Heartbleed: A Formal Methods Perspective

Aalok Thakkar
University of Pennsylvania
athakkar@cis.upenn.edu

ABSTRACT

The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptographic software library caused due to a buffer overflow error. This report surveys work in the domain of formal methods motivated by and targeted at this bug. We discuss the bug in detail, its causes, possible attacks, and its impact and then look at two tools, Flinder-SCA, which combines static and dynamic analysis for detecting Heartbleed and Angelix, a semantics-based automated repair tool which synthesizes a correct patch for the vulnerability using a test suite. We also review methods and practices to avoid security vulnerabilities in general, and especially those due to buffer errors, and conclude with a discussion on possible directions for future works.

1 INTRODUCTION

"I cannot believe it. The internet was supposed to be a lawless frontier where all of humanity's desires and vices merge into a royal collective aid held in check by a barely regulated rat's nest of technical abstractions I don't understand. How did that get out of control?"

- The Colbert Report, Comedy Central, April 9, 2014

Stephen Colbert's sarcasm was in reference to the vulnerability named CVE-2014-0160 [10] in OpenSSL. OpenSSL is a cryptographic software library for applications that secure communications over computer networks against eavesdropping or need to identify the party at the other end. It is widely used by Internet servers, including the majority of HTTPS websites. Versions 1.0.1 through 1.0.1f had this severe memory handling bug that could be used to reveal the application's memory [6]. By reading the memory of the web server, attackers could access sensitive data, including the server's private key. This could allow attackers to decode earlier eavesdropped communications if the encryption protocol used does not ensure perfect forward secrecy. Knowledge of the private key could also allow an attacker to mount a man-in-the-middle attack against any future communications. The vulnerability can potentially reveal unencrypted parts of other users' sensitive requests and responses, including session cookies, login credentials, passwords, and other private data which might allow attackers to steal data directly from the services and users, and impersonate services and users [13]. At its disclosure on April 7, 2014, around 17 % of the Internet's secure web servers (about half a million servers) certified by trusted authorities were believed to have been vulnerable to the attack [19].

It was also the first time a computer bug became a sensation on media of every form, got a catchy logo and a name now known to all: *Heartbleed* [3]. The Heartbleed vulnerability also became a classic benchmark for network security, cryptography, and formal methods community and a number of papers were published in these communities that focused on the analysis, prevention, and patching Heartbleed and similar bugs. In this report, we analyze the impact of Heartbleed vulnerability from a formal methods perspective and discuss two tools: (1) Flinder-SCA, a combination of static and dynamic analysis to detect vulnerabilities and (2) Angelix, a semantic analysis based program repair tool which can synthesize a patch for Heartbleed.

Figure 1: The Heartbleed Logo. CVE-2014-0160 was the first computer vulnerability to have such a branding.

2 ANATOMY OF THE BUG

OpenSSL is a widely-used open-source cryptographic library that implements the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols. Their official web-page lists 176 vulnerabilities found in OpenSSL since 2002 [6], Heartbleed being the most popular and impactful in the list.

Heartbleed was introduced by the Heartbeat Extension in the OpenSSL version 1.0.1[1]. This extension enables a low-cost, keep-alive mechanism for peers to know that they're still connected and all is well at the TLS layer. Standard implementations of TLS do not require the extension as they can rely on TCP for equivalent session management.

The design of this extension is simple. Peers indicate support for the extension during the initial TLS handshake. Either end-points of the TLS connection can send a package called HeartbeatRequest along with an arbitrary payload and a field that defines the payload length[2]. The request should be answered by a HeartbeatResponse package that contains an exact copy of the payload and the sender's own random padding. This mechanism allows for a more streamlined checking of connection state and lowers client or server overhead on long-lived connections.

