p++;
n2s(p, payload);
if (1 + 2 + payload + 16 > s->s3->rrec.length)
   return 0; /* silently discard per RFC 6520 sec. 4 */
pl = p;
```

As the comment in the new code indicates, it seems that the previous implementation was not following the standard correctly.

3. Method

The usage of KLEE is not straightforward nor user friendly. It uses a command line interface for its tools and outputs text files for the results. The installation process is also error prone and dependant on specific versions of software dependencies. Above all, the hardest thing is to modify your code to avoid KLEE's limitations.

3.1 Execution

Before getting started with a bigger code, the execution process of KLEE must be explained. In order to do so, a small function is used to show the simplest KLEE execution. The following function receives three integers as input and returns the more repeated one, or the maximum number in case all of them are different. The function has been implemented with several nested ternary operators, which makes it difficult to manually debug it.

```
int example(int a, int b, int c) {
	return (a==b)?a:(a==c)?a:(b==c)?b:(a>b)?(a>c)?a:c:(b>c)?b:c;
}
```

The main function consists of the declaration of the variables and the execution of the function. As part of the declaration, they are made symbolic using one of KLEE's functions. This function is the responsible of making a variable appear as any possible value during the execution with KLEE.

```
int main() {
	int a, b, c;
	klee_make_symbolic(&a, sizeof(a), "a");
	klee_make_symbolic(&b, sizeof(b), "b");
	klee_make_symbolic(&c, sizeof(c), "c");
	return example(a, b, c);
}
```

Next thing to do is compile it with LLVM and execute it with KLEE.

```
llvm-gcc --emit-llvm -c -g example.c
klee example.o
```

The output consists of some information from KLEE. It describes where the results are stored and gives a summary of the statistics. For this example only 5 different branches are possible, and the tool has generated a set of values for the symbolic variables in each case.

```
KLEE: output directory = "klee-out-0"

KLEE: done: total instructions = 143
KLEE: done: completed paths = 5
KLEE: done: generated tests = 5
```

For KLEE to work effectively, it needs to have definitions for all the external functions the program may call. In order to properly execute a more elaborated program, the uClibc [And] library, a small C standard library implementation, should be included too. KLEE's developers have modified the uClibc C library implementation for use with the tool. The –libc=uclibc KLEE argument tells KLEE to load that library and link it with the application before it starts execution. This introduces several warnings and a lot more executed instructions, due to the loading of the library. For our purpose now, this is not necessary, but it will be further on. Notice how the warnings refer to C functions implemented not by the C library but by the operating system.

```
klee --libc=uclibc example.o

KLEE: NOTE: Using klee-uclibc : /usr/local/lib/klee-uclibc.bca
KLEE: output directory = "klee-out-1"
KLEE: WARNING: undefined reference to function: __syscall_rt_sigaction
KLEE: WARNING: undefined reference to function: fcntl
KLEE: WARNING: undefined reference to function: fstat
KLEE: WARNING: undefined reference to function: ioctl
KLEE: WARNING: undefined reference to function: open
KLEE: WARNING: undefined reference to function: write
KLEE: WARNING: undefined reference to function: kill (UNSAFE)!
KLEE: WARNING ONCE: calling external: ioctl(0, 21505, 44579648)
KLEE: WARNING ONCE: calling __user_main with extra arguments.

KLEE: done: total instructions = 5559
KLEE: done: completed paths = 5
KLEE: done: generated tests = 5
```

The warnings indicate undefined references to some functions that the C library leaves to be specified by the operating system. This is partially solved by also including a library implementing some of the operating system functions, although a few new functions are still left out. The developers provide a POSIX runtime which is designed to work with KLEE and the uClibc library to provide the majority of operating system facilities used by command line applications – the –posix-runtime argument tells KLEE to link this library in as well. Undefined references will appear in real software for any library not compiled and linked for KLEE. As long as the program does not use this functions, no errors will appear. As was before, the completed paths and generated tests are the same, because the current example does not require any libraries.

```
klee --libc=uclibc --posix-runtime example.o

KLEE: NOTE: Using klee-uclibc : /usr/local/lib/klee-uclibc.bca
KLEE: NOTE: Using model: /usr/local/lib/libkleeRuntimePOSIX.bca
KLEE: output directory = "klee-out-2"
KLEE: WARNING: undefined reference to function: __xstat64
KLEE: WARNING: undefined reference to function: fwrite
KLEE: WARNING: undefined reference to function: lseek64
KLEE: WARNING ONCE: calling external: syscall(16, 0, 21505, 40788240)
KLEE: WARNING ONCE: calling __user_main with extra arguments.
KLEE: WARNING ONCE: calling external: __xstat64(1, 40704144, 40820208)

KLEE: done: total instructions = 6027
KLEE: done: completed paths = 5
KLEE: done: generated tests = 5
```

This is the output for the fist test case results, in which all the variables have the same value. The argument –write-ints is to show the data as integers, instead of bytes in hexadecimal.

