Information leakage analysis of complex C code and its application to OpenSSL

Pasquale Malacaria[1], Michael Tautchning[1], and Dino DiStefano[1]

School of Electronic Engineering and Computer Science,
Queen Mary University of London, UK
{p.malacaria,michael.tautschnig,d.distefano}@qmul.ac.uk,

Abstract. The worldwide attention generated by the Heartbleed bug has demonstrated even to the general public the potential devastating consequences of information leaks.
While substantial academic work has been done in the past on information leaks, these works have so far not satisfactorily addressed the challenges of automated analysis of real-world complex C code. On the other hand, effective working solutions rely on ad-hoc principles that have little or no theoretical justification.
The foremost contribution of this paper is to bridge this chasm between advanced theoretical work and concrete practical needs of programmers developing real world software. We present an analysis, based on clear security principles and verification tools, which is largely automatic and effective in detecting information leaks in complex C code running everyday on millions of systems worldwide.

1 Introduction

The OpenSSL Heartbleed vulnerability (CVE-2014-0160)[1] has attracted international attention both from media and security experts. It is difficult to imagine a more serious security flaw: devastating (clear-text passwords are leaked), widespread (running on millions of systems), untraceable, and repeatable while leaking up to 64 KB of memory at a time.

Automated security analysis of code have so far proven to be of limited help: Heartbleed seemingly demonstrated the limitations of current static analysis tools for this kind of leaks. As noted by Kupsch and Miller [14], static analysis tools struggle detecting Heartbleed due to the use of pointers, and the complexity of the execution path from the buffer allocation to its misuse.

Static analyses capable of scrutinising large code bases are effective at detecting bugs that may bring undefined behaviour (e.g., a crash), but they are less effective at detecting deep intricate bugs which represent functional misbehaviour in code of any size.

The code analysis technique used in this paper, while being static in the sense of being applied at compile time and considering all (bounded) execution paths,

[1] cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160 and heartbleed.com

is an ideal complement to classical static analysis. Our analysis aims at detecting deep, intricate confidentiality violations. While our methodology allows for abstractions and sometimes may need them, it is largely a precise analysis down to the bit level. As such, all aspects of the code and the security requirements are translated into logic formulae and then checked by SAT or SMT solvers. Our technique would be a valuable tool for both developers and for code reviewers. The manual effort required is a labelling of confidential information and to write appropriate drivers. In this context, it is worth noting that Heartbleed was originally discovered as part of a code review, described as "laborious auditing of OpenSSL" [16].

Our methodology is not about detecting undefined behaviour in the code, such as generic memory errors, but rather detecting confidentiality violations.

The principles underpinning this work go back to the fundamental definition of security. To the best of our knowledge, however, it was unknown how to implement such principles for large and complex C code. As such, the first and foremost contribution of this work is in enabling such real-world, complex security analysis.

Related work There are several commercial static analysers for C such as Grammatech's CodeSonar [9], Coverity [7], Klokwork [12], HP/Fortify [11]. None of these tools detected Heartbleed ahead-of-time. Some of the vendors of these tools are now extending their heuristics for being able to catch similar bugs [1, 4]. Their approach is based on the general idea of *taint analysis.* All these tools are very effective at detecting implementation bugs (which may or may not necessarily be security vulnerabilities) violating certain patterns. OpenSSL code is extremely complex; it includes multiple levels of indirection and other issues that can easily prevent these tools from finding vulnerabilities. Heartbleed is not an exception. Most importantly, these tools are not confidentiality checkers and so may not be able to find leaks not originating from undefined behaviour. Our technique instead is aimed at detecting subtle information leaks.

Dynamic analysis used in tools like Valgrind [22] is very effective in finding code defects and improving the security of code. While extremely useful, dynamic analysis techniques can only check for a limited number of inputs and, therefore, do not provide the same strong security guarantees as our approach does. Similar to dynamic analyses symbolic execution tools are very scalable and our approach can be implemented in KLEE and similarly in other such platforms.

There is a large body of literature on non-interference [17, 20] with related type systems, abstract domains, and data-flow and dynamic analysis. As already mentioned these approaches have had limited success on complex C code. Our work builds on the security literature of self-composition and its implementation [2, 21]. Previously, none of these works was able to deal with complex C code. CBMC has been used to implement self-composition also in [10]. Comparing their work with the proposed methodology, they neither use quantifiers, hence are limited to bounded analysis. Also they didn't attack the engineering challenges of an automated analysis of a large code basis like OpenSSL.

2 Background

Our confidentiality analysis is based on the definition of non-interference [8]. Informally:

> A program is *non-interfering* (i.e., doesn't leak confidential information) if and only if two runs of the program that only differ in some confidential value do not yield different behaviours that can be observed by an attacker.

In other words a non-leaking program behaves, from the point of view of an attacker, as a constant function once its non-confidential arguments are fixed. More formally noting $\langle P, \mu \rangle \downarrow \nu$ for "the program P starting from memory configuration (contents) μ terminates with a resulting memory ν" then (termination insensitive) non-interference is defined as: for all memory configurations $\mu_1, \mu_2, \nu_1, \nu_2$:

$$[\langle P, \mu_1 \rangle \downarrow \nu_1 \wedge \langle P, \mu_2 \rangle \downarrow \nu_2 \wedge \mu_1 =_L \mu_2] \rightarrow \nu_1 =_L \nu_2$$

where $\mu_1 =_L \mu_2$ means the memory configurations agree on the non-confidential values (also called the public values or *low* values; public values are assumed to be observable).