[1] The extension was introduced on March 14, 2012 and the vulnerability was first detected only in March 2014 and made public in April 2014

[2] the exact message includes a one-byte type field, a two-byte payload length, the payload, and at least 16 bytes of random padding

Version 1.0.1 of OpenSSL added support for the Heartbeat functionality and enabled it by default. This implementation of the extension contained a vulnerability that allowed either end-point to read data following the payload message in its peer's memory by specifying a payload length larger than the amount of data in the HeartbeatRequest message. Because the payload length field is two bytes, the peer responds with up to 64 KB of memory. The source of the problem is the assumption that the attacker-specified length of the attacker-controlled message is correct. This can be traced to a single line of code:

```
if (hbtype == TLS_HB_REQUEST) {
    ...
    memcpy(bp, pl, payload);
    ...
}
```

memcpy() is the command that copies data. bp is the place it's copying it to, pl is where it's being copied from, and payload is the length of the data being copied. There's never any attempt to check if the amount of data in pl is equal to the value given of payload.

It was only two years later that this vulnerability was found. The XKCD comic strip in Figure 2 gives a simplified understanding of how the vulnerability can be exploited to leak sensitive information.

As one may guess, the patch is simple. The developer-provided patch from April 9, 2014 adds a bounds check that discards the HeartbeatRequest message if the payload length field exceeds the length of the payload. The patch looks like:

```
/* Read type and payload length first */
if (1 + 2 + payload + 16 > s->s3->rrec.length):
    return 0;
    /* discard */
...
if (hbtype == TLS_HB_REQUEST) {
    /* receiver side : */
        /* replies with TLS1_HB_RESPONSE */
}
else if (hbtype == TLS_HB_RESPONSE) {
   /* sender side */
}
```

It is perhaps the simplicity of the bug because of which it took two years to detect it. However, while simple and easy to fix, its impact should not be understated. Heartbleed not only affected web servers, but also affected many embedded systems, including printers, firewalls, VPN endpoints, NAS devices, video conferencing systems, and security cameras. It also affected Mail Servers, Tor Project, Bitcoin Clients, Android devices and even Wireless Networks [12]. It is because of this impact that Heartbleed haunts the benchmarks suits of modern formal methods tools.

3 DIAGNOSIS: COMBINING STATIC AND DYNAMIC ANALYSES

A central challenge in verification and bug-finding is to define exactly what behavior corresponds to a bug. That is, to have a formal specification for the correctness of a program. For security vulnerabilities, this process is challenging (as well as expensive) and the effort is justifiable only when high-integrity security is required.

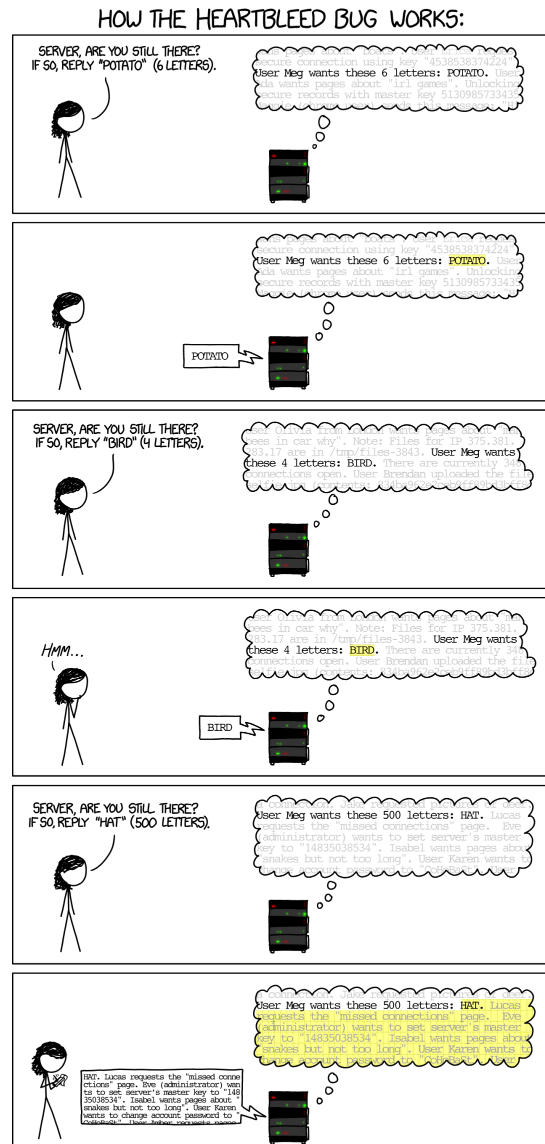

Figure 2: A summary of the working of the Heartbleed vulnerability presented an an XKCD comic strip.