```
ktest-tool --write-ints klee-last/test000001.ktest

ktest file : 'klee-last/test000001.ktest'
args       : ['example.o']
num objects: 3
object    0: name: 'a'
object    0: size: 4
object    0: data: 0
object    1: name: 'b'
object    1: size: 4
object    1: data: 0
object    2: name: 'c'
object    2: size: 4
object    2: data: 0
```

All three values are 0. A summary of the other tests results is as follows.

```
test000002                     test000003
a == 0                         a == 0
b == 1                         b == 1
c == 0                         c == 1

test000004                     test000005
a == 1342111743                a == 2
b == 1878982655                b == 0
c == 0                         c == 1
```

Using these values, the paths in the function can be traced. All the possible branches in the small code are taken. We do nothing with it, but in a more elaborated example it could be used to check the correctness of every outcome.

```
return (a==b)?a:(a==c)?a:(b==c)?b:(a>b)?(a>c)?a:c:(b>c)?b:c ;
```

```
Test 1 -> a==b
Test 2 -> a==c
Test 3 -> b==c
Test 4 -> a<b && b>c
Test 5 -> a>b && a>c
```

Apart from the results in the ktest files, KLEE generates some other output files with metadata and statistics about the execution itself. Also, in the event of some concrete errors, per-path files are generated specifying the error found in that path. Per-path files include all the information regarding the executed path and the error found in it. These are the types of errors.

- ptr: Stores or loads of invalid memory locations.
- free: Double or invalid free().
- abort: The program called abort().
- assert: An assertion failed.
- div: A division or modulus by zero was detected.

- user: There is a problem with the input (invalid klee intrinsic calls) or the way KLEE is being used.
- exec: There was a problem which prevented KLEE from executing the program; for example an unknown instruction, a call to an invalid function pointer, or inline assembly.
- model: KLEE was unable to keep full precision and is only exploring parts of the program state. For example, symbolic sizes to malloc are not currently supported, in such cases KLEE will concretize the argument.

These errors can be useful to detect some pointer problems with memory, but beyond that bug detection is left to be implemented by the programmer. Usually programs are silent, most of their operations do not produce any visible output, and that is why bugs stay hidden, because maintainers cannot really see what is going on inside. What KLEE does is execute as much lines in the code as possible, but the code should be modified in advance to make it log all the steps in the program. If every instruction, or chunks of instructions, logs what its doing, after KLEE executes all the possible paths, the logs can be analysed to find irregularities. Comparing the real traces of execution in the program with the intended ones from the developers, you can detect incorrect behaviour.

3.2 OpenSSL

As it has been pointed out before, KLEE has several limitations, including: floating point numbers, assembly code, threads, memory objects of variable size, some jumps and sockets. Any external function should also be compiled and linked following KLEE's restrictions. Several of these limitations affect OpenSSL code, specially the inability to use sockets.

OpenSSL is a cryptographic library, but the TLS we are focusing on is intended for communication. Without sockets, there is no communication. The protocol requires proper initialization and procedure that cannot be simulated on an isolated process without interaction. Trying to execute any part of the TLS library with KLEE leads to inevitable errors. KLEE's errors mean that the path that reached a forbidden operation stops being evaluated. Taking into account that every relevant path in the library uses in some way a communication socket, all the execution is useless.

It may be possible to solve this problem using one of the previously mentioned projects, KleeNET, that models a networking system for KLEE. However, using that system with OpenSSL is out of the scope of this project.

In order to continue with the experiment, the TLS library was studied beginning where the Heartbleed bug was found. The process of adapting the code for KLEE implies removing every function dependant on sockets, or other sources of errors, and substituting it for something that KLEE can work on. For the sockets, this usually means converting any input into constant values or symbolic variables, and every output redirect it to a file or stdout.

The bug is located in the tls1_process_heartbeat, a function used to process the heartbeat requests. It is called by the function ssl3_read_bytes whenever it detects that the packet is of the TLS1_RT_HEARTBEAT type. The ssl3_read_bytes on the other hand is used multiple times by ssl_read when the program wants to read something from the secure socket. These function calls work around the SSL structure that stores all the SSL session details. The function ssl3_read_bytes internally uses ssl3_get_record to actually do the reading of the datagrams. The datagrams sent during communication are stored in SSL3_RECORD. All of these happens after the server and client sides have opened the communication, with handshaking, specifying the encryption method, compression and other details.

Apart from these functions, a few others are called along the way to perform small tasks. At the end all of them return to a function that either uses sockets or uses values in structures that have been updated by such functions.

The only way to execute KLEE with the Heartbleed setup is to reproduce the bug from scratch, without any OpenSSL dependency. This defies the purpose of the experiment, that was to find the bug in OpenSSL, but it is the only way KLEE can be used. Even if not being able to be used in OpenSSL is a failure of the tool, we may continue to see what would happen in case it could be done.

4. Simplification

KLEE's limitations prevent it from being used directly with OpenSSL. However we still want to know what would happen in case it could be executed. That is why a standalone example is used instead, featuring the same Heartbleed bug.

4.1 Buffer over-read example

The secure connections in OpenSSL with all the encryption, compression and different features it have, buried in layers of abstraction, have proved too much to be adapted for KLEE. Instead of executing KLEE in that scenario, a simulated example of the bug is analysed. The goal is to understand how to address the problem of finding the particular kind of bug we are looking for, even if it cannot be done in the original library.

In order to do so, the following program is presented as an alternative version of the Heartbleed bug. The principle is the same, an echo program where the client specifies the returning length of the message and the server does not check it before using it to copy an arbitrary length of its own memory. The bug is located on the server side, and this is the code.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <limits.h>

int main(int argc, char *argv[])
{
        unsigned short length;
        char buf_length[6];
```

```
    char *buf_out;

    char program_memory[USHRT_MAX] = "Lorem ipsum dolor sit
        amet, consectetur adipiscing elit. Cras rhoncus bibendum
        bibendum. Sed porttitor a dui ut porta. Duis et
        ultricies sem. Curabitur neque arcu, mollis at justo sit
        amet, dictum dapibus lorem. Fusce pharetra leo vitae
        euismod vulputate. Morbi nec consectetur tortor. Duis
        eget malesuada dui, vitae eleifend sapien. In commodo
        facilisis hendrerit.";

    int listen_fd, comm_fd;

    struct sockaddr_in servaddr;

    listen_fd = socket(AF_INET, SOCK_STREAM, 0);

    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htons(INADDR_ANY);
    servaddr.sin_port = htons((argc>1)?(unsigned short)
        strtoul(argv[1], NULL, 10) : 7777);

    bind(listen_fd, (struct sockaddr *) &servaddr,
        sizeof(servaddr));

    listen(listen_fd, 10);

    comm_fd = accept(listen_fd, (struct sockaddr*) NULL, NULL);

    read(comm_fd,buf_length,6);
    length = (unsigned short) strtoul(buf_length, NULL, 10);
    read(comm_fd,program_memory,length);
    buf_out = malloc(length);
    memcpy(buf_out, program_memory, length);
    write(comm_fd, buf_out, length);
    free(buf_out);
}
```

This echo program simulates the behaviour of the Heartbleed bug. When executed, it creates a listening socket that waits for a connection. A client application can connect to it, write (as string) the length in bytes of the payload message, and then the message itself. The server allocates enough memory for a response message with the length specified by the client. Then it copies the received data from his memory to the output buffer, and sends it. The copy from the memory program to the output buffer is done using the same length the client sent. When a client sends a message shorter than the length, it receives the difference from the server's memory. For illustration purposes, the program's memory is filled with Lorem ipsum text.

An execution example for the client looks like this.

```
250
Hi server, give me your memories!
Hi server, give me your memories!
tetur adipiscing elit. Cras rhoncus bibendum bibendum. Sed
    porttitor a dui ut porta. Duis et ultricies sem. Curabitur
    neque arcu, mollis at justo sit amet, dictum dapibus lorem.
    Fusce pharetra leo vitae euismod vulpu
```

Now that we have an appropiate example, we compile it with LLVM. There are no compilator errors, but they appear when it is executed using KLEE as interpret.

```
KLEE: NOTE: Using klee-uclibc : /usr/local/lib/klee-uclibc.bca
KLEE: NOTE: Using model: /usr/local/lib/libkleeRuntimePOSIX.bca
KLEE: output directory = "klee-out-2"
KLEE: WARNING ONCE: function "__libc_accept" has inline asm
KLEE: WARNING ONCE: function "bind" has inline asm
KLEE: WARNING ONCE: function "listen" has inline asm
KLEE: WARNING ONCE: function "socket" has inline asm
KLEE: WARNING: undefined reference to function: __xstat64
KLEE: WARNING: undefined reference to function: fwrite
KLEE: WARNING: undefined reference to function: lseek64
KLEE: WARNING ONCE: calling external: syscall(16, 0, 21505,
    42344352)
KLEE: WARNING ONCE: calling __user_main with extra arguments.
KLEE: WARNING ONCE: calling external: __xstat64(1, 42258800,
    42376352)
KLEE: ERROR: /home/klee-uclibc/libc/inet/socketcalls.c:362: inline
    assembly is unsupported
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 1119693
KLEE: done: completed paths = 1
KLEE: done: generated tests = 1
```

The output can be interpreted as: Using uClibc, POSIX runtime, results stored in klee-out-2 directory. Warning, there are four functions related to sockets that cannot be executed due to inline assembly. Five more warnings are related to the POSIX runtime. During the execution, a socket was used producing an error, this execution path is stoped. A lot of instructions, most of them part of uClibc and POSIX initiations, and only one path. In order to keep it simple and short, further outputs will be trimmed of repeated messages.

As we knew, KLEE does not support sockets. The number of paths is just one because there are no symbolic variables declared. The next step is to introduce symbolic variables and get rid of the socket. In this case, the variables that must be symbolic are the ones read by the socket, so substituting the socket reads by symbolic inputs will do the trick. Instead of writing to the socket, stdout is used.

```
    klee_make_symbolic(buf_length, 6, "buf_length");
    length = (unsigned short) strtoul(buf_length, NULL, 10);
    klee_make_symbolic(program_memory, length, "program_memory");
    buf_out = malloc(length);
    memcpy(buf_out, program_memory, length);
    write(1,buf_out, length);
    free(buf_out);
```

The new code gets three test cases, including errors. However, the errors found are not related to the bug we are looking for. The first error is that it cannot make only part of an array of variables symbolic. The other two are out of bounds pointers in strtoul when trying to convert the string to an integer.

```
KLEE: ERROR: /home/Project/echo1.c:21: wrong size given to
    klee_make_symbolic[_name]
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee-uclibc/libc/stdlib/stdlib.c:526: memory
    error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee-uclibc/libc/stdlib/stdlib.c:559: memory
    error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 1159159
KLEE: done: completed paths = 721
KLEE: done: generated tests = 3
```

The first error is solved here. For the second symbolic variable in the code, the size specified does not include the whole array. KLEE cannot handle that and gives an error. To solve this, we add a new function (originally written by Milen Dzhumerov in the klee-dev mailing list).

```
void klee_make_symbolic_range(void* addr, size_t offset, size_t
    nbytes, const char* name) {
       assert(addr != NULL);
       assert(name != NULL);

       if(nbytes == 0)
              return;
       void* start = addr + offset;

       void* symbolic_data = malloc(nbytes);
       klee_make_symbolic(symbolic_data, nbytes, name);
       memcpy(start, symbolic_data, nbytes);
       free(symbolic_data);
}
```

The second error case of the previous three is generated for strtoul, some values make the conversion from string to long to get an error. These values are showed below. Although this is really a bug, it is not the one we are looking for. If nothing else is specified, the symbolic values can be anything, which sometimes locates bugs like this. In a real program, the input string to strtoul should be restricted to a valid number. This is the output file KLEE generates where it is somehow explained that strtoul produced an out of bound pointer error in its execution.

```
Error: memory error: out of bound pointer
File: /home/klee-uclibc/libc/stdlib/stdlib.c
Line: 526
assembly.ll line: 2624
Stack:
       #0 00002624 in _stdlib_strto_l (str=47899040, endptr=0,
           base=10, sflag=0) at
           /home/klee-uclibc/libc/stdlib/stdlib.c:526
       #1 00001845 in strtoul (str=47899040, endptr=0, base=10) at
           /home/klee-uclibc/libc/stdlib/stdlib.c:386
       #2 00000219 in __user_main (argc=1, argv=39454320) at
           /home/Project/echo1.c:20
       #3 00001019 in __uClibc_main (main=37259040, argc=1,
           argv=39454320, app_init=0, app_fini=0, rtld_fini=0,
           stack_end=0) at
           /home/klee-uclibc/libc/misc/internals/__uClibc_main.c:401
       #4 00000000 in main (=1, =39454320)
Info:
       address: 47899046
       next: object at 47900640 of size 8
              MO724273[8] allocated at _stdlib_strto_l(): %0 =
                  alloca i64
       prev: object at 47899040 of size 6
```

```
            MO3344[6] allocated at __user_main(): %buf_length =
                alloca [6 x i8]
```

The output file shows the value that was passed to strtoul to get an error. In this case, the value passed to strtoul is the byte x0C repeated six times. This character in ASCII means form feed, or page break. Obviously not a valid number.

```
object    1: name: 'buf_length'
object    1: size: 6
object    1: data: '\x0c\x0c\x0c\x0c\x0c\x0c'
```

The third test case is the same as the second one, just reached in a different place. The stdlib library from uClibc gets the error in a different line.

```
Line: 559
assembly.ll line: 2728
```

Moving on. The code was modified to allow the second variable, the program's memory, to be partially symbolic. That is where the client is writing to, the beginning of the program's memory, the place where the sender's message enters the program. However, that produces a new error related to one of KLEE's limitations. No memory allocation can be symbolic. The length specified in klee_make_symbolic() cannot be symbolic, otherwise it is concretized before going on. It takes a concrete value, stops being symbolic and thus its execution is no longer useful for out purpose because it will only follow one execution path. This is the error file generated for these type of error.

```
cat klee-last/test000003.model.err

Error: concretized symbolic size
File: /home/Project/echo1.c
Line: 21
assembly.ll line: 240
Stack:
       #0 00000240 in klee_make_symbolic_range (addr=46834912,
           offset=0, nbytes, name=46499216) at
           /home/Project/echo1.c:21
       #1 00000300 in __user_main (argc=1, argv=40777312) at
           /home/Project/echo1.c:37
       #2 00001084 in __uClibc_main (main=36052208, argc=1,
           argv=40777312, app_init=0, app_fini=0, rtld_fini=0,
           stack_end=0) at
           /home/klee-uclibc/libc/misc/internals/__uClibc_main.c:401
       #3 00000000 in main (=1, =40777312)
```

```
Info:
  size expr: (ZExt w64 (Extract w16 0 (Sub w64 0
                                        (ZExt w64 (Add w8 208 (Read
                                            w8 1 buf_length))))))
  concretization : 65533
  unbound example: 65535
```

Again we have to modify the code to simplify it a bit more. Neither the klee_make_symbolic_range nor the malloc accept symbolic variables as parameters. Instead of leaving it for KLEE concretize the values, we can choose constant ones that comply with the rest of the bug.

Remember that the symbolic values represent both user inputs, the length to be replied and the message sent. The bug we are hunting happens when the length is greater than the real size of the message. We are forced by KLEE's limitations to concretize the size of the original message and the size of the program's response. This is bad, because it is too narrow and focused. We were taking a broad perspective, trying to reach the bug without centering on it. Now we have to specify a value for the response bigger than a value for the message in order for the bug to happen, thus assuming that the developer already knows where the bug is.

```
        klee_make_symbolic(buf_length, 6, "buf_length");
        length = (unsigned short) strtoul(buf_length, NULL, 10);

        klee_make_symbolic_range(program_memory, 0, 400,
            "program_memory");
        buf_out = malloc(500);
        memcpy(buf_out, program_memory, length);
        write(1,buf_out, length);
```

In order to keep it simple, we can also remove the strtoul call and just make the length value symbolic, instead of converting it from a symbolic string.

```
        klee_make_symbolic(&length, sizeof(short), "length");

        klee_make_symbolic_range(program_memory, 0, 400,
            "program_memory");
        buf_out = malloc(500);
        memcpy(buf_out, program_memory, length);
        write(1,buf_out, length);
```

Using that code, KLEE finds another error case. The length variable is limited to 16 bits, but in this small example, the total amount of memory allocated to the program is so small that moving that much memory with memcpy() will go out of the memory space of the program. If memcpy() uses a length value too high, it gets a memory error: out of bound pointer, because it is bigger than the actual memory size the program owns in the system. In Heartbleed this could also happen, but the OpenSSL library allocates a lot of memory to work, so even a big length value does not go out of the program's memory, that is why it was not detected earlier. But the Heartbleed bug itself is not an out of bound pointer. We are looking for a length value big enough to get more memory than it is supposed to, but without going out of the program's memory, in which case the operating system would most likely kill the process.

The execution time is also too high, which was another of KLEE's problems, there are too many possibilities. KLEE is designed to keep running even for exponential growths that will make the execution almost infinite. The execution can be halted to stop this and get only part of the results. However for our purpose, we will trim it a bit more. Using klee_assume, the possible values for length are limited between 20 and 300. Using 300 is arbitrary, any number lower than the memory (400) could be used to illustrate the example.

```
klee_make_symbolic(&length, sizeof(short), "length");
        klee_assume((length < 300) & (length > 20));

klee_make_symbolic_range(program_memory, 0, 400,
    "program_memory");
buf_out = malloc(300);
memcpy(buf_out, program_memory, length);
```

Finally, the execution is concrete enough to get a good view of the bug. However the values in the message created by klee_make_symbolic_range are not limited to printable ones, so they appear as unknown characters. For the sake of visibility, and because its length was already made constant, we can substitute this input for a constant string. This string represents the message payload sent by the client, and it will always be "HI SERVER, GIVE ME YOUR MEMORIES!", which is 33 bytes long.

The output is very long, but it can be divided in three sections. The first one is when the length is actually smaller than the the real length of the message, in which case the string is not shown complete.

```
HI SERVER, GIVE ME YOHI SERVER, GIVE ME YOUHI SERVER, GIVE ME
    YOURHI SERVER, GIVE ME YOUR HI SERVER, GIVE ME YOUR MHI SERVER,
    GIVE ME YOUR MEHI SERVER, GIVE ME YOUR MEM
```

The second possibility is actually when the length is the real one, 33. It is the only time the program behaves correctly and the only one it should print in a program without the bug. Here is the chunk corresponding to three iterations, with length 32, 33 and 34.

```
HI SERVER, GIVE ME YOUR MEMORIESHI SERVER, GIVE ME YOUR MEMORIES!HI
    SERVER, GIVE ME YOUR MEMORIES!c
```

The last one, and the biggest one, happens whenever the length is greater than the message. KLEE iterates through all the possibilities one by one, leaking one byte more on each iteration.

```
HI SERVER, GIVE ME YOUR MEMORIES!cHI SERVER, GIVE ME YOUR
    MEMORIES!ctHI SERVER, GIVE ME YOUR MEMORIES!cteHI SERVER, GIVE
    ME YOUR MEMORIES!ctetHI SERVER, GIVE ME YOUR MEMORIES!ctetuHI
    SERVER, GIVE ME YOUR MEMORIES!cteturHI SERVER, GIVE ME YOUR
    MEMORIES!ctetur HI SERVER, GIVE ME YOUR MEMORIES!ctetur adHI
    SERVER, GIVE ME YOUR MEMORIES!ctetur aHI SERVER, GIVE ME YOUR
    MEMORIES!ctetur adiHI SERVER, GIVE ME YOUR MEMORIES!ctetur adip
```

The end of the output includes the program's memory up to the imposed 300 limit, and the statistics of the execution.

```
HI SERVER, GIVE ME YOUR MEMORIES!ctetur adipiscing elit. Cras
    rhoncus bibendum bibendum. Sed porttitor a dui ut porta. Duis
    et ultricies sem. Curabitur neque arcu, mollis at justo sit
    amet, dictum dapibus lorem. Fusce pharetra leo vitae euismod
    vulputate. Morbi nec consectetur tortor. Duis eget mal
KLEE: done: total instructions = 1163556
KLEE: done: completed paths = 279
KLEE: done: generated tests = 279
```

Now compare that with the same execution when the bug in the code is fixed, that is, when the symbolic length is compared to the real length of the message. One case for printing, another for discarding.

```
    char program_memory[USHRT_MAX] = "HI SERVER, GIVE ME YOUR
        MEMORIES!ctetur adipiscing elit. Cras rhoncus bibendum
        bibendum. Sed porttitor a dui ut porta. Duis et ultricies
        sem. Curabitur neque arcu, mollis at justo sit amet, dictum
        dapibus lorem. Fusce pharetra leo vitae euismod vulputate.
        Morbi nec consectetur tortor. Duis eget malesuada dui, vitae
        eleifend sapien. In commodo facilisis hendrerit.";

    int real_length = 33;
    klee_make_symbolic(&length, sizeof(short), "length");
    klee_assume((length < 300) & (length > 20));

    if(length != real_length) return 0;
    buf_out = malloc(300);
    memcpy(buf_out, program_memory, length);
    write(1,buf_out, length);
    free(buf_out);
```

```
HI SERVER, GIVE ME YOUR MEMORIES!
KLEE: done: total instructions = 1120475
KLEE: done: completed paths = 2
KLEE: done: generated tests = 2
```

The process included a lot of problems, first with KLEE's limitations, then with specifics for the test case. However once it is done, anyone can see that a correct execution is kept small, while a execution without proper validation of the inputs leads to a huge number of possible outcomes.