We refer the interested reader to the literature [17] for a more extensive background on non-interference and confidentiality.

2.1 An Introductory Example

To illustrate non-interference, consider an authentication system testing whether a user-provided string is a valid password:

```
  int authenticate(int passwd, int guess)
{
  int authenticated;
  if(passwd==guess)
  authenticated=1;
  else authenticated=0;
  return authentic;
}
```

The authenticate function above is not secure because we can find two different confidential values for the variable `passwd` resulting in two observables by an attacker. The first one is the value of `passwd` being equal to the value of `guess`. The second can be chosen as any different value. In this case the program will return two different values for `authenticated`, which is observable by an attacker.

More specifically, by observing `authenticated==1` the attacker will know the password is the value of `guess`, and by observing `authenticated==0` the attacker will know the password is not the value of `guess`. In both cases the attacker will learn something about the password, hence some information is being leaked.

Handling randomness The classic definition of confidentiality fails to account for programs the behaviour of which depends on sources of randomness. Consider the following variation of the above program:

if(random_value) authenticated=1; **else** authenticated=0;

This program would be deemed non secure following the definition of non-interference. Assuming `random_value` doesn't use any confidential values in its computation then the above program is, however, secure. One way to understanding this in the context of non-interference is to think that a `random_value` in a deterministic systems is in fact the result of a deterministic function on some possibly difficult-to-guess non-confidential inputs, e.g. the seed used in the function generating random numbers in standard programming languages. A full discussion of this topic is beyond the scope of this work. For the purposes of our analysis hence when allowing random values we need to check whether the source of randomness is non confidential and if that is the case then not count that as a security violation. If fact we will deal with random values using CBMC in the same way as we deal with missing code, which is explained in Sec. 4.4. Handling randomness is crucial when analysing some OpenSSL functions, such as `dtls1_heartbeat` or `tls1_heartbeat` (see Sec. 4.5). To correctly label the source of randomness is usually the task of the developer.

3 Confidentiality Analysis using CBMC

The workflow of the analysis implemented using CBMC is summarised in Fig. 1. We first expand on how the driver is defined, its relation to non-interference and self-composition and how C code is handled by CBMC. We explain pre-processing in Sec. 4.2, i.e., how to prepare the source code for the analysis, and in Sections 4.3 and 4.4 we will discuss how to deal with missing code and unbounded analysis using quantifiers.

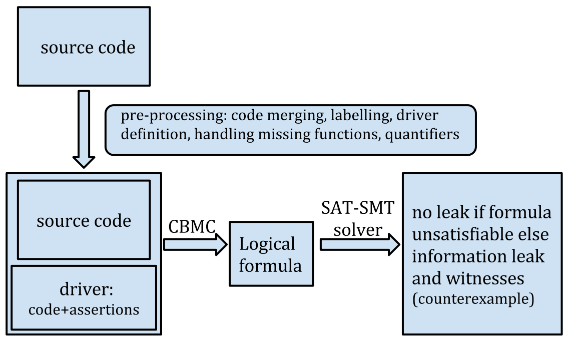
source code
pre-processing: code merging, labelling, driver definition, handling missing functions, quantifiers
source code
driver:
code+assertions
CBMC
Logical formula
SAT-SMT solver
no leak if formula unsatisfiable else information leak and witnesses
(counterexample)

Fig. 1. Workflow of the analysis using CBMC

To start with let's explain how we check non-interference using the bounded model checker CBMC [5, 13]. To illustrate the use of CBMC in this context, let us consider the program of Fig. 2, taken from [6]. The first listing on the left contains the program with an assertion describing the desired postcondition. That is, for all possible executions of the program it holds that $x \leq 3$ at the end. As first step, CBMC transforms the program into Static Single Assignment (SSA) form, which introduces the new variables `x_1, x_2, x_3` corresponding to the different definitions of the variable `x` in the program, and similarly `y_1` for variable `y`. The code in SSA form induces a system of equations which is then translated to a propositional formula $\mathcal{C}$ the atoms of which are bit vector equations. $\mathcal{C}$ represents the program as equation system and a model of $\mathcal{C}$ can be interpreted as an input and its execution trace. Finally the assertion is translated to the formula $\mathcal{P}$.

```
x=2; y=1;
if(x!=1)
{
  x=2;
  if(y) x++;
}
assert(x<=3);
```

```
x_1=2; y_1=1;
if(x_1!=1)
{
  x_2=2;
  if(y_1!=0) x_3=x_2+1;
}
assert(x_3<=3);
```

$$\mathcal{C} := x_1 = 2 \wedge y_1 = 1 \wedge$$
$$x_2 = ((x_1 \neq 1)?2 : x_1) \wedge$$
$$x_3 = ((x_1 \neq 1 \wedge y_1 \neq 0)?x_2 + 1 : x_2)$$
$$\mathcal{P} := x_3 \leq 3$$

Fig. 2. Example of renaming and transformation in CBMC

Following the rules of Hoare Logic, the postcondition $\mathcal{P} \equiv x_3 \leq 3$ holds if and only if $\mathcal{C} \Rightarrow \mathcal{P}$ is valid. Equivalently, the original assertion is valid in the original program if only if the propositional formula $\mathcal{C} \wedge \neg\mathcal{P}$ is unsatisfiable.

To see that the above statement is true reason as follows: if $\mathcal{C} \wedge \neg\mathcal{P}$ is satisfiable then the satisfying assignment will provide a counterexample for the property $\mathcal{P}$, i.e., a trace showing why the program doesn't satisfy $\mathcal{P}$. If, however, $\mathcal{C} \wedge \neg\mathcal{P}$ is unsatisfiable then the property $\mathcal{P}$ holds for all execution traces.