The correctness properties of OpenSSL are not well defined and hence verification remains a challenge. The papers [9], [14], and [20] examine why static analysis and dynamic analysis methods missed out on finding this bug.

The main difficulties in detecting Heartbleed with static analysis tools were rooted in the way data is stored and referenced, complexity of following the execution path, difficulty of identifying the specific parts in the storage structure that are misused, and resistance to taint analysis heuristics due to the difficulty of determining whether a specific part within a complex storage structure has become untainted. On the other hand, the custom memory management used by OpenSSL would prevented dynamic testing

frameworks to successfully detect a memory corruption or over-read problem. This — combined with encapsulation of the heartbeat length field within the payload — made its detection via fuzz testing infeasible. In the end, Heartbleed was detected by Neel Mehta of Google's Security Team using manual code review in March 2014.

One must note that program analysis methods are based on heuristics and continuously improve. When important vulnerabilities like Heartbleed are missed by these methods, the developers tend to add certain functions and heuristics to focus on them. Coverity, for example, developed some new heuristics that to detect Heartbleed and similar vulnerabilities [2].

We now look at a technique to combine static and dynamic analyses to find Heartbleed and similar vulnerabilities. The Flinder-SCA tool developed by Balázs Kiss, Nikolai Kosmatov, Dillon Pariente, and Armand Puccetti puts together four techniques - Abstract Interpretation, Taint Analysis, Program Slicing, and Fuzzing to detect vulnerabilities in source code. We review these four techniques and discuss the application of Flinder-SCA tool for detecting the Heartbleed vulnerability as illustrated in their 2015 paper [16].

3.1 Abstract Interpretation

The first step is to detect potential runtime errors using abstract interpretation.

Abstract interpretation is a framework for automatic and sound approximation the semantics of the program. It provides an automatic extraction of information about all the possible executions of program.

The concrete semantics of programs is the set of all possible executions of the program in all possible execution environments. Non-trivial properties of this (potentially infinite) set can be hard to analyse and are often undecidable. However, one can find an over approximation of this set, called abstract semantics, which may be easier to work with. If one can show that the over approximation satisfies the desired properties, then the concrete semantics also satisfy the properties. It is easy to see that the converse isn't true, which makes this technique sound but not complete.

For the Heartbleed vulnerability, we compute the abstract (over-approximated) domains of possible values for program variables at each program location. To show the absence of bugs, it is sufficient to show that for the memcpy call, pointers bp and pl are in the range of the payload size.

Flinder-SCA is based on the Frama-C framework, which supports the Value[3] plug-in. This plug-in has two functionalities of use here. The first is Runtime Error Annotation Generation which generates the assertions of the form:

```
// assert A1 : mem_access: \valid(bp[0 .. (payload -1)]);
// assert A2 : mem_access: \valid(pl[0 .. (payload -1)]);
```

The second is the abstract interpretation based value analysis method which can indicate that the pointers bp and pl may exceed these ranges and therefore dereferencing them may be dangerous.

3.2 Taint Analysis

As the analysis method is sound but not complete, we are guaranteed to generate alarms for all potential runtime errors. However, we may also generate false alarms due to over-approximations. The paper first reduces the alarms of interest by focusing on only security vulnerabilities by using a version of value propagation analysis called taint analysis [11]. Taint analysis identifies code variables and statements concerned with the propagation of potentially corrupted inputs (which may contain information controlled by an attacker). Such inputs are propagated through pointer aliasing, copy of memory zones, etc. If the user (or developer) can identify potentially corrupted inputs and vulnerable functions, value propagation analysis can identify the assertions related to security alarms and distinguish them from other alarms generated by the abstract interpretation.

The proposed taint analysis approach is based on static analysis results computed by Value plug-in. To apply it on the Heartbleed case, the user specifies the potentially taintable inputs (rrec.data, the major part of the HeartBeat message sent by the client), and the potentially vulnerable functions (e.g. memcpy). The tool reports that the assertions related to memcpy call handle the taintable input flow, and the memcpy statement is identified as vulnerable.

3.3 Program Slicing

The third step is to reduce the complexity of program which is to be analysed using program slicing. Program slicing consists of computing the set of program instructions, called *program slice*, that may influence the program state at some point of interest. This too is an over approximation. Slicing preserves the behaviors of the initial program at the selected criterion after removing irrelevant instructions.

One can use Slicing plug-in of Frama-C to implement this technique. In the case of Heartbleed, slicing allows us to simplify the code with respect to the set of assertions produced by the previous analysis. The original program with the containing 8 defined functions and 51 lines of code is simplified to keeping only 2 functions and 38 lines in the slice used in the last step.

3.4 Fuzz Testing

The first three methods - Abstract Interpretation, Taint Analysis, and Program Slicing are implemented by static analysis of the control flow of the program. At the end of these static analyses we identify a small part of code which is potentially vulnerable. Now we wish to find a concrete attack for the program.

Fuzz testing is a technique to provide a system faulty or erroneous input and monitor its states. Detecting an error error indicates that the system cannot properly handle the given input, confirming the existence of a bug in the code. Unlike the previous static analyses methods, this method is complete but not sound. To implement fuzz testing, we need to generate syntactically correct inputs that raise alarms in the program.

The Flinder fuzz testing framework uses *white box* fuzz testing which has the following steps:

(1) Generate a list of fuzzing parameters for each variable to be modified, specifying what kind of values should be generated for them to look for certain kinds of vulnerabilities.
(2) Compile the annotated code (from the first three analysis steps) and feed it to the tool.

[3] Eva plug-in in the latest versions of Frama-C

(3) Test vectors are generated according to the fuzzing parameters.
(4) Each test vector is sent to the test harness where its values are used to initialize the variables targeting by the fuzzing tool at runtime. The test harness detects the alarms raised and logs them.
(5) Based on the presence of anomalies in the logs the tool decides whether the vulnerability is confirmed or not.

The white box fuzz testing used in the paper is Flinder [1]. In the Heartbleed example, the static analysis step reports to Flinder six potential bugs. Flinder generates 10 test cases for a different-size Heartbeat message buffer, and 32 test cases each for different Heartbeat message length and sequence number values. The first test case where the Heartbeat message length is larger than the buffer size causes an invalid memory read attempt. Captured by the test harness, this operation allows Flinder to identify the specific alarm connected to the test.

4 PREVENTION: THE GOOD HABITS OF PROGRAM DEVELOPMENT

OpenSSL is open source. Anyone could look at the code, and presumably hundreds did, but nobody noticed the fairly elementary coding error. A common misconception is that open source is magically protected by the community as a large number of eyeballs go though each line of code. However, this also causes a type of digital bystander effect: no one actively looks for a bug assuming that someone else may have already checked the correctness. Heartbleed is an example of where this baseless trust in the open source community can have drastic effects lead us. While the Heartbleed wiki page specifies how and when the bug was introduced into the code base, it doesn't disclose how the OpenSSL code was security tested, either statically or at runtime. It was only through a manual reading of the code that Neel Mehta of Google Security identified the vulnerability in March 2014.