It cannot be said that KLEE found Heartbleed, but after some work it found a bug of similar caracteristics.

5. Conclusions

The results of experimenting in our little example show that KLEE is not yet ready to be the perfect solution. The tool has severe limitations like the inability to handle sockets, floating point numbers, inline assembly, external calls and other functions present in nowadays software. For the purpose of this project, the main problems were sockets and external calls.

It is also a limitation not being able to use variable memory allocations. Real software like OpenSSL often use external inputs to allocate and work with memory. Without the ability simulate it, the developer must work around it, modifying the behaviour of the code and potentially missing or introducing new bugs.

The previous limitations make the task of using KLEE more complicated. The developer not only needs to use the tool, he has to modify and limit his code in order to do so. When dealing with complex interaction with other processes, using sockets or other means of exchange of information, adapting the scenario to KLEE becomes impossible.

Even when everything works after all the modifications in the code, the automatic bug detection is very limited. Out of bounds pointers and double free of memory are the only bugs that KLEE can detect by default. Other bugs must be searched by manually exposing all the details in the code and analysing its execution afterwards.

In the example, Heartbleed, the bug acts in a way that cannot be detected immediately. Even if it does something that it was not supposed to do, KLEE does not know it. Maybe the developer wanted a program that did exactly that, echoing back part of its own memory.

The only way in which it can be spotted is with a human debugger that knows what is wrong and what it is not. There is where KLEE is useful. As shown in the example, the output to stdout made the bug visible. KLEE helped not by detecting the bug, but by exploring every possible path and printing everything. A further analysis of that output, possibly by a human or maybe with another program, might find the bug.

That is the way some of the related works used KLEE. Using logs to write down all the steps in the code, executing KLEE to get every possible path, and then analysing all the logs to find inconsistencies.

In conclusion, KLEE requires a lot of work in real software in order to be executed. The code must be heavily modified to meet the limitations. It also must be applied in small parts of the software at a time, narrowing how much paths can be explored. However, once all that is overcome, the output of every possible path can be analysed to find bugs in the software, making KLEE a useful tool to be considered for future work.

6. Further work

Despite its current limitations, the idea behind the tool is really interesting. Being able to explore all the paths in your software makes it easier to find bugs and avoid vulnerabilities in your system. With more time to work with it, KLEE could be extended to properly operate on more software.

The limitations should be overcome, specially the networking one. Maybe the KleeNet project achieved this, so further work should explore that project.

The interface could also be improved, maybe with an html representation of the output with all the explored paths and debug information of each one.

The process of adapting any software to KLEE could also be automated, using some predefined modifications that remove problematic functions and introduce some well tested and equivalent alternatives that work well with KLEE.

Another thing worth of more study would be using some high level specification language to define the proper behaviour of a program. A language that could work with the traces made by KLEE to determine if everything is correct. There are several specification languages like CASL, VDM or Z notation [Spe].

Exploring other uses for KLEE beyond bug detection is also possible, like the usage made in Howard: A Dynamic Excavator for Reverse Engineering Data Structures [SSB11]. Howard uses dynamic analysis of memory access patterns to reveal the layout of data structures. To do that it needs to execute all the code, because any data structure not used during execution cannot be found. It uses KLEE only as a code coverage tool to execute everything and independently find the data structures in the executed code. Any problem that could make use of a code coverage tool may benefit from taking a look at KLEE.

Finally, a good area in which using KLEE is with microcontroller's code, because it is always more simple and with less requirements than fully featured software. Microcontrollers do not use IP sockets, do not have big external libraries, do not have an operating system to interact with and the code is kept small due to memory limitations. The tool should be tested in this environment to see if it works better or other problems arise.

Bibliography

[And] Erik Andersen.
uClibc.
`http://www.uclibc.org`.
last visited 27/07/2014.

[BoK] BoKS ServerControl.
`http://www.foxt.com/product/`.
last visited 20/06/2014.

[CCK11] Peter Collingbourne, Cristian Cadar, and Paul H. J. Kelly.
Symbolic testing of opencl code.
In *Haifa Verification Conference (HVC 2011)*, 1 2011.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler.
Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.
In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[Chra] Chris Lattner.
`http://blog.llvm.org/2011/05/what-every-c-programmer-should-know_14.html`.
last visited 20/06/2014.

[Chrb] Chris Lattner and Vikram Adve.
The LLVM Compiler Infrastructure.
`http://llvm.org`.
last visited 20/06/2014.

[Coda] Codenomicon.
`http://www.codenomicon.com`.
last visited 27/07/2014.

[Codb] Codenomicon.
Heartbleed bug.
`http://heartbleed.com`.
last visited 20/06/2014.

[Cri] Cristian Cadar and Daniel Dunbar and Dawson Engler.
KLEE.
http://klee.github.io/klee/.
last visited 20/06/2014.

[Dar10] Darrell Bethea and Robert A. Cochran and Michael K. Reiter.
Server-side Verification of Client Behavior in Online Games.
http://www.isoc.org/isoc/conferences/ndss/10/pdf/01.pdf, 2010.
last visited 03/04/2014.

[FOX] FOX Technologies AB.
http://www.foxt.com.
last visited 20/06/2014.

[Jona] Jon Brodkin.
OpenSSL code beyond repair, claims creator of LibreSSL fork.
http://arstechnica.com/information-technology/2014/04/openssl-code-beyond-repair-claims-creator-of-libressl-fork/.
last visited 20/06/2014.

[Jonb] Jon Brodkin.
Tech giants, chastened by Heartbleed, finally agree to fund OpenSSL.
http://arstechnica.com/information-technology/2014/04/tech-giants-chastened-by-heartbleed-finally-agree-to-fund-openssl/
last visited 20/06/2014.

[KCC10] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea.
Testing closed-source binary device drivers with ddt.
In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association.

[Kel] Leo Kelion.
US government warns of Heartbleed bug danger.
http://www.bbc.com/news/technology-26985818.
last visited 20/06/2014.

[LA04] Chris Lattner and Vikram Adve.
LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.
In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[Lin] Linda Rosencrance.
OpenSSL patches 7 security flaws, 2 critical.

http://www.cio-today.com/article/index.php?story_id=111003TUMQQI.
last visited 20/06/2014.

[LXC13] Li Lei, Fei Xie, and Kai Cong.
Post-silicon conformance checking with virtual prototypes.
In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 29:1–29:6, New York, NY, USA, 2013. ACM.

[Net] Netcraft.
http://www.netcraft.com.
last visited 20/06/2014.

[Opea] OpenBSD team.
OpenBSD.
http://www.openbsd.org.
last visited 20/06/2014.

[Opeb] OpenSSL team.
List of OpenSSL fixed vulnerabilities.
https://www.openssl.org/news/vulnerabilities.html.
last visited 20/06/2014.

[Paua] Paul Mutton.
Half a million widely trusted websites vulnerable to Heartbleed bug.
http://news.netcraft.com/archives/2014/04/08/half-a-million-widely-trusted-websites-vulnerable-to-heartbleed-bug.html.
last visited 20/06/2014.

[Paub] Paul Mutton.
Netcraft releases Heartbleed indicator for Chrome, Firefox, and Opera.
http://news.netcraft.com/archives/2014/04/17/netcraft-releases-heartbleed-indicator-for-chrome-firefox-and-opera.html.
last visited 20/06/2014.

[SLA+10] Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle.
Kleenet: Discovering insidious interaction bugs in wireless sensor networks before deployment.
In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '10, pages 186–196, New York, NY, USA, 2010. ACM.

[Spe] Specification language.
http://en.wikipedia.org/wiki/Specification_language.
last visited 28/07/2014.

[SSB11] Asia Slowinska, Traian Stancescu, and Herbert Bos.
Howard: A dynamic excavator for reverse engineering data structures.
In *NDSS*. The Internet Society, 2011.

[SToAS$^+$12] R. Seggelmann, M. Tuexen, Muenster Univ. of Appl. Sciences, M. Williams, and GWhiz Arts & Sciences.
Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension.
RFC 6520, RFC Editor, February 2012.

[The] The OpenSSL Core and Development Team.
OpenSSL.
https://www.openssl.org.
last visited 20/06/2014.

[Vij] Vijay Ganesh and Trevor Hansen.
STP.
https://github.com/stp/stp.
last visited 25/07/2014.

[Vit10] Vitaly Chipounov and George Candea.
Reverse Engineering of Binary Device Drivers with RevNIC.
http://dslab.epfl.ch/pubs/revnic.pdf, 2010.
last visited 03/04/2014.

[W.A98] Andrew W.Appel.
Modern Compiler Implementation in ML.
Cambridge Univ. Press, 1998.

[XPi] XPilot.
http://www.xpilot.org.
last visited 20/06/2014.