CBMC is a bounded model checker, hence only a bounded version of a program, where the loops are unwound up to a user-defined bound, can be analysed. Consequently it is first and foremost a bug-finding approach, unless the program under scrutiny only exhibits bounded loops or bounded recursion.

While unbounded verification is thus beyond the scope of CBMC, the user has options that may, in certain cases, provide unbounded verification results (i.e., proofs of correctness): an example is mentioned in Sec. 4.3 where we replace loops with universally quantified expressions.

Self-composition The definition of non-interference is a semantic one. A translation of this definition to verification terms, called *self-composition*, has been introduced in [2, 21]. In self-composition we consider a program P and a copy P' of P. The copy P' consists of P with all variables renamed (public variables

$\overrightarrow{x}$ renamed as $\overrightarrow{x'}$). Let $\uplus$ be disjoint union. Then non-interference is defined as: for all memory configurations μ, μ', ν, ν':

$$[\langle P; P', \mu \uplus \mu' \rangle \downarrow \nu \uplus \nu' \wedge \mu' =_L \mu[\overrightarrow{x'} := \overrightarrow{x}]] \rightarrow \nu' =_L \nu[\overrightarrow{x'} := \overrightarrow{x}]$$

In words: the program $P; P'$, i.e., the sequential composition of P and P', starting from the memory $\mu \uplus \mu'$ (where μ' is the same memory as μ on the public variables $\overrightarrow{x}$, except for renaming of $\overrightarrow{x}$ to $\overrightarrow{x'}$) will terminate resulting in memory $\nu \uplus \nu'$ (where ν' is the same memory as ν on the public variables $\overrightarrow{x}$, except for renaming of $\overrightarrow{x}$ to $\overrightarrow{x'}$).

Our implementation of self-composition using CBMC follows the approach in [10]. Here we only give an intuition about the approach and refer the interested reader to the literature for a more formal definition and relationship between self-composition and non-interference [2, 21, 10]. Recall that by definition of non-interference to find a violation of confidentiality we need to find two runs of the function under analysis that only differ in some confidential value and result in two different observables. To implement this using CBMC we add a driver to the program where we assert that *any two runs of the function which differ on only the confidential values will result in the same observable.* A violation to this assertion (i.e., a counterexample) will hence be an assignment describing two confidential values for which the function will return two different observables.

Going back to the simple password program in Sec. 2.1, its security analysis using CBMC is realised using the following code:

```
int authenticate(int passwd, int guess)
{
  int authenticated;
  if(passwd==guess)
  authenticated=1;
  else authenticated=0;
  return authentic;
}
```

```
void driver()
{
  int pwd1, pwd2, guess;
  int res1=authenticate(pwd1, guess);
  int res2=authenticate(pwd2, guess);
  assert(res1==res2);
}
```

We have inserted a driver method with the declaration of three variables of type `int`. These variables will be used as arguments to the function `authenticate` in the two calls and finally an assertion is made about the equality of the results of the calls. CBMC will translate the above code into a formula and will look for an assignment satisfying that formula. As the variables are not initialised their values will be determined by the SAT solver. By running CBMC on the above code we will get a counterexample, and thus values v, v', u for `pwd1`, `pwd2`, `guess`, respectively, have been found by the SAT solver. As those result in the assertion to fail, this means that the program is leaking confidential information. In the terminology of self-composition

$$\mu = \{\texttt{pwd1} \mapsto v,\ \texttt{guess} \mapsto u\},\ \mu' = \{\texttt{pwd2} \mapsto v',\ \texttt{guess} \mapsto u\}$$

and the renaming[2] of μ to μ' is

$$\{\{\texttt{pwd1} \mapsto v\} := \{\texttt{pwd2} \mapsto v'\},\ \{\texttt{guess} \mapsto \texttt{u}\} := \{\texttt{guess} \mapsto \texttt{u}\}\}.$$

To sum up there are three key ingredients to identify and label when performing a non-interference analysis when using a model checker like CBMC:

1. *Confidential inputs*: the secret we don't want the code to leak (in the above example, the values of `password` in function `authenticate`).
2. *Public inputs*: the inputs that do not contain confidential information (in the above example the argument `guess`).
3. *Observables*: what we assume an attacker can observe when the function is run (in the above example is the return value of function `authenticate`).

4 Analysis of OpenSSL

4.1 Labelling and drivers for OpenSSL

A key aspect of the analysis is the labelling of confidential, public, and observable data. This step cannot be fully automated because it is easy to imagine how the same code may be used for different purposes and hence the meaning of confidential, public, and observable data may be application dependent.

We assume that this process of labelling is in general a simple task for the code developer (and the code reviewer): by writing the code they should know easily what the confidential, non confidential, and observable components in the code are.

Of course the labelling is more challenging for a third party not familiar with the code and, in that case, may require some non-trivial reverse engineering.

In the case of OpenSSL our labelling is determined by reverse engineering what confidential, public and observable are in the functions where Heartbleed originated (i.e. functions: `dtls1_process_heartbeat` and `tls1_process_heartbeat`). Once this labelling is determined we can proceed to analyse all of OpenSSL for leaks from similar inputs to similar observables. The labelling is the following:

1. *Confidential data*: this is the process memory. It is confidential because it holds confidential data, such as passwords or private keys [23]. Notice that this is not an input to a function.
2. *Public data*: these are the `ssl_st` structures containing the payload from the sender or any argument that are provided by the user as arguments to OpenSSL functions. The attacker can control these inputs, which is how Heartbleed is triggered. In security jargon we are considering an active attacker because the attacker can control the public inputs.

[2] Here we map `guess` on the same name whereas we should use different names; it is easy to see this is harmless in this context.

```
void driver(){
        declare a_1, ..., a_n, b_1, ..., b_n;
        a_1 = ...;  // optional initialisation argument a_1
         :
         :
        a_n = ...;  // optional initialisation argument a_n
        b_1 = ...;  // optional initialisation argument b_1
         :
         :
        b_n = ...;  // optional initialisation argument b_n
        assert(observable(f(a_1, ..., a_n))==observable(f(b_1, ..., b_n)));
}
```

Fig. 3. Driver template for checking function `f(i1, ..., in)`.

3. *Observables*: this is the structure used for communicating between the client and server. They use (part of) the structures of type `ssl_st` for communication and the medium is (the function pointer) `msg_callback`. The third and fourth arguments of `msg_callback` consists of the data buffer of communication and its length. We hence select the third and fourth arguments of `msg_callback` as the observables.

We stress that, while this labelling originated from the Heartbleed bug functions, it is not specific to the Heartbleed bug: labelling the process memory as confidential is natural and general because the process memory, no matter what OpenSSL function we consider, contains data like passwords. Labelling `msg_callback` as observable is natural and general because this is the main communication medium between client and server for all OpenSSL functions, and is also the medium by which data is transferred and so it could be leaked. Labelling the structure `ssl_st` as public is natural and general because this is a structure, argument to most OpenSSL functions, that both parties have access to and can manipulate.

To check a function, say `f(i1, ..., in)`, for information leaks, we write a driver function defined according to the template in Fig. 3. This driver declares and possibly initialises the arguments (ensuring $\mathtt{a_i} = \mathtt{b_i}$ if that argument is public) and then checks that the results for the observables are the same. Given the OpenSSL labelling above described it is easy to instantiate such schema for a particular function that needs to be checked. In the case of these OpenSSL functions the driver asserts that given two calls to the function which have the same public inputs the resulting `msg_callback` observables are the same. For two of these functions, namely the ones with the Heartbleed bug we were able, by using quantifiers, to perform an unbounded security analysis. For the remaining functions the security analysis is bounded. Bounded means we can only assert that the observables in the resulting `msg_callback` are the same for the first n elements. The OpenSSL functions using `msg_callback` are shown in Fig. 4.

```
int dtls1_process_heartbeat(struct ssl_st *)
int dtls1_heartbeat(struct ssl_st *)
int dtls1_do_write(struct ssl_st *, int)
long int dtls1_get_message(struct ssl_st *, int, int, int, long int, int *)
long int dtls1_get_message_fragment(struct ssl_st *, int, int, long int, int *)
int dtls1_read_bytes(struct ssl_st *, int, unsigned char *, int, int)
int dtls1_dispatch_alert(struct ssl_st *)
int ssl23_client_hello(struct ssl_st *)
int ssl23_get_server_hello(struct ssl_st *)
int ssl23_get_client_hello(struct ssl_st *)
int ssl3_do_write(struct ssl_st *, int)
long int ssl3_get_message(struct ssl_st *, int, int, int, long int, int *)
int ssl3_read_bytes(struct ssl_st *, int, unsigned char *, int, signed int)
int ssl3_dispatch_alert(struct ssl_st *)
int tls1_process_heartbeat(struct ssl_st *)
int tls1_heartbeat(struct ssl_st *)
```

Fig. 4. OpenSSL functions analysed

Most OpenSSL are of the form `f(x)` where `x` is public, however if `f` uses process memory say by a call to `malloc` then it may well be that the two calls with the same public input result in different observables. This is automatically detected by CBMC thanks to its memory model.

We stress that while we check for information leakage on individual functions, our analysis is an information leakage analysis of the whole OpenSSL and not just a "unit testing" of a subset of OpenSSL. OpenSSL is essentially a library whose functions are called by server and client. By considering all functions affecting the observable `msg_callback` we are considering the whole of OpenSSL involving the data communication medium `msg_callback`.

4.2 Preparing for Analysis

Software projects of the scale of OpenSSL cannot be analysed at source-code level by picking up a single C file: numerous header files and configuration parameters contribute to each compilation unit. To employ CBMC in such a context, we use `goto-cc`, which can be used as drop-in replacement of various common compilers, including GCC. Running OpenSSL's standard build process, `goto-cc` builds an intermediate representation, called "goto programs" – a control-flow graph like representation – rather than executable binaries. The compiled files could be used directly with CBMC; for our experiments, however, we took the additional step of decompiling to C source code using `goto-instrument` (which is also part of CBMC's distribution). The resulting C code has all preprocessor macros and typedefs expanded, and adheres to any compile-time command-line options affecting the semantics of the program. A key benefit of this decompilation step is that our analysis could potentially be performed using any software analysis tool for C programs – such as KLEE.

4.3 Using Quantifiers for Unbounded Verification

Bounded model checkers unfold loops up to user-defined bounds. In certain cases, however, it is possible to use CBMC in a more powerful way. If we can replace a loop with a *quantified formula characterising the loop* then we can achieve unbounded verification.

The OpenSSL functions which suffered from Heatbleed allowed for this transformation. These functions call the standard library function `memcpy` in the following way:

memcpy((**void** ∗)bp, (**const void** ∗)pl, (**unsigned long**)payload)

The semantics of the function `memcpy` is to copy `payload` bytes of memory from the area pointed-to by `pl` to the memory area pointed-to by `bp` (we assume the memory regions involved do not overlap). Therefore the effect of this call can be summarised by the following quantified formula:

$$\forall\ (0 \leq i < \texttt{payload})\text{: } \texttt{bp}[i] == \texttt{pl}[i]$$

When loops are replaced by quantifiers, we can then use CBMC to translate the program and the assertions into a first-order formula over the theory of bitvectors. The obtained formulae are then passed to the SMT-solver Z3 [15] for satisfiability checking.

4.4 Missing Source Code and Compositionality Principle

When CBMC encounters a function call like `v=g(b)` and has no source code for the function `g` then a non-deterministically chosen value of the appropriate type is given to `v`. The implication for our analysis is that if there are some calls to missing functions and the analysis is successful, then the verification would be successful also if the source code were not missing[3].

On the other hand if the verification is unsuccessful then the failure may be spurious and originate from the non-deterministic choice of the missing function return value, because in each of the two runs different values may be non-deterministically chosen. A way to determine whether this is indeed the case is to make sure that the non-deterministically chosen return value for `g` is the same for the two calls of the function under analysis. This is easily achieved by defining this symbolic value as a non-deterministic global variable. Because of scalability issues we have excluded from the analysis the code of a few functions which we believe are safe to exclude, e.g., `dtls1_write_bytes`.

Compositionality Principle : If a function `f(a)` calls a function `g(b)` and the analysis reports `f(a)` to be secure while the source code for `g(b)` is missing (where the missing code is handled as explained above), and in an independent analysis `g(b)` is reported to be secure, then `f(a)` is secure[4].

[3] Provided these functions don't leak and return deterministic values. Also if these functions have side-effects these should be deterministic.

[4] A soundness proof of this principle for a complex language like C is arguably infeasible and surely beyond the scope of this work.

To verify confidentiality we can thus split the code base in several fragments. This compositionality principle is helpful when dealing with a large code base.

Notice that the converse direction is not valid, i.e., it is possible that the analysis returns that `f(a)` is not secure in the analysis where the source code for `g(b)` is missing, and the analysis returns `g(b)` is not secure but in fact the function `f` *is* secure. A simple example is the following program:

```
int f(int a) {
  int b=1;                  int g(int b) {
  int v=g(b);                 if(b) return 0;
   if(v) leak ...             else leak ...
  else non−leak ...         }
}
```

The function `f` leaks only if the value of `v` is 1 and `v` is set by the call to `g`. The function, `g` leaks only when `b` is 0. As `b` is set to 1 in `f` before calling `g` then `g` will not leak and return 0. This in turn will prevent `f` from leaking.

The analysis will return that `f` and `g` both leak when analysed in isolation. However, `f` is secure as `v` is never 1 inside `f` which is the only case when `f` leaks.

4.5 Analysis of OpenSSL Functions

For the analysis of the OpenSSL functions we use the basic driver pattern of Fig. 3. An example of initialisation of arguments for `dtls1_process_heartbeat` and `tls1_process_heartbeat` is reported in Fig. 5. The data size used is 37, because the size of payload and padding of a non-malicious heartbeat sent by the client is 34 bytes plus 2 bytes for the length and 1 byte for the type. Pointer `rrec.data` points to a structure for which we provide an unspecified values: this can be achieved in CBMC by giving to the element of the structure a non-initialised value.

Other functions analysed in OpenSSL use the pointer `init_buf.data` instead of `rrec.data`; however, the initialisation is similar. For `init_buf.data` we used the value 12, 24, and 48 as possible lengths. These values are simple guessworks on possible sizes and are just meant to prove that our methodology provides us with the ability to perform the analysis. An OpenSSL developer would be able to assign appropriate range of sizes for `init_buf.data` allowing therefore a more complete security analysis of the OpenSSL functions unrelated to Heartbleed.

Tab. 1 summarises the experimental results of the automated analysis using CBMC version 5.0. The tests were performed on Linux systems with 64-core AMD Opteron processors running at 2.5 GHz, equipped with 256 GB of memory.

In the table we write `fun__N__OPTION` meaning that the function `fun` was analysed by unrolling its loops `N` times and `OPTION` is one of the following:

– `C_NO_OBSERVATION_IS_LEAK`: with this option the assertion used is precisely the one from non-interference, i.e. it states that the observables are equal. If

```
struct ssl_st s_1,s_2;
int i;
s_1.msg_callback=fobservable_1;
s_2.msg_callback=fobservable_2;
struct ssl3_state_st s3_1,s3_2;
unsigned char r_data_1[37], r_data_2[37];
for(i=0; i<37; i++) {
  r_data_2[i]=r_data_1[i];
}
s3_1.rrec.data=r_data_1;
s3_2.rrec.data=r_data_2;
s3_1.rrec.length=37;
s3_2.rrec.length=37;
s_1.s3=&s3_1;
s_2.s3=&s3_2;
```

Fig. 5. Initialisation of data structures for `dtls1_process_heartbeat` and `tls1_process_heartbeat`. These structures are the public inputs for those functions

this option is not selected we use a weaker assertion, i.e. the assertion states that either the observables are equal or one of the observables is null, i.e. with the option not selected we accept a possible 1 bit leakage because, depending on the value of the secret, the function may produce a null observable or a specific non-null observable. The combination of these two assertions has shown to be helpful to detect spurious 1 bit leakage (details below).

- `C_INIT_BUF_LENGTH__M`: this option sets `init_buf.data` to size `M`.
- `C_HB_SEQ_HIGH`: this option sets `tlsext_hb_seq` field as high (i.e., confidential). This option only applies to `dtls1_heartbeat` and `tls1_heartbeat`.
- `C_HB_ART_LEAK`: this option adds an artificial information-flow leak (described later on) inside the function `(d)tls1_process_heartbeat`.
- `C_HB_BUG`: this option disables the Heartbleed patch.
- `C_FORALL`: this option introduces quantifiers.
- `C_HB_CORR_SIZE`: configures the heartbeat payload to the correct size.
- `C_RANDOM_LOW__M`: set `M` random bytes in the heartbeat payload to be public.
- `C_CLIENT_HELLO_CONSTRAINED`: forbid `ssl23_write_bytes` return value between 2 and 5.

Notice that a few functions with no option selected verify successfully and with option `C_NO_OBSERVATION_IS_LEAK` yield a counterexample. This indicates a possible maximal one-bit leak. A quick code inspection following the CBMC error trace suggests this small leak is spurious and caused by some missing initialisation or missing functions called by the analysed functions.

Functions `dtls1_process_heartbeat`, `dtls1_heartbeat`, `tls1_heartbeat`, `tls1_process_heartbeat`, and `ssl23_client_hello` show more serious failures: from a security perspective they are the most interesting and we now comment more in details on our findings.

Functions ***dtls1_process_heartbeat*** *and* ***tls1_process_heartbeat*** The verification fails when there is no patch and `rrec.data[1], rrec.data[2]` are left unspecified (i.e., option `C_HB_BUG`). This is the Heartbleed bug. In fact `rrec.data[1]` and `rrec.data[2]` together define the payload size. By not initialising these variables CBMC will find values mismatching the real payload size and so triggering Heartbleed. Notice that we are not only able to detect the leak but CBMC's counterexample tells us precise inputs triggering Heartbleed.

An important point is that our analysis require absolutely no knowledge or suspect of the existence of the Heartbleed in order to detect it. We stress that by leaving the size of the buffers `rrec.data[1], rrec.data[2]` unspecified we are eliminating the guesswork on the buffer size. That is we leave to CBMC to determine if there exist buffer sizes for which there is an information leak. CBMC is able to find the buffer sizes triggering the bug. This is an important feature of our analysis because if it were to rely on this guess work it would require the developer already to suspect the leak and where it could arise.

Once the patch is applied (i.e., removing option `C_HB_BUG`), the verification becomes successful. We add option `C_FORALL` to perform an unbounded verification by using quantifiers. As such our result provides the first formal verification that the patch actually fixes the Heartbleed bug.

Another case where the verification is successful is when the code is unpatched but `rrec.data[1]` and `rrec.data[2]` are given as values the correct payload size (option `C_HB_CORR_SIZE`). Since `rrec.data` is 37 bytes (the first byte is the type; the following two bytes are the length description and 16 bytes are padding) this is achieved by setting `rrec.data[1]=0;rrec.data[2]=18;`. As expected the verification is in this case successful.

To test the power of our approach we then inserted in hearbeat functions a leak originating from an indirect flow modelling the reading of one byte of process memory (option `C_HB_ART_LEAK`). Fig. 6 reports a snippet of the modified function once `C_HB_ART_LEAK` is used. The added lines test whether some byte from the process memory has a specific value (say 1). In that case the function assigns to the 6th element of `bp` the value 0 otherwise 1. Because `bp` is in fact a name for the buffer becoming later observable via `msg_callback` that bit of information about the process memory is leaked. Given this setting we get a verification failure. This case shows our ability to detect all possible leaks, i.e., not only leaks due to the bugs as in Heartbleed, but also those originating from direct and indirect flows of confidential information in code without bugs.

Functions ***dtls1_heartbeat*** *and* ***tls1_heartbeat*** The verification fails. On code inspection following the counterexample we notice that the reason is that the payload is randomly generated (see Sec. 2.1 for discussion). Once we assume that the payload is not confidential we can eliminate this leak from our analysis (option `C_RANDOM_LOW`). Consistently with the handling of random data described in the introduction, to implement the assumption that payload is not confidential we initialise all elements in the payload buffer to arbitrary yet identical values for the two runs. Under these conditions the verification succeeds.

```
bp = bp + (signed long int)2;
memcpy((void *)bp, (const void *)pl, (unsigned long int)payload);
char process_memory_byte;                              //ADDED CODE
if(process_memory_byte) bp[5]=0; else bp[5]=1; //ADDED CODE
bp = bp + (signed long int)payload;
RAND_pseudo_bytes(bp, (signed int)padding);
```

Fig. 6. Modified `(d)tls1_process_heartbeat` code with artificial leak.

We detected another potential leak (option `C_HB_SEQ_HIGH`) which could lead an eavesdropper to estimate how many heartbeats are exchanged. The leak originates from the `tlsext_hb_seq` field of the structure argument to the functions `dtls1_heartbeat` and `tls1_heartbeat`. This field stores a heartbeat sequence number and this information is leaked in the observable. Our default assumption is that the argument is public. However, our methodology is flexible enough to consider arguments that have both confidential and public components.

Function `ssl23_client_hello` The verification fails. On code inspection following the error trace provided by CBMC we discovered a possible (very large) information leak depending on the return value of `ssl23_write_bytes` which is called by `ssl23_client_hello`. With option `C_CLIENT_HELLO_CONSTRAINED` this return value is assumed not to be between 2 and 5 and we then succeed to verify the absence of leaks. The bound 5 comes from the packet header and should guarantee no abnormal behaviour is triggered. It would be possibly better to add a fail-safe feature to enforce these bounds, e.g., an if-then-else making sure the return value of `ssl23_write_bytes` is within those safe bounds and exit otherwise. This case illustrates how our analysis can help to determine possible conditions triggering a leak.

5 Conclusion

We presented a general technique for the analysis of confidentiality in complex C code. We applied our analysis to OpenSSL and showed that it correctly detects Heartbleed as a form of information leak. Moreover we verified that the patched code does not leak information. We verified the whole of OpenSSL for similar leaks. The analysis returned interesting findings and where CBMC failed to verify the absence of leaks, by using error traces we have found some possible security problems with the functions `dtls1_heartbeat`, `tls1_heartbeat` and `ssl23_client_hello`. In doing so we didn't have to modify the analysed code, but our approach, except for labelling, and writing the driver, works out of the box. The only annotation required is to label the confidential and non confidential data and what data and structures are observables to an attacker.

As any program analysis, our approach presents limitations. The main are:

Benchmark	Result	Time	RAM
		[s]	[GB]
dtls1_dispatch_alert__104__	✓	267.6	3.0
dtls1_dispatch_alert__104__C_NO_OBSERVATION_IS_LEAK	X	222.5	3.0
dtls1_do_write__58__	✓	305.3	0.8
dtls1_do_write__58__C_INIT_BUF_LENGTH__24	✓	234.1	0.9
dtls1_do_write__58__C_INIT_BUF_LENGTH__48	✓	202.3	1.0
dtls1_do_write__58__C_NO_OBSERVATION_IS_LEAK	✓	243.2	0.8
dtls1_get_message__7__	✓	32842.6	16.8
dtls1_get_message__7__C_INIT_BUF_LENGTH__24	✓	26933.5	16.9
dtls1_get_message__7__C_INIT_BUF_LENGTH__48	✓	27528.4	17.2
dtls1_get_message__7__C_NO_OBSERVATION_IS_LEAK	✓	30636.2	16.8
dtls1_get_message_fragment__18__	✓	5655.1	8.7
dtls1_get_message_fragment__18__C_INIT_BUF_LENGTH__24	✓	5493.6	8.8
dtls1_get_message_fragment__18__C_INIT_BUF_LENGTH__48	✓	3397.5	9.0
dtls1_get_message_fragment__18__C_NO_OBSERVATION_IS_LEAK	✓	5649.6	8.7
dtls1_heartbeat__20000__	X	3.5	0.1
dtls1_heartbeat__20000__C_HB_SEQ_HIGH__C_RANDOM_LOW__32	X	3.5	0.1
dtls1_heartbeat__20000__C_RANDOM_LOW__32	✓	3.3	0.1
dtls1_process_heartbeat__102__	✓	13.6	0.3
dtls1_process_heartbeat__102__C_FORALL	✓	3.1	0.0
dtls1_process_heartbeat__102__C_HB_ART_LEAK	X	8.3	0.3
dtls1_process_heartbeat__102__C_HB_ART_LEAK__C_FORALL	X	2.4	0.0
dtls1_process_heartbeat__102__C_HB_BUG	X	8.3	0.3
dtls1_process_heartbeat__102__C_HB_BUG__C_FORALL	X	636.3	0.2
dtls1_process_heartbeat__102__C_HB_BUG__C_HB_CORR_SIZE	✓	5.2	0.2
dtls1_process_heartbeat__102__C_NO_OBSERVATION_IS_LEAK	✓	10.8	0.3
dtls1_process_heartbeat__102__C_HB_ART_LEAK__C_FORALL__C_HB_CORR_SIZE	X	2.6	0.0
dtls1_process_heartbeat__102__C_HB_ART_LEAK__C_HB_CORR_SIZE	X	5.5	0.2
dtls1_read_bytes__	✓	247.2	4.0
dtls1_read_bytes__C_NO_OBSERVATION_IS_LEAK	✓	211.4	4.0
ssl23_client_hello__100__	X	108.9	2.0
ssl23_client_hello__100__C_INIT_BUF_LENGTH__24	X	96.5	2.1
ssl23_client_hello__100__C_INIT_BUF_LENGTH__48	X	99.7	2.1
ssl23_client_hello__100__C_NO_OBSERVATION_IS_LEAK	X	95.8	2.0
ssl23_client_hello__100__C_CLIENT__HELLO__CONSTRAINED	✓	83.8	2.0
ssl23_get_client_hello__1040__	✓	1026.1	10.4
ssl23_get_client_hello__1040__C_NO_OBSERVATION_IS_LEAK	X	933.0	10.3
ssl23_get_server_hello__1040__	✓	600.2	7.0
ssl23_get_server_hello__1040__C_NO_OBSERVATION_IS_LEAK	X	552.4	7.0
ssl3_dispatch_alert__18__	✓	1603.7	11.3
ssl3_dispatch_alert__18__C_NO_OBSERVATION_IS_LEAK	✓	1465.1	11.3
ssl3_do_write__58__	✓	1.1	0.0
ssl3_do_write__58__C_INIT_BUF_LENGTH__24	✓	1.4	0.1
ssl3_do_write__58__C_INIT_BUF_LENGTH__48	✓	1.6	0.1
ssl3_do_write__58__C_NO_OBSERVATION_IS_LEAK	X	1.4	0.0
ssl3_get_message__6__	✓	21.5	0.1
ssl3_get_message__6__C_INIT_BUF_LENGTH__24	✓	14.9	0.1
ssl3_get_message__6__C_INIT_BUF_LENGTH__48	✓	15.3	0.1
ssl3_get_message__6__C_NO_OBSERVATION_IS_LEAK	✓	20.0	0.1
ssl3_read_bytes__	✓	158.1	4.1
ssl3_read_bytes__C_NO_OBSERVATION_IS_LEAK	✓	222.7	4.1
tls1_heartbeat__102__	X	3.0	0.1
tls1_heartbeat__102__C_HB_SEQ_HIGH__C_RANDOM_LOW__32	X	3.7	0.1
tls1_heartbeat__102__C_RANDOM_LOW__32	✓	2.8	0.1
tls1_process_heartbeat__102__	✓	10.1	0.2
tls1_process_heartbeat__102__C_FORALL	✓	3.2	0.0
tls1_process_heartbeat__102__C_HB_ART_LEAK	X	8.8	0.2
tls1_process_heartbeat__102__C_HB_ART_LEAK__C_FORALL	X	1.7	0.0
tls1_process_heartbeat__102__C_HB_BUG	X	8.1	0.2
tls1_process_heartbeat__102__C_HB_BUG__C_FORALL	X	4.7	0.0
tls1_process_heartbeat__102__C_HB_BUG__C_HB_CORR_SIZE	✓	5.5	0.2
tls1_process_heartbeat__102__C_NO_OBSERVATION_IS_LEAK	✓	11.8	0.2
tls1_process_heartbeat__102__C_HB_ART_LEAK__C_FORALL__C_HB_CORR_SIZE	X	1.4	0.0
tls1_process_heartbeat__102__C_HB_ART_LEAK__C_HB_CORR_SIZE	X	5.1	0.2

Table 1. Benchmarks results obtained using CBMC 5.0; ✓is successful (bounded) verification, X denotes a counterexample

– As it is based on the bounded model checker CBMC, the approach is in general bounded. In some simple, yet crucial, case we were able to overcome this limitation by encoding loops with quantified formulae. However a general automated translation from loops to quantified formulae is a challenging problem and a topic left for further research.
– The analysis is not completely automatic but it requires some simple annotations by the user: *public*, *secret*, and *observable* data. The driver also requires some user effort, but it follows a simple pattern easy to implement. Also for a given specific software contexts the driver can be automated.
– While the methodology is completely general there may be some limitation introduced by the implementation platform. For example CBMC provides limited support for string manipulation functions. Hence it may return false positive when analyzing leakage from string formatting attacks involving uninterpreted functions in CBMC.

Acknowledgments

This research was supported by EPSRC grant EP/K032011/1

References

1. Anderson, P.: Finding heartbleed with codesonar, `www.grammatech.com/blog/finding-heartbleed-with-codesonar`
2. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: 17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004). pp. 100–114. IEEE Computer Society (2004)
3. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008. pp. 209–224. USENIX Association (2008)
4. Chou, A.: On detecting heartbleed with static analysis, `security.coverity.com/blog/2014/Apr/on-detecting-heartbleed-with-static-analysis.html`
5. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS 2004. pp. 168–176. Springer (2004)
6. Clarke, E.M., Kroening, D., Yorav, K.: Behavioral consistency of C and verilog programs using bounded model checking. In: DAC 2003. pp. 368–371. ACM (2003)
7. Coverity: `www.coverity.com`
8. Goguen, J.A., Meseguer, J.: Security policies and security models. In: 1982 IEEE Symposium on Security and Privacy. pp. 11–20. IEEE Computer Society (1982)
9. Grammatech: `www.grammatech.com/codesonar`
10. Heusser, J., Malacaria, P.: Quantifying information leaks in software. In: Twenty-Sixth Annual Computer Security Applications Conference 2010. pp. 261–269. ACM
11. HP/Fortify: `saas.hp.com/software/fortify-on-demand`
12. Klokwork: `www.klokwork.com`
13. Kroening, D., Tautschnig, M.: CBMC - C bounded model checker - (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014). pp. 389–391. Springer (2014)

14. Kupsch, J.A., Miller, B.P.: Why do software assurance tools have problems finding bugs like heartbleed? (Apr 2014), `continuousassurance.org/swamp/SWAMP-Heartbleed-White-Paper-22Apr2014-current.pdf`
15. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008. pp. 337–340. Springer (2008)
16. Risky Business: #339 – Neel Mehta on Heartbleed, Shellshock (Oct 2014), `media.risky.biz/RB339.mp3`
17. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)
18. Schneier, B.: Heartbleed (Apr 2014), `www.schneier.com/blog/archives/2014/04/heartbleed.html`
19. Seggelmann, R., Tuexen, M., Williams, M.: Transport layer security (TLS) and datagram transport layer security (DTLS) heartbeat extension. RFC 6520, RFC Editor (Feb 2012), `www.rfc-editor.org/rfc/rfc6520.txt`
20. Smith, G.: Principles of secure information flow analysis. In: Malware Detection, Advances in Information Security, vol. 27, pp. 291–307. Springer (2007)
21. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Static Analysis, 12th International Symposium, SAS 2005. pp. 352–367. Springer (2005)
22. Valgrind: `valgrind.org`
23. Zhang, L., Choffnes, D.R., Levin, D., Dumitras, T., Mislove, A., Schulman, A., Wilson, C.: Analysis of SSL certificate reissues and revocations in the wake of heartbleed. In: Internet Measurement Conference 2014. pp. 489–502. ACM