As we have realized, security vulnerabilities are hard to detect and if left unattended can have a drastic impact. On one extreme, one can stick to working only with a coherent system of well specified and verified components. One line of work in this area is the Deep Specification project which addresses this problem by showing how to build software that does what it is supposed to do, no less and no more [7]. This is a very ambitious goal. Given that OpenSSL did not undergo a thorough human review, nor is any formal correctness specification available (even after the Heartbleed episode), having a fully verified framework is a distant dream, especially in the open source community. The following are a few good habits of program development which can help prevent such security vulnerabilities.

A deeper study of the root cause of the bug reveals how we can prevent it. A simple first step is to ensure that buffers are handled safely. The MITRE Corporation, which maintains the Common Weakness Enumeration (CWE) catalog, ranked buffer handling errors (including buffer overflows and out-of-bound reads) the most common type of software vulnerability [5]. One way to prevent these errors is to train developers to avoid making those errors. Another way is to move away from languages like C which do not have any detection mechanisms or countermeasures for improper buffer restriction (including buffer overwrites and overreads) built in. A number of other languages (including Java and Python) handle this by techniques like resizing data structures or by raising an exception when the buffer is exceeded. One can also use a memory-safe programming language (like Rust) to prevent buffer overflows.

Another good habit is to write code which is easy to analyse, both through manual review and by using fully automated formal methods. D. A. Wheeler discusses them in detail in his essay [20]. Some of them are:

(1) Simplify the code and its API. Security-sensitive code needs to be simple enough that errors are apparent. Complex code impede formal methods.
(2) Secure programs shouldn't involve special, program-specific allocation or memory caching systems. Mechanism that subdivides memory inside the application can thwart analysis.
(3) Using a standard open source software license (such as GPL, LGPL, MIT/X, Revised BSD, FreeBSD or Apache 2.0) allow for more code reviews and contributions to code. This can potentially detect more bugs.

Additionally, Wheeler discusses eight practices that can reduce the impact of similar security vulnerabilities. Some of them are:

(1) Program and functions that manipulate sensitive and critical information should overwrite, erase, or destroy this as quickly as possible (once it is not needed anymore).
(2) A perfect forward security (PFS) encryption algorithms should be used. PFS algorithms non-deterministically generate new random public keys for each session, preventing the compromise of a private key that may enable attackers to decrypt past communications that they have recorded.
(3) Separating critical cryptographic secrets from the rest of the code can ensure that even if even the rest of the program is subverted, the attacker cannot directly access secrets like private keys.
(4) The problems in the SSL/TLS infrastructure such as including untrustworthy root certificate authorities, poorly-vetted certificates, and the certificate revocation process should be fixed.
(5) Software updates should be easy and required (users of unupdated Android 4.1.4 are currently vulnerable to Heartbleed)
(6) The incentives for attackers should be reduced through policy, legal, and economic changes.[4]

But apart from these measures, the most important and effective way to prevent security threats is to foster better understanding and awareness for secure software development. Understanding the operational semantics of programming languages, libraries and tools used can help avoid most bugs. Formal specification and rigorous validation and verification of programs, if made an essential part of educational and training curriculum for developers, can ensure that their code meets the intention and has fewer bugs and vulnerabilities.

[4] As of now, people can make more money selling information about vulnerabilities to attackers and they have neither any obligation (legal or otherwise) nor much incentive to notify the developers or governments.

5 TREATMENT: AN APPLICATION OF SYNTHESIS

Bugs are fixed by patching the vulnerable part of the code in the updated versions of the software. In general, even when the bug has been detected, it might be difficult to design a patch to repair the program. Moreover, once the developer designs a patch, they may have to go through the verification process to ensure that the patch is free of vulnerabilities.

This challenge provides an opportunity for *automated program repair* an emerging of research that allows for automated rectification of software errors and vulnerabilities. Automated repair techniques try to automatically identify patches for a given bug which can then be applied with little, or possibly even without, human intervention. In this section we review a semantic analysis based program repair tool called Angelix which and scales up to real world software and is also the first tool to synthesize a patch for Heartbleed [18].

5.1 Repair as Synthesis

Angelix belongs to the class of constraint-based repair techniques. These techniques proceed by first [5] constructing a repair constraint that the patched code should satisfy and then treat the patch code as a function to be synthesised. A choice of program analysis and synthesis techniques can be used for both of these steps. Because a formal specification is often not available for the requirements of the patch, Angelix relies on a comprehensive test-suit and generates repair patches which are guaranteed to be correct for the given test suite. It follows the schema:

for test $t \in$ test suite T :
  compute repair constraint ϕ_t
let $\phi = \bigwedge_T \phi_t$
synthesize e as a solution for ϕ

In this case, T is the test suite used as the correctness criterion. The constraint for a given test case $t$is computed by the symbolic execution of the path traversed by test case t. Let π_t be the path condition of the path traversed by test t (that is π_t holds only for inputs which traverse the program path as t), let ω_t be the symbolic expression capturing the output variable in the execution of t and ξ_t be the expected output. Then, the constraint for the test case is of the form:

$$\phi_t \equiv (\pi_t \wedge \omega_t = \xi_t)$$

One can then compute the conjunction ϕ and use constraint solving methods generate the patch which satisfies this constraint. These constraint specifications can grow with the size of the program and solving them can be intractable causing scalability issues. Angelix uses the technique called *Angelic Forest Extraction* for a more efficient inference of value-based specifications which localizes the bug to program expressions.

[5] In practice, program repair methods first transform the program into the abstract syntax tree or a choice of a standardized syntax and then localize the bug to a particular function or section of the code. We skip these two steps in this report.

5.2 Angelic Forest Extraction

We first define three concepts.

(1) Given an expression e in the program and a failing test case t an *angelic value* is a constant α such that replacing e by α makes the program pass t.
(2) For a program P, given a set of program expressions E and a test case t, the angelic path $\pi(t, E)$ is a set of triples (e, v, σ) such that $e \in E$ is an expression, v is an angelic value of e and σ maps every variable visible at location of e to their values such that on replacing each e with its corresponding v implies P passes test t and visible variables x at location of e have value $\sigma(x)$.
(3) For a program For a program P, given a set of program expressions E and a test case t, an *angelic forest* is a collection of angelic paths $\{\pi_1(t, E), \dots, \pi_n(t, E)\}$.

The angelic forest collects enough semantic information to enable repair, while being independent of the size of the program.[6] The angelic forest is computed by a variant of symbolic execution. Instead of initiating symbolic execution with symbolic input, few potentially vulnerable program locations are installed with uninterpreted symbols[7].

The variant of symbolic execution used in the paper is named controlled symbolic execution (CSE). In this method, symbols are installed during symbolic execution by replacing the value of each instance of a suspicious expression with a fresh symbol. Additionally, for each visible variable x at the location of a suspicious expression e, we add the constraint that the value of x in the context of e should be equal to $\sigma(x)$, the concrete or symbolic value of x evaluated during symbolic execution. This allows us to maintain angelic states of the suspicious expressions. Non suspicious expressions are evaluated using conventional symbolic execution. The full paper details the algorithm and it is implemented on top of KLEE tool [8].

As sketched in the algorithm in Figure 3, we perform CSE for every explorable path of the program. CSE produces a pair of a path condition pc and an actual output O_a of the program. Given the expected output O_e available in the test, we find a model of $(pc \wedge O_a = O_e)$ using a constraint solver. This model is used to extract an angelic path, and thereafter grow the angelic forest.

5.3 Patch Code Generation

Once an angelic forest is obtained, it is fed to the repair synthesizer as a synthesis specification. A synthesized repair, when executed, follows one of the angelic paths for each test, and therefore all tests in the suite pass. As tests describe the correctness criterion, this method gives us a "correct" patch.

The repair synthesizer used is an implementation of component based repair synthesis (CBRS) [17], which is a generalization of component-based program synthesis [15] to program repair. It uses

[6] The caveat is that it also under-approximates, in the sense, it does not capture whole (possibly infinite) set of values for the suspicious expressions that make the test pass. The method is sound but not complete, that is the repair obtained by this method passes all the provided tests but it may miss out on some repairs, due to the under-approximation of angelic values.

[7] these may be identified by methods such as statistical fault localization. The choice of the number of vulnerable locations control the execution paths to be explored during symbolic execution.

$A = \{\}$
while (there is an unexplored path $\wedge \neg$ timeout) **do**
\\ Perform CSE
$(pc, O_a) \leftarrow \mathsf{CSE}(I, E)$
$R \leftarrow (pc \wedge O_a = O_e)$
if R is satisfiable **then**
$M \leftarrow \mathsf{GetModel}(R)$ \\ using constraint solver
$A \leftarrow A \cup \mathsf{ExtractAngelicPath(M)}$
end if
end while
return A

Figure 3: Algorithm for Angelic Forest Extraction using Controlled Symbolic Execution (CSE). Given program P, test case (I, O_a) and suspicious expressions E, this algorithm returns the angelic forest A.

a technique called Partial Maximum Satisfiability Modulo Theories (pMaxSMT) The pMaxSMT solver takes two types of SMT clauses as input - hard constraints and soft constraints, and solves the problem of finding an assignment of the variables that satisfy all hard constraints and maximum possible number of the soft constraints.

The constraints for CBRS are formulated in terms of components. A component is a variable, a constant or a term over components and operations defined in a given background theory. To represent the connection between components, the input and output of components are associated with distinct variables called *location variables*. Two components are considered connected if and only if the location variable of one component has the same value as the location variable of another component. Then, the following constraints are imposed:

(1) a *range constraint* that places all components inputs and outputs within a legal range,
(2) a *consistency constraint* that ensures that the output of each component has a distinct location,
(3) an *acyclicity constraint* that prohibits cyclic connections, and
(4) a *connections constraint* that connects location variables with their corresponding components.

These constraints ensure that a synthesized expression is well-formed. Additionally, we have the *semantics constraint* (or the specification for the synthesis problem from section 5.2). These constraints are given to a constraint solver and the repair patch can compiled from the solution.

However, there can be multiple patches satisfying a given repair constraint. In those cases, it is assumed that a patch which makes minimum changes to the original program would be prefered by the developers as they are are easier to validate and they are less likely to change the correct behavior of the original program than more complex patches. Additionally, without this constraint, the synthesizer would always modify all the suspicious expressions making the repair impractical. For this, the aforementioned constraints are used as hard constraints to the pMaxSAT solver and the *structural constraints* that capture the structure of the original buggy program are used as soft constraints. The solution to the pMaxSMT with these hard and soft constraints gives us the desired patch which satisfies all test cases while making minimal changes to the original program.

5.4 A Patch for Heartbleed

Angelix was applied to OpenSSL for repairing the Heartbleed bug. The 12 tests from Mike Bland's regression test suite [4] were used, along with four additional tests to cover missing corner cases. It generated the following patch:

```
if (hbtype == TLS_HB_REQUEST
    && payload + 18 < s->s3->rrec.length) {
    /* receiver side : */
        /* replies with TLS1_HB_RESPONSE */
}
```

With this fix, memcpy cannot be invoked if the payload length (pl) is larger than the length of the payload. Observe that the synthesized patch is composed of the same operators and the developer provided patch. In both the patches the bounds check is performed by the added conditional, makes the receiver simply return zero, instead of replying with a response packet. Hence the two are also functionally equivalent. The authors neither provide the heartbleed case study nor the test case suit used by them as a part of the public version of the Angelix tool.

6 CONCLUSIONS

The popularity of the Heartbleed vulnerability has catalysed the research in the formal methods community towards security vulnerabilities. This report surveys and summarizes some of these works. The works discussed in this report provide novel and practical solutions to the problem of detecting, preventing, and patching bugs like Heartbleed. An immediate challenge is the osmosis of these tools and techniques from academicians to software developers. A few possible directions for future work are as follows:

(1) Combining Static Analysis and Testing: Success of combining formal methods techniques that provide different kind of guarantees have worked well in the past, including the work discussed in this report for detecting Heartbleed. The design of a general framework to combine static analysis and testing is an exciting and unexplored area of formal methods.
(2) Formal Specification of Security Properties: Unlike the safety and liveness properties of reactive programs which have a rich language for formal specification, and general tools built for these languages and fragments, much remains to be explored in the area of formal languages for specifying security properties. Formal specification often promote and catalyze the development of verification techniques.
(3) Safe by Design Programming Languages: The core of Heartbleed is how C deals with buffers. Developing and using domain specific programming languages that ensure that a class of vulnerabilities are absent by design is a possible future direction.
(4) Automated Program Repair: The technique remains an enticing yet achievable possibility that can improve program quality as well as the development experience. Challenges in this domain such as defining and pruning the space of patches, generalizing overfitting repairs, scalability, and the

lack of formally defined specification offer many research opportunities.

Additionally, designing an economic, political, and legal system of incentives and penalties that promote good software engineering and bug finding practices remain an open problem. There is also much scope for transforming software engineering pedagogy by including the art of developing correct programs, mathematically thinking about their properties and requirements, and formally validating and verifying them.

REFERENCES

[1] 2010. *Flinder automated security and robustness testing.* http://www.flinder.hu/flinder/index.html

[2] 2014. *Coverity Releases Platform Update for OpenSSL 'Heartbleed' Defect.* https://news.synopsys.com/2014-04-25-Coverity-Releases-Platform-Update-for-OpenSSL-Heartbleed-Defect

[3] 2014. *The Heartbleed Bug.* https://www.heartbleed.com/

[4] 2014-2016. *Unit test for TLS heartbeats.* https://github.com/xbmc/openssl/blob/master/ssl/heartbeat_test.c

[5] 2019. *Common Weakness Enumeration.* https://cwe.mitre.org/

[6] 2019. *OpenSSL Vulnerabilities.* https://www.openssl.org/news/vulnerabilities.html

[7] 2020. *The Science of Deep Specification.* https://deepspec.org/main

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. 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* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[9] M. Carvalho, J. DeMott, R. Ford, and D. A. Wheeler. 2014. Heartbleed 101. *IEEE Security Privacy* 12, 4 (2014), 63–67.

[10] National Vulnerability Database. 2014. Vulnerability Summary for CVE-2014-0160. (2014).

[11] Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (May 1976), 236–243. https://doi.org/10.1145/360051.360056

[12] Zakir Durumeric et. al. 2014. The Matter of Heartbleed. *Proceedings of the 2014 Conference on Internet Measurement* (2014), 475–488.

[13] IPSec.pl. 2014. *Why Heartbleed is dangerous? Exploiting CVE-2014-0160.* https://ipsec.pl/ssl-tls/2014/why-heartbleed-dangerous-exploiting-cve-2014-0160.html

[14] Barton P Miller James A Kupsch. 2014. Why Do Software Assurance Tools Have Problems Finding Bugs Like Heartbleed? *Continuous Software Assurance Marketplace* (2014).

[15] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 215–224. https://doi.org/10.1145/1806799.1806833

[16] Balázs Kiss, Nikolai Kosmatov, Dillon Pariente, and Armand Puccetti. 2015. Combining Static and Dynamic Analyses for Vulnerability Detection: Illustration on Heartbleed. In *Hardware and Software: Verification and Testing*, Nir Piterman (Ed.). Springer International Publishing, Cham, 39–50.

[17] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 448–458.

[18] S. Mechtaev, J. Yi, and A. Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 691–701.

[19] Paul Mutton. 2014. *Half a million widely trusted websites vulnerable to Heartbleed bug.* https://news.netcraft.com/archives/2014/04/08/half-a-million-widely-trusted-websites-vulnerable-to-heartbleed-bug.html

[20] D. A. Wheeler. [n.d.]. *Preventing Heartbleed.* https://dwheeler.com/essays/heartbleed.html