Securing the Web

by

Ting Yuen Lam

Keble College

This dissertation is submitted to the University of Oxford in partial fulfilment of the requirements for the degree of *Master of Science in Computer Science*

Supervisor: Prof. Daniel Kroening

Department of Computer Science
University of Oxford
Trinity 2016

Submitted on: August 30th, 2016

Acknowledgements

I would like to express my deepest appreciation to many individuals and organisations who have supported me to complete my project. A special gratitude to my project supervisor, Prof. Daniel Kroening and his research assistant, Martin Brain. I am grateful for their friendly advice and encouragements, especially for numerous consultations and their enlightening views on the structural improvement of my report. I am also thankful to Ernie Cohen, Principal Software Engineer at Amazon Web Services and Dr Michael Tautsching, Lecturer at the Queen Mary University of London, who have explained me the structure of s2n and clarified some basic issues on the CBMC operations, respectively, at the early stage of my project.

Moreover, I would like to thank my program supervisor, Prof. Michael Benedikt, who have warmly guided me in academics during the year.

I appreciate my sponsors, Alistar Harvey Foundation Scholarship, who have fully sponsored my study at the University of Oxford. Thanks for their generosity, I could complete my master's study without worrying about my financial status.

A special thanks to Keith Leung, Jason Mo, Jacky Chan and Brenda Sham for proof-reading my dissertation and saving it from a mess of poor English.

Last but not least, I would like to express my deepest gratitude to my parents and Brenda Sham, my girlfriend, for their moral supports and maintaining the quality of my life on daily basis. Their supports and encouragements have made the complete of my dissertation possible and have allowed me to realise my potential.

Abstract

Software verification is critical to ensuring the overall safety of complex systems in different industries. Examples include bounded model checking used in securing aircraft control systems, nuclear reactor controllers and medical systems. With regard to the network infrastructure of these sophisticated systems, the respective implementation of the TLS/SSL protocols is crucial as it is responsible for the privacy and maintenance of data integrity of the entire network communication. In light of the Heartbleed problem, Amazon designed a lightweight implementation of the TLS/SSL protocols known as s2n, with security and reviewability in mind. This dissertation aims to verify the memory safety of the s2n implementation by using CBMC.

The research problems for this project are to investigate whether there are any hidden bugs in a heavily-tested critical software package (s2n) and the capability of a bounded model checker, CBMC, to detect these hidden bugs. In this project, we demonstrate a possible way to verify an industrial software package as well as show the memory safety of its implementation and the over-approximation techniques to scale up the verification. From the experimental results, we found two arithmetic overflow bugs that exist in the s2n implementation and the respective solution has been reported to and accepted by Amazon's s2n developers.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Failure of ensuring the safety of network infrastructure components could cause confidential information leakage and fatal damages to enterprises. In April 2014, a simple careless mistake caused a security nightmare, namely the Heartbleed Bug[1], to the Internet disclosed by both Google's Security team and Codenomicon separately. It is a serious vulnerability in the extensively used Transport Layer Security (TLS) protocol implementation, OpenSSL cryptographic library[2]. This vulnerability enabled attackers to steal sensitive information from HTTPS services and even impersonate services or users. In this case, about half a million (17.5%) of trusted HTTPS websites were influenced, which included banking services, email services, cloud storage, and any web services using the vulnerable versions of OpenSSL, according to the Secure Sockets Layer (SSL)[3] survey conducted by the Netcraft [35].

This security weakness occurred in the TLS "heartbeat" extension, which is a keep-alive feature designed to ensure the connection by sending an arbitrary payload to, and receiving the exact same copy of it from the other end. The weakness was due to the fact that the implementation mistakenly relied on non-sanitised user input and took it as the length parameter of `memcpy()` without any bounds check [49]. Such a missing of a proper input validation makes the reading of the payload beyond the end of the buffer. Hence, attackers were able to extract the memory contents, such as identifications of service providers, secret keys for encrypting communications and users login information under the protection of vulnerable versions of OpenSSL in the server. In other words, they can steal data from communications directly and impersonate the services and users. However, this serious security vulnerability caused by buffer overruns can be easily detected using formal

[1] The Heartbleed Bug: http://heartbleed.com/

[2] The OpenSSL Github repository: https://github.com/openssl/openssl

[3] Secure Sockets Layer: the predecessor of TLS, often be called TLS/SSL

software verification techniques, for example, Bounded Model Checking, as demonstrated by the study of M. Voelter et al. [45]. Therefore, not only the approach itself must be formally proven secure, but also the corresponding implementations are necessary to be verified.

Since data integrity and privacy of Internet communications for the online applications, especially for those in financial usage, and virtual private networks (VPNs) are provided by TLS/SSL. Therefore, ensuring the safety of the TLS/SSL implementation is particular importance so as to secure the web. This project is motivated by the necessity of ensuring the safety and correctness of such critical software components and the success of software verification. We aim at verifying memory safety of the Amazon's s2n due to its characteristics of light-weight implementation of the TLS protocol using a bounded model checking tool called CBMC.

1.2 Objectives

The main objective of this project is to improve the web security by securing the critical network infrastructure component. Therefore, we propose this project, Securing the Web, in order to investigate the memory safety of the Amazon's s2n TLS implementation by using the Bounded Model Checking technique. As a result, a detailed experience report will be delivered and demonstrate how CBMC can be made use of in finding bugs in a heavily-tested critical software and showing the memory safety of the implementation.

1.3 Main Contributions

There are seven main contributions we have achieved in this project:

1. Two hidden arithmetic overflow bugs are found in the s2n.

2. Show CBMC is capable of finding hidden bugs in a heavily tested software package.

3. Show the memory safety of the s2n implementation covered in our verification.

4. Present an verification approach suitable to object-oriented software.

5. Demonstrate two different ways to scale up the verification by using over-approximation.

6. Analyse the experiment results and limitations of the current approach.

7. Document a detailed experimental report on the s2n verification using CBMC.

1.4 Dissertation Structure

This dissertation consists of eight chapters that fully described the background knowledge and demonstrated all the work in this project. It is organised as follows: Chapter 1 is a brief introduction of this dissertation including the project motivation, objectives and the dissertation organisation.

Chapter 2 describes an overview of the existing software verification techniques, and their strengths and weakness as well as highlights the trade-offs in a software verification. Moreover, it also describes some researched approaches on using the bounded model checking technique.

Chapter 3 gives an overview of the Amazon's s2n TLS implementation and demonstrates the importance of ensuring the safety and correctness of such an implementation. Furthermore, it also comprehensively explains the fundamentals of bounded model checking and provides a thorough introduction of the bounded model checker, CBMC, being used in our verification.

Chapter 4 describes the hypothesis of this dissertation and presents our verification approach in detail.

Chapter 5 shows our experiment setup and the instruction for installing CBMC and s2n. Moreover, it describes the procedure of how a CBMC execution command is constructed.

Chapter 6 displays the overall verification results and explains each failure found during the verification. More importantly, it also shows the two arithmetic overflow bugs found in the s2n implementation and the corresponding solutions.

Chapter 7 discusses on how the memory safety of the s2n implementation can be shown, the trade-off we have made and the limitation of the verification.

Chapter 8 draws a conclusion for this project and discusses several possible future works.

Chapter 2

Context and Rationale

Clarification The terms of soundness and completeness of a software verification technique often have various meanings in different papers. Sometimes soundness and completeness refer to an under-approximation and an over-approximation to a program behaviour respectively while sometimes they are defined in the opposite way. To be clearer, we will avoid using them throughout the report instead of using ”no missing bugs” and ”no false alarms” of a detection.

2.1 Overview of Software Verification

Software verification is an important step in the software development life cycle, which is easily confused with software validation. It is questioning about ”*are we building the right product?*”, while software verification cares about ”*are we building the product right?*” [30]. In another word, we concern about **whether a system meets its specification**, including safety properties, liveness properties, concurrency properties, as well as functional and non-functional properties. These are the keys to ensuring quality and reliability of the software, especially for *critical software*, such as safety-critical, security-critical, financial-critical. By analysing the system behaviour and detecting errors to minimise or prevent system failure.

There are numerous verification techniques, based on formal or informal methods, available into to achieve the ultimate goal of software verification. These techniques can be divided into three categories according to their approach: *Static Analysis*, *Dynamic Analysis*, and *Human-directed Analysis*. Although each of them has their focuses, strengths and weaknesses, their verification achievements can be concluded in three main properties: being automatic, not missing bugs, and excluding false alarms by M. Brain and D. Kroening [6]. These can also be visualised in a triangle model, which has been illustrated in Figure 2.1 below. Since there always is a trade-off between the three proprieties, it becomes a weakness of such techniques. They can be considered as sitting at a corner of the triangle

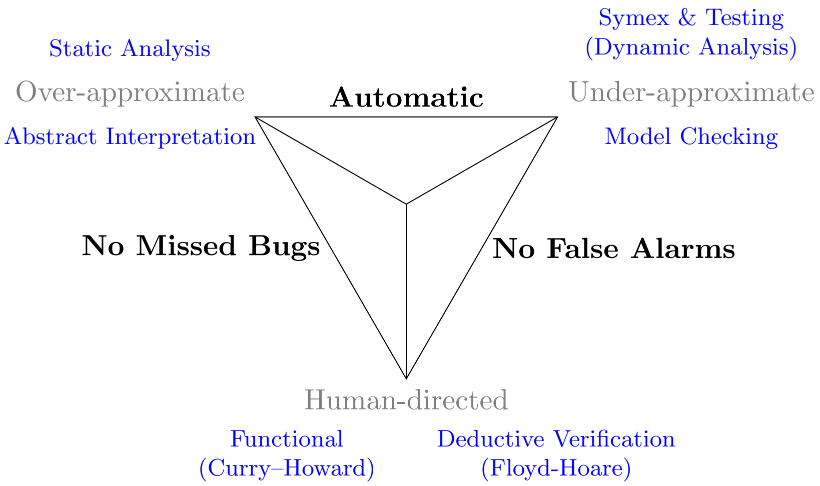


Figure 2.1: The Triangle Model of Software Verification adapted from the paper by M. Brain and D. Kroening [6].

and climbing to the top of a pyramid while they are trying to tackle their weakness. We will discuss each of them with some highlighted techniques as examples and the use of software verification in the following subsections.

2.1.1 Software Verification Techniques

2.1.1.1 Static Analysis

It refers to a family of techniques determining the run-time properties of a program without execution [17]. With an approximation of the fixpoint collecting semantics, it can indicate run-time errors automatically at compilation time without any code instrumentation or user interaction [16, 46]. However, the capability of static analysers can be various depended on techniques implemented. Sometimes predefined facts or patterns are often used for efficiently showing the absence of simple errors, which can be found in compiler optimisation, such as Clang[1], GCC[2]. Besides that, there are two well-known static analysis techniques, *Abstract Interpretation* and *Model Checking*, that are widely researched and capable of analysing large size software systems with minimal manual effort.

Model Checking is a formal technique to examine each reachable states whether a correctness property holds without execution and generate a counterexample once a violation exists, which will be discussed in detail in Section 3.2.

[1] A C language family front-end for LLVM: http://clang.llvm.org/

[2] The GNU Compiler Collection: https://gcc.gnu.org/

Abstract Interpretation is a theory of semantics approximation to predict the program behaviour by using an abstract domain instead of the actual domain of the execution [17]. It guarantees a full coverage of the analysis by over-approximating the possible execution paths, therefore no bugs will be missed [15]. However, the accuracy of the analysis relies on the approximation, which becomes a limitation to static analysis. Due to the undecidability of static analysis, it is not possible to come up with an approximation that produces no false positive and no false negative detections, which means no false alarms [17]. Moreover, it is difficult to generate counterexamples as the precision loss during the analysis which is a significant trade-off for efficiency.

Despite the weakness above, the performance and the scalability of abstract interpretation are beneficial to the industry as some abstract interpretation-based analysers, such as **ASTRÉE**, have been successfully applied to the verification of safety-critical software, such as flight and rocket control system. Moreover, a study conducted by P. Cousot et al. [16] further uses ASTRÉE as an example to claims that such a difficulty of abstract interpretation has been overcome by domain-specific static analysers.

2.1.1.2 Human-directed Analysis

As its name suggested, refers to techniques relied on human assistance to determine the software correctness by using mathematical proof. **Deductive Verification** is an example of such technique to verify the correctness of a program by expressing them into a set of mathematical statements, such that the validity can then be proven by deduction [18,22]. In contrast to static analysis and model checking, it can be the most precise way to model the program behaviour without abstracting any data structures and instructions as expressed in the triangle model in Figure 2.1. However, the price of such precision is relatively high as it requires a huge amount of human effort to construct the inductive arguments as well as computing resources to achieve the verification. This technique is intended to trade off automation for precision by heavily relying on human interaction during the theorem proving process. Hence, it is capable of verifying the functional correctness, but, hardly to be scaled up for large size system [15].

2.1.1.3 Dynamic Analysis

It refers to techniques that test and evaluate a program by running it. It can derive the correctness properties that hold for a certain number of executions, but not for all executions, which is in contrast to static analysis mentioned above [3]. Although it is not able to prove that a particular property holds, violations of properties can still be detected as well as provide some useful information about the program behaviour. This technique is easier and cheaper to carry out comparing with other formal verification techniques as it generally requires less knowledge than formal method does. Therefore, various testing techniques exist and commonly be used in the software industry.

Concolic testing is a hybrid software verification technique combining symbolic static analysis and concrete dynamic analysis, which is able to generate test cases automatically and to examine execution paths exhaustively. M. Kim et al. [27] conducted an experiment on concolic testing to analyse the multi-sector read operation for a flash memory and summarised its advantages and weaknesses compared to model checking techniques. Concolic testing algorithm in general consists of five steps: 1) Instrumentation, 2) Concrete execution, 3) Symbolic execution, 4) Deciding the next execution path, and 5) Choosing the next input values. As the symbolic execution performs along the concrete execution path, it can prevent any false alarms be created. Once the path formulas are not able to be solved by a constraint solver, some symbolic constraints will be simplified by replacing some of the symbolic values with concrete value. This may lead to an incomplete coverage. The experimental result showed it had a better applicability and lower memory usage. However, the study also reflected that concolic testing is a time-consuming method as time will be wasted on generating invalid test cases in a complex environment model. Concolic testing can be a good choice for finding bugs, but not for demonstrating program correctness. Comparatively, model checking generally provides high accuracy and a better performance of the verification than concolic testing, which is essential for verifying critical software.

2.1.2 The Use of Software Verification

Software verification is commonly used for two main purposes: 1) hunting bugs, and 2) proving the correctness of a program. Below are several case studies making good use of the model checking, which is the technique we focused on in this project:

2.1.2.1 Hunting bugs

M. Voelter et al. [45] took several well-known bugs as examples to show how the modular extension of the C programming language can improve software security. One of the examples is the Heartbleed bug in OpenSSL TLS mentioned in Section 1.1. By constructing a simple harness and a message with a non-deterministic data buffer, CBMC is able to identify the failure in a short period. This is a great demonstration of the importance of formal verification in software development.

Another remarkable case studies presented by Y. Kim and M. Kim [28] has successfully detected four hidden bugs in an embedded software, lazybox ls, using CBMC. This study highlighted the pivot of loops analysis, determining the minimum iterations to exit a loop, to avoid false negative detections in practice. As an accurate unwinding loop bound can minimise the chances of missing bugs, state explosion, and execution timeout. The experiment also showed the effectiveness of using such model checking technique and its limitation on the loop analysis.

There is a case study conducted by J. Yang et al. [50] demonstrated a systematical way to model check three widely-used and heavily-tested file systems. Two classes of checks are suggested: 1) generic checks for those general properties should hold at any point, and 2) consistency checks for the functional properties that the file system specified. Concerning the complexity of such file systems, it is impractical to be fully tested in the traditional way as there could be an exponential number of test cases for checking the recovery mechanism. Model checking make use of several state-reducing techniques, which makes it capable of exploring such vast state spaces efficiently. As a result, 32 serious bugs are found during the experiment.

2.1.2.2 Proving correctness

An experience report presented by R. Metta [34] described their experience in verifying the correctness of a 200-line implementation of Cyclic Redundancy Check. The study indicated that sometimes the specification and verification process could hardly be automated and required intense human efforts even for such a small code bases. It is hard to scale up if such a program containing huge loops. The authors further suggested a bottom-up approach to verify the correctness of each function independently through CBMC first, and then to verify the correctness of the loop through an induction manually using CBMC.

A. Jääskelüi et al. [23] verified the implementation of the 2003 voting scheme in a systematic capability 3 level shutdown system through model checking. The experiment is conducted in two stages: 1) refine and translate the requirements into pseudo-code directly for a preliminary verification, and then move on to 2) verify the actual implementation of the voting scheme. The study concluded in several suggestions for using model checking techniques. Beginning with some simple test cases can help to understand how to give precise and effective assertions before jumping into those complicated properties. Fault seeding techniques are recommended for inexperienced users to check the correctness of assertions by introducing violation to the corresponding properties.

Another study conducted by B. Beckert et al. [4] verified the functional and relational properties of voting rules, and demonstrated the effectiveness of using bounded model checking and deductive verification. Moreover, it also showed that the verification effort can be greatly reduced by using symmetry properties.

F. Werner and D. Farago [48] investigated CBMC to the domain of wireless sensor networks and proved that it is capable of performing automatic verification on a system about 21000 LoC. It showed that the scalability could be greatly improved by the proposed abstractions and simplification heuristics, with the –slice-formula option enabled. The study further concluded that CBMC with the abstractions and heuristics is powerful enough to verify large scale programs.

2.1.2.3 Other applications

There are several studies showing that model checking is applied to various usages, such as embedded software verification [7, 14], cross-platform verification [47], sequentialisation-based verification [19], fault localisation [20], equivalence checking [31], cyber-physical system co-verification [52], and test-vector generation [2].

2.2 Researched approaches on using Bounded Model Checking

The use of bounded model checking techniques have been presented and widely researched over the last couple of decades. In this section, we discuss some experiences of applying model checking on software security audit and assurance. It is important to learn from the successful cases from others to observe how the model checking is being used effectively and what kind of contribution to did it make to the software development process and the overall quality.

Abstract Testing is a software testing approach proposed by F. Merz et al. [33], which focuses on the relation between requirements and the corresponding test cases to replace the traditional test cases by an abstract one. Each abstract test case is derived from the requirements and formulated into assumption and assertion statements on the source code level, which is often a one to one relation. Assumption and assertion statements are the constraints encoding the preconditions and postconditions respectively. Setting up these constraints for the testing environment with respect to non-deterministic values, rather using concrete values. Hence, an abstract test case is possible to represent a large number of concrete test cases, which can remarkably simplify the test cases generation and maintenance process. The experimental study showed that abstract testing is more capable and efficient to discover underlying bugs than traditional software testing. Moreover, the performance of CBMC is promising with less than 20s runtime on average for an abstract test case, which is suitable for the agile-like development process. Once there is a change of requirement, it is easily and directly visible to the corresponding abstract test cases.

Besides the incomplete coverage of conventional testing stated above, M. Kim et al. [24–26] also mentioned that once a violation is detected, it still require a large amount of human effort to locate where the violation occurred by replaying the scenario and tracing the execution step-by-step. Hence, model checking techniques are well-suited to overcoming these weaknesses of the conventional testing method through exhaustive analyses. This study applied SAT-based bounded model checking technique to verify the functional correctness and increase the reliability of the device driver software for Samsung's OneNAND$^{\text{TM}}$ flash memory. In order to extract the code level properties, the study also suggested a top-down approach to identify the properties from several design documents, such as software requirement specifications, architecture design specifications, and detailed design specifications. It demonstrated a practical way on how the functional correctness properties can be extracted from the real world environment. By replacing the conventional testing with constraint-based exhaustive testing, which is the same as abstract testing above, CBMC achieved great success in the experiment that discovered some bugs including incomplete exceptions handling and logical bugs. This promising result shows that model checking techniques are mature enough to be applied on verifying an industrial level software.

Property-based code slicing Despite the fact that model checking is a powerful technique, it often requires more knowledge and cost than testing. To reduce the cost while maintaining comprehensiveness, M. Park et al. [36] proposed an approach that consists of three strategies: 1) property-based environment generation, 2) property-based abstraction, and 3) collaborative verification using model checking and testing. The study applied it to verify the Trampoline operating system. The result showed that the approach could reduce the verification cost by scaling down the target code, and to simplify the analysis process by localising the verification activity.

False positive elimination Software verification based on abstract interpretation is scalable for verifying industrial level code bases but imprecise. Many spurious errors are generated as the abstract interpretation over-approximated the execution traces than the program actually can perform. Manual investigation is needed to review each error, such manual efforts are costly to the development process. To overcome this problem, H. Post et al. [38] and P. Darke et al. [29] integrated model checking to reduce the number of false positive generated by the static analysis tools. The experimental result shows that 69% of warnings have been successfully removed by CBMC.

Another study by T. Muske et al. [43] pointed out the above approach could involve numerous verifications, and further proposes an approach consisting of three techniques to achieve a faster false positives elimination. By avoiding redundant equivalent assertions, results in a 60% accelerated false positives elimination.

Aspect-oriented programming technique M. Ludwich and A. Frohlich [32] introduced an approach to formally verify both *functional correctness* and *safety properties* of embedded operating system components. Such components are supposed to be shifted between different domains, therefore, the implementation of formal verification must also be domain-independent. The study proposed a corresponding strategy that consists of three main steps: 1) creating contracts for each component that to be verified , 2) implementing such components with the corresponding contracts accordingly, and 3) performing software bounded model checking to verify whether the implementation of components respects to their contracts. The benefits of keeping the specification and implementation close to each other is that any violations in the implementation could be detected at the early stage of development. However, this may introduce an extra run-time overhead to its instances. Hence, those contracts should be partially eliminated from the implementations beyond the verification stage. The experiment made used of aspect-oriented programming technique to achieve this purpose, and further facilitate a modular verification by isolating different components. Note that components isolation is not suitable for verifying a monolithic design. More importantly, the experiment also demonstrated that the functional correctness and safety properties can be verified by the contracts captured from requirements and those properties generated by the model checker respectively. This approach is a kind of white box testing and is more suitable to carry out at the early stage or during the development.

Falsification-Driven Verification A. Groce et al. [21] stated the idea of a ”verification successful” in model checking or even theorem proving could be a sign of insufficient verification properties. In order to gain more knowledge of the meaning behind ”successful” results, the study proposed a falsification-driven methodology that adapts mutation testing to show the weakness of current specification. Mutate both the harness and the program with a small syntactic change assuming a good test suite should be able to detect a bug introduced by such a change. Hence, the mutation kill rate can act as a measurement unit of the accuracy rate and correctness of a harness. As a better harness should able to detect more errors from the implementation. This approach is useful for ensuring the quality of harnesses, thus a more reliable verification. However, it requires many manual efforts to verify and examine the mutants while some of them can be semantically equivalent. It would be suitable for developers who are not familiar with formal verification but intend to verify a system.

Configuration Lifting is a specification analysis technique presented by H. Post and C. Sinz [37], which transforms all variants into a meta-program and facilitates the analysis in three different domains: 1) inconsistencies within a feature model, 2) inconsistencies between the feature model and their implementations, and also 3) the coverage of run-time errors in all product variants. The study demonstrated the technique on verifying Linux kernel and device driver, which successfully found two bugs in the kernel configuration system. The experimental result showed that this technique is able to apply on a large size system included more than 4600 features and model checking on the meta-program provides a significant speed-up compared with traditional enumeration based analysis.

Hybrid Approach L. Cordeiro et al. [13] propose a semi-formal verification approach combined dynamic and static verification to stress and cover the state spaces of embedded software exhaustively, in order to improve the coverage and reduces the verification time. This approach allows developers to reason both the functional and temporal properties quantitatively to guarantee the timeliness and correctness of the design. The experimental result shows that model checkers have limitation to specify more complex temporal properties and state space explosion problem.

Chapter 3

Background

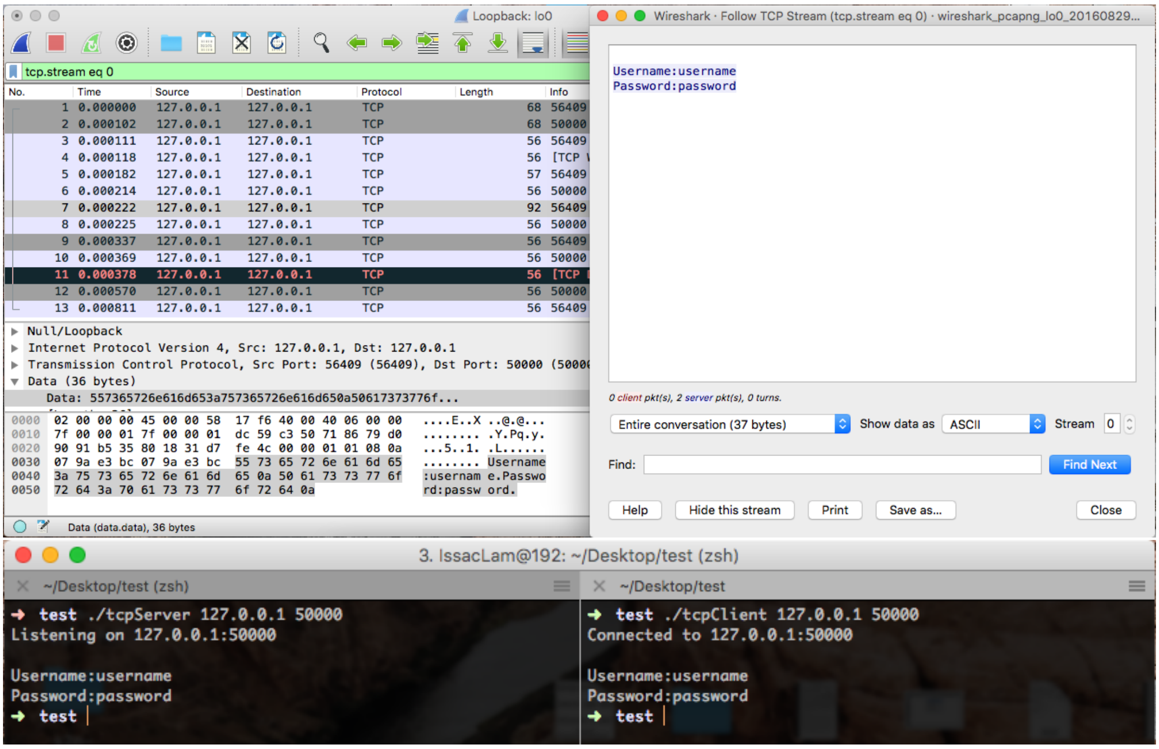
3.1 Overview of s2n

s2n [1,41] is an open-source implementation of the TLS/SSL protocols released in late June 2015 by the Amazon Security Labs. Due to the severe impact caused by the Heartbleed bug, Amazon designed a light-weight TLS/SSL implementation with the concern of security and reviewability. Concerning the complexity of the TLS/SSL implementations, s2n is only around 6,000 lines of code while the one of OpenSSL requires more than 70,000 lines. In general, more code means more chance to have bugs and harder to be reviewed. Therefore, Amazon has taken these advantages to make s2n easier to be verified, as well as making sure the correctness and safety of the implementation. To achieve this, Amazon has already invested a huge amount of effort and numerous resources on testing the implementation including several external security evaluations and penetration tests.

In terms of TLS/SSL, it refers to the Transport Layer Security (TLS) and its predecessor, Secure Sockets Layer (SSL), which is a set of cryptographic protocols for providing privacy and data integrity between network communications. According to its characteristics, TLS/SSL is responsible for establishing secure connections and managing those sessions keys used in the connections, which has been classified as a protocol sit between the Transport Layer and Application Layer (L3 and L4 of the TCP/IP model or L4 and L7 of the OSI model respectively). In recent years, it has been widely adopted by numerous application layer protocols, such as SMTPS, IMAPS, FTPS. The most popular usage of TLS/SSL must be HyperText Transfer Protocol Secure (HTTPS), which is used to secure web applications, especially for those in financial usage, for examples, online banking, stock trading and online shopping. Since TLS/SSL plays an essential role in securing the entire network communications, the implementation of it would, of course, be a critical component of network infrastructure.

To show the importance of TLS/SSL protocol, we would like to demonstrate how the privacy and data integrity are being provided by using two showcases below. They simulates the user login behaviour in the network communication between the server and a client: 1) without the TLS/SSL protection or 2) under the TLS/SSL protection. Suppose the communications can be eavesdropped by any third party using Wireshark[1], a network protocol analyser.

Without the protection of TLS/SSL: All contents of the communication are simply sent in plaintext, including the user login information. Everyone on the same network is able to extract the information by capturing the network traffic. As we can see clearly from the top half of Figure 3.1 below, without the protection of TLS/SSL, there is no privacy and data integrity provided unless the web services implement a secure protocol by themselves but the level of security provided is usually limited.



```
Loopback: lo0
tcp.stream eq 0
No.  Time      Source     Destination  Protocol  Length  Info
  1  0.000000  127.0.0.1  127.0.0.1    TCP           68  56409
  2  0.000102  127.0.0.1  127.0.0.1    TCP           68  50000
  3  0.000111  127.0.0.1  127.0.0.1    TCP           56  56409
  4  0.000118  127.0.0.1  127.0.0.1    TCP           56  [TCP
  5  0.000182  127.0.0.1  127.0.0.1    TCP           57  56409
  6  0.000214  127.0.0.1  127.0.0.1    TCP           56  50000
  7  0.000222  127.0.0.1  127.0.0.1    TCP           92  56409
  8  0.000225  127.0.0.1  127.0.0.1    TCP           56  50000
  9  0.000337  127.0.0.1  127.0.0.1    TCP           56  56409
 10  0.000369  127.0.0.1  127.0.0.1    TCP           56  50000
 11  0.000378  127.0.0.1  127.0.0.1    TCP           56  [TCP
 12  0.000570  127.0.0.1  127.0.0.1    TCP           56  50000
 13  0.000811  127.0.0.1  127.0.0.1    TCP           56  56409
▶ Null/Loopback
▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▶ Transmission Control Protocol, Src Port: 56409 (56409), Dst Port: 50000 (5000
▼ Data (36 bytes)
    Data: 557365726e616d653a757365726e616d650a50617373776f...
0000  02 00 00 00 45 00 00 58  17 f6 40 00 40 06 00 00   ....E..X ..@.@...
0010  7f 00 00 01 7f 00 00 01  dc 59 c3 50 71 86 79 d0   ........ .Y.Pq.y.
0020  90 91 b5 35 80 18 31 d7  fe 4c 00 00 01 01 08 0a   ...5..1. .L......
0030  07 9a e3 bc 07 9a e3 bc  55 73 65 72 6e 61 6d 65   ........ Username
0040  3a 75 73 65 72 6e 61 6d  65 0a 50 61 73 73 77 6f   :usernam e.Passwo
0050  72 64 3a 70 61 73 73 77  6f 72 64 0a              rd:passw ord.
Data (data.data), 36 bytes
```

```
Wireshark · Follow TCP Stream (tcp.stream eq 0) · wireshark_pcapng_lo0_20160829...
Username:username
Password:password

0 client pkt(s), 2 server pkt(s), 0 turns.
Entire conversation (37 bytes)   Show data as ASCII   Stream 0
Find:
```

```
3. IssacLam@192: ~/Desktop/test (zsh)
~/Desktop/test (zsh)
➜  test ./tcpServer 127.0.0.1 50000
Listening on 127.0.0.1:50000

Username:username
Password:password
➜  test
```

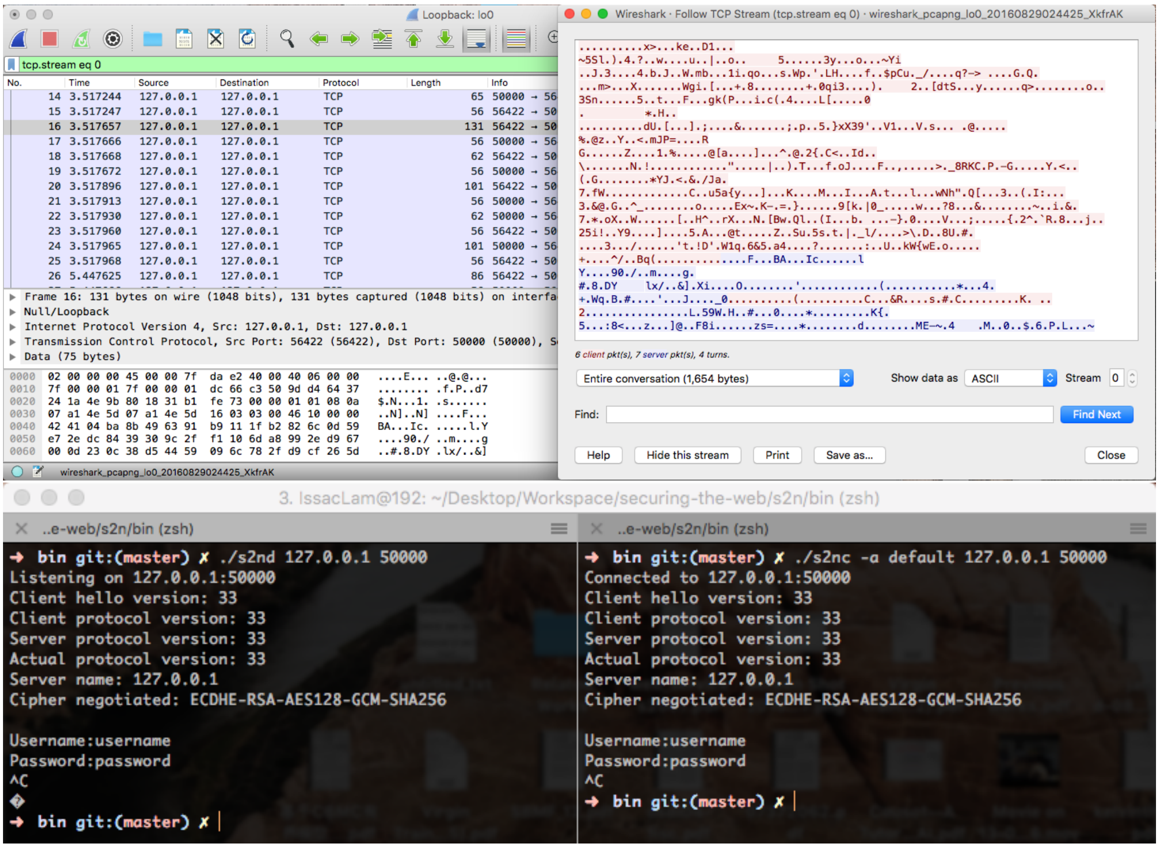
```
~/Desktop/test
➜  test ./tcpClient 127.0.0.1 50000
Connected to 127.0.0.1:50000

Username:username
Password:password
➜  test
```

Figure 3.1: The screenshot of the network traffic captures and the result of the communication

With the protection of TLS/SSL: The entire communication are encrypted as shown at the top half of Figure 3.2 and the eavesdropper cannot steal any information from the

[1] The Wireshark Website: https://www.wireshark.org/

traffic captures anymore. As a result, the privacy and data integrity are guaranteed by the implementation of TLS/SSL protocol, which is s2n in this case.



```
No.      Time      Source      Destination   Protocol   Length   Info
    14 3.517244  127.0.0.1   127.0.0.1     TCP             65  50000 → 56
    15 3.517247  127.0.0.1   127.0.0.1     TCP             56  56422 → 50
    16 3.517657  127.0.0.1   127.0.0.1     TCP            131  56422 → 50
    17 3.517666  127.0.0.1   127.0.0.1     TCP             56  50000 → 56
    18 3.517668  127.0.0.1   127.0.0.1     TCP             62  56422 → 50
    19 3.517672  127.0.0.1   127.0.0.1     TCP             56  50000 → 56
    20 3.517896  127.0.0.1   127.0.0.1     TCP            101  56422 → 50
    21 3.517913  127.0.0.1   127.0.0.1     TCP             56  50000 → 56
    22 3.517930  127.0.0.1   127.0.0.1     TCP             62  50000 → 56
    23 3.517960  127.0.0.1   127.0.0.1     TCP             56  56422 → 50
    24 3.517965  127.0.0.1   127.0.0.1     TCP            101  50000 → 56
    25 3.517968  127.0.0.1   127.0.0.1     TCP             56  56422 → 50
    26 5.447625  127.0.0.1   127.0.0.1     TCP             86  56422 → 50
▶ Frame 16: 131 bytes on wire (1048 bits), 131 bytes captured (1048 bits) on interfa
▶ Null/Loopback
▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▶ Transmission Control Protocol, Src Port: 56422 (56422), Dst Port: 50000 (50000), S
▶ Data (75 bytes)
0000  02 00 00 00 45 00 00 7f  da e2 40 00 40 06 00 00   ....E... ..@.@...
0010  7f 00 00 01 7f 00 00 01  dc 66 c3 50 9d d4 64 37   ........ .f.P..d7
0020  24 1a 4e 9b 80 18 31 b1  fe 73 00 00 01 01 08 0a   $.N...1. .s......
0030  07 a1 4e 5d 07 a1 4e 5d  16 03 03 00 46 10 00 00   ..N]..N] ....F...
0040  42 41 04 ba 8b 49 63 91  b9 11 1f b2 82 6c 0d 59   BA...Ic. .....l.Y
0050  e7 2e dc 84 39 30 9c 2f  f1 10 6d a8 99 2e d9 67   ....90./ ..m....g
0060  00 0d 23 0c 38 d5 44 59  09 6c 78 2f d9 cf 26 5d   ..#.8.DY .lx/..&]
wireshark_pcapng_lo0_20160829024425_XkfrAK
```

Wireshark · Follow TCP Stream (tcp.stream eq 0) · wireshark_pcapng_lo0_20160829024425_XkfrAK

```
..........x>...ke..D1...
~5Sl.).4.?..w....u..|..o..      5......3y...o...~Yi
..J.3....4.b.J..W.mb...1i.qo...s.Wp.'.LH....f..$pCu._/....q?-> ....G.Q.
...m>...X.......Wgi.[...+.8........+.0qi3....).     2..[dtS...y......q>........o..
3Sn......5..t...F...gk(P...i.c(.4....L[.....0
.          *.H..
..........dU.[...].;....&.......;.p..5.}xX39'..V1...V.s... .@.....
%.@z..Y..<.mJP=....R
G......Z....1.%.....@[a....]...^.@.2{.C<..Id..
\.......N.!............".....|..).T...f.oJ....F..,.....>._8RKC.P.-G.....Y.<..
(.G........*YJ.<.&./Ja.
7.fW............C..u5a{y...]...K....M...I...A.t...l...wNh".Q[...3..(.I:...
3.&@.G..^_........o.....Ex~.K-.=.}......9[k.|0_.....w...?8...&........~..i.&.
7.*.oX..W......[..H^..rX...N.[Bw.Ql..(I...b. ...-}.0....V...;.....{.2^.`R.8...j..
25i!..Y9....]....5.A...@t.....Z..Su.5s.t.|._l/....>\.D..8U.#.
....3.../......'t.!D'.W1q.6&5.a4....?.......:..U..kW{wE.o.....
+....^/..Bq(..............F...BA...Ic......l
Y....90./..m....g.
#.8.DY     lx/..&].Xi....0........'...........(..........*...4.
+.Wq.B.#....'...J...._0..........(..........C...&R....s.#.C.........K. ..
2...............L.59W.H..#...0....*.........K{.
5...:8<...z...]@..F8i......zs=....*........d........ME-~.4    .M..0..$.6.P.L...~
```

6 client pkt(s), 7 server pkt(s), 4 turns.

Entire conversation (1,654 bytes) Show data as: ASCII Stream: 0

Find:

```
➜ bin git:(master) ✗ ./s2nd 127.0.0.1 50000
Listening on 127.0.0.1:50000
Client hello version: 33
Client protocol version: 33
Server protocol version: 33
Actual protocol version: 33
Server name: 127.0.0.1
Cipher negotiated: ECDHE-RSA-AES128-GCM-SHA256

Username:username
Password:password
^C
�
➜ bin git:(master) ✗
```

```
➜ bin git:(master) ✗ ./s2nc -a default 127.0.0.1 50000
Connected to 127.0.0.1:50000
Client hello version: 33
Client protocol version: 33
Server protocol version: 33
Actual protocol version: 33
Server name: 127.0.0.1
Cipher negotiated: ECDHE-RSA-AES128-GCM-SHA256

Username:username
Password:password
^C
➜ bin git:(master) ✗
```

Figure 3.2: The screenshot of the network traffic captures and the result of the encrypted communication

Concerning the level of security provided completely depends on the safety and correctness of the implementation. Therefore, there is a definite necessity to verify the s2n implementation by using a well-proven and reliable verification technique, *bounded model checking*.

3.2 Software Model Checking

In order to discuss the capability of bounded model checking, we would like to begin with the fundamental of it, which is *model checking*. It is an algorithmic technique, based on graph theory, automata theory and logic, [44] to establish the correctness of the model of a state system with respect to given properties, which are formalised from the specification [11,12]. A model checking problem can be defined as whether all reachable states s of a given model M satisfies the given property φ, which is expressed by a temporal logic formula, such that $M, s \models \varphi$. This concept can be fully adapted to software verification with the consideration of M and φ referring to a model of a program and its specification respectively. While a program consists of a set of *states* describing its memory state at a particular time frame and a set of *transitions* describing how the program evolves from one state to another. Model checking will first compute all the reachable states, then by traversing exhaustively, the algorithm will be able to tell whether the property p holds at all states if there is a finite number of states. Once a violation is found on any execution path, a *counterexample* can be generated accordingly. Otherwise, *witnesses* are presented to confirm that the given property φ is true. These concepts are all illustrated in Figure 3.3 below.

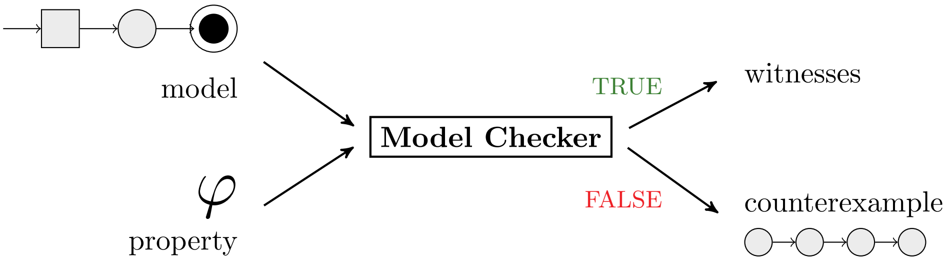


Figure 3.3: The Model Checking Structure adapted from E. Clarke [11]

Model checking is capable of verifying both *safety* and *liveness* properties of software in a slightly different manner [17]. Safety means nothing bad happens in the program execution, such as arithmetic overflow, buffer overflow, or NULL dereference, which can be verified by proving the unreachability of these states. Liveness means something good eventually happens. This can be proved by showing that a program fulfils its functional specifications or it guarantees a termination at the end. A program can be claimed as verified when all the given properties are satisfied by the given model. This technique is precise as all reachable states have been examined. However, the number of states can grow exponentially along with the complexity of the program [5, 8]. The state explosion problem becomes a major weakness of such techniques for analysing a medium sized code bases [51]. In the history of model checking, several techniques have been applied in order to increase the capacity of the exploration states. For examples, some efficient graph traversal techniques, the symbolic representation, and Binary Decision Diagrams (BDDs). However, the size of BDDs also can grow exponentially and therefore it becomes a bottleneck of verification. Since the efficiency and capability are restricted, it will not be enough to carry out a full verification to meet the industrial needs.

3.3 Bounded Model Checking

Bounded Model Checking (BMC) based on Boolean Satisfiability (SAT) is a complementary technique to model checking with an upper bound of the number for exploration state introduced by A. Biere et al. in 1999 [5]. Although it does not overcome the entire complexity weakness of model checking, it is more capable of verifying different circumstances comparing with using the BDD-based techniques. In BMC, only those execution paths within the user-defined bound are covered, properties beyond the bound will not be verified. Instead of proving "*all reachable states satisfies the property* is true directly, the formula can also be checked in its negation by showing "*there exists a reachable state violates the property*" is false [8]. Therefore, the concept of BMC is to check the negation of given property φ up to a given depth k, in order to find a witness for showing such a violation exists [39], as shown in Figure 3.4 below.

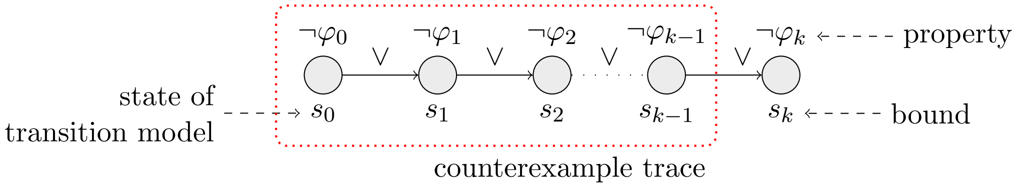


Figure 3.4: The idea behind Bound Model Checking adapted from H. Rocha et al. [39]

In addition, by unrolling the transition operations for k times and then combining those properties to form a propositional formula, such a boolean formula can be reduced to a SAT problem and hence it can be solved by a SAT solver [5, 40]. The formula is satisfiable as long as the property is violated by a trace of length k. The advantages of replacing BDDs with SAT include the avoidance of the exponential growth of space in BDD, and the application of smart depth-first search, which can reduce the memory consumption when using breadth-first search in BDDs.

In reality, there is a possibility of missing bugs in the verification caused by the condition of given bound not covering all the possible execution paths. This reflects the paramount importance of choosing an appropriate bound to minimise such a possibility. In practice, it is suggested to begin with a smaller length k, and then extend the length gradually if there is no error found. More importantly, the verification result is promising since a counterexample can locate exactly where the violation is and all bugs within the bound are guaranteed to be found. Even if the property does not allow for proving the correctness, BMC is still useful and reliable for finding bugs. Because of the above improvements and this impressive feature, BMC based on SAT has been successfully used in the industry. Moreover, V. D'Silva et al. [17] further concluded that BMC is the best technique to find shallow bugs comparing with static analysis and model checking as a complete counterexample trace is provided once a bug is found.

A practical experiment conducted by J. Toman et al. [42] combines exhaustive test generation and bounded model checking technique in order to detect memory safety errors. A bounded verifier, **CRUST**[2], was developed, which relies on **CBMC** for performing automated analysis. It will first generate a set of testing drivers for the relevant APIs and then convert the driver codes from Rust to C for bounded model checking. With the help of exhaustive test generation, large input spaces can be explored efficiently, and multiple test drivers can be generated for covering the input spaces as many as possible. By evaluating data structures from Rust standard library, CRUST has successfully detected some underlying memory safety bugs in it.

In contrast to static analysis, model checking is more precise in proving complex properties since an under-approximated model is used in the verification. This is supported by the experimental result conducted by K. Vorobyov and P. Krishnan [46]. The experiment compares the verification result of static program analysis with the one of model checking by applying **Parfait**[3] and CBMC respectively on some benchmarked code bases. By configuring CBMC to focus on particular types of error, it showed a high accuracy rate (97% overall true positive detections, and found about 19% more bugs), whereas only 77% with Parfait. Moreover, both of them reported a 0% false positive detections, which is common for model checking and a specific design to minimise it for static program analysis. However, model checking is much more expensive with respect to computation time and memory usage, as it reached the memory limit in the experiment. The study further concluded that: 1) an insufficient unwinding was one of the reasons for false negative detections of CBMC, and 2) scalability can be a problem for model checking on a large size code base, such as an operating system.

A counterexample-guided abstraction refinement (CEGAR) based technique, which was presented by A. Yeolekar and D. Unadkat [51], utilises dynamic analysis to overcome the scalability limitation of model checking. Dynamic inference is used to guess invariants and refine the abstraction from spurious counterexamples, thus a better precision of the abstraction and an accelerated loop refinement are achieved in their experiment.

[2] CRUST Github repository: https://github.com/maidsafe/crust

[3] Parfait Github repository: https://github.com/performancecopilot/parfait

3.4 C Bounded Model Checker (CBMC)

CBMC [4], despite its name, is a bound model checking tool for the formal verification of ANSI-C programs using SAT solvers developed by D. Kroening et al. [10]. It aims at reasoning about the safety properties of low-level ANSI-C implementation as much safety-critical software are written in such low-level languages.

3.4.1 Getting Ready for verifying a program

In order to analyse a given C/C++ code, CBMC will first reduce the model checking problem to a validity of bit vector equation problem by translating the program statement into Static Single Assignment (SSA) form [9, 10]. The translation consists of three main procedures described as follows:

Program translation Suppose the ANSI-C program has been preprocessed already, the program will be translated into a control-flow graph, which is only consisting of `if`, `while`, `goto` statements and simple assignments without any side effects, and then be analysed by using standard compiler technique.

Loop unwinding In order to model the computation up to a given depth, loops are necessary to be unwound into a fixed number of iterations. This can be done by repeating the loop body with certain amounts of copies. While each copy is guarded by an `if` statement with the same condition as the loop, it is prepared for the case that fewer iterations are required for the loop. In addition, an assertion with the negated condition can be added after the last copy to ensure that the loop does not require more iterations by using `--unwinding-assertions` option in CBMC. This is essential to show whether the unwinding bound is large enough to model the program behaviour. After the program translation, there are three different loop constructions remain including `while` statements, recursive function calls, and `goto` statements. We use `while` loop as an example to demonstrate the concept of loop unwinding below:

[4] The CBMC Homepage: http://www.cprover.org/cbmc/

```
                                    if(cond) {
                                        instr; /** 1st copy **/
                                        if(cond) {
                                            instr; /** 2nd copy **/
                                                ...
          while(cond) {                         if(cond) {
              instr;        →                       instr; /** kth copy **/
          }
                                                    /** unwinding assertion **/
                                                    assert(!cond)];
                                                }
                                            }
                                        }
```

Figure 3.5: Loop unwinding for k times with unwinding assertion adapted from CPROVER tutorials [5]

Variable Renaming After the previous operations, the program only containing `if` statements, assignments, assertions, `goto` instructions and labels is ready to be transformed into SSA form by using pointer analysis. The transformation is demonstrated by the simple example below:

```
x=x+y;                    x1=x0+y0;
if(x!=1)                  if(x1!=1)
    x=2;                      x2=2;
else              →       else                 →
    x++;                      x3=x1+1;

assert(x<=3);             x4=(x1!=1)?x2:x3;
                          assert(x4<=3);
```

$$\begin{aligned} C &:= \texttt{x}_1\texttt{=x}_0\texttt{+y}_0 \wedge \\ &\quad\; \texttt{x}_2\texttt{=2} \wedge \\ &\quad\; \texttt{x}_3\texttt{=x}_1\texttt{+1} \wedge \\ &\quad\; \texttt{x}_4\texttt{=(x}_1\texttt{!=1)?x}_2\texttt{:x}_3 \\ P &:= \texttt{x}_4 \leq 3 \end{aligned}$$

Figure 3.6: SSA form transformation adapted from [9, 10]

The procedure above renames all variables with states to ensure that each variable is fresh and will only be assigned once. As a result, the program can be viewed as a set of constraints and two bit-vector equations, constraints C and properties P, can be produced accordingly. As a result, the property can now be verified by converting $C \wedge \neg P$ into Conjunctive Normal Form (CNF) through a SAT solver. Once a violation of the given property is found, the equation is identified as *satisfiable*; otherwise, the property holds when the equation is *unsatisfiable*.

3.4.2 Assumption & Assertion

As mentioned above, CBMC considers the verification conditions as a pair of bit-vector equations, constraints C and properties P, which are specified by using *assumption* (`__CPROVER_assume()`)

[5] The CPROVER manual: http://www.cprover.org/cprover-manual/

and *assertion* (`__CPROVER_assert()`) statement respectively. The `__CPROVER_assert()` statement takes a Boolean condition, which is an ANSI-C logic expression, and a string description as arguments. CBMC will check whether the given condition holds for all executions. The description is useful for locating the violated assertion. The `__CPROVER_assuem()` statement will also take a Boolean expression, which is used to restrict the program traces exploration. The program trace examination will be aborted while the given assumption is false.

3.4.3 Non-Determinism

Non-deterministic choice functions are supported in CBMC by declaring `nondet_` as the prefix of their names and it is useful for modelling user inputs. The range of generated values depends on the return type of the function and the restrictions that are given by the corresponding assumptions.

3.4.4 Automatic Verification

Concerning the verification of safety properties, CBMC takes care of a wide range of program behaviour regarding such properties, including dynamic allocations using `malloc()` and `free()`, buffer overflow, pointer safety, arithmetic overflow, user-defined assertions. Moreover, by enabling CBMC with different options, the verification condition generator of CBMC will automatically generate the safety conditions accordingly, resulted in a full automation in the verifications above. The options of supported built-in safety check are as follows:

```
--div-by-zero-check                      --signed-overflow-check
--bounds-check                           --unsigned-overflow-check
--pointer-check                          --float-overflow-check
--memory-leak-check
```

Because of the advantages of automation and the verification capability, CBMC has been widely used in various applications, such as error explanation, embedded programs verification, security bugs detection in Windows binaries, etc.

Chapter 4

Hypothesis and Approach

4.1 Hypothesis

In software verification, we are always interested in the question that:

"Does the *System* meet its *Specification*?"

System In terms of the system, we will focus on Amazon's s2n implementation. Since s2n represents a critical part of the network infrastructure, providing security and privacy for network communications, as explained in Section 3.1, it is necessary to ensure the correctness and safety of its implementation. More importantly, Amazon has made a significant amount of effort and invested numerous resources on testing the implementation of s2n.

Specification The specification defines a wide range of properties describing the behaviours of a system, such as safety properties, functional properties, non-functional properties, etc. In this project, we aim at checking the memory safety properties of the s2n implementation, which include:

1. Array Bounds (Buffer Overflows)
2. Pointer Safety (Null Dereference)
3. Memory Leak
4. Divided by Zero Check
5. Arithmetic Overflow
    - Unsigned Integer Overflow
    - Signed Integer Overflow
    - Float Overflow

Other properties can be checked by testing, but memory safety bugs are easily missed by such techniques due to the insufficient coverage, as mentioned in Section 2.1.1.3.

Verification Tool CBMC is a bounded model checker designed for analysing the low-level properties of C/C++ implementations. It supports the full set of ANSI-C features, and it is also able to generate error traces on violations, which provides us with detailed evidence to support our hypothesis. Moreover, CBMC has been well developed and continuously maintained. It is a reliable tool for software verification.

The following two questions will be the realms to investigate in this project:
Is it possible to

1. Find bugs in a heavily tested critical network infrastructure (s2n) using CBMC?
2. Show the memory safety of the implementation?

4.2 Verification Approach

4.2.1 Identifying the Verification Scope

Since s2n is not a stand-alone software package, it relies on the *libcrypto* library, which is a general-purpose cryptography library, provided by OpenSSL. To study the structure of s2n, we construct a simple *bash* program, shown in Appendix F.3, to extract all the external function calls used in the s2n implementation, then examine and categorise each of them. The result, listed in Appendix B, reveals that there are 112 different external function calls in total. Besides the *libcrypto* library, s2n also stands on the standard C99 library and the system libraries. The architecture of s2n is outlined in Figure 4.1 below.

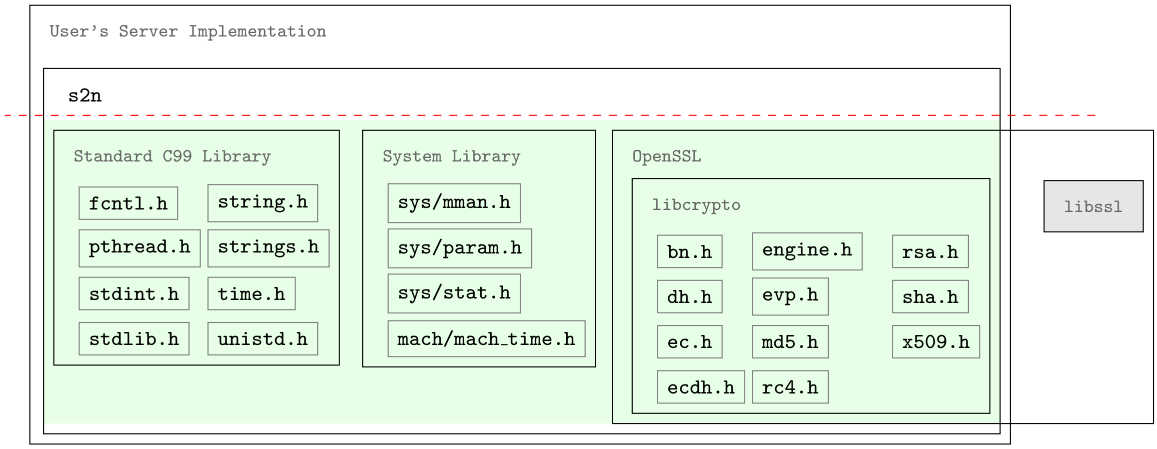


Figure 4.1: The Architecture of s2n

Since CBMC requires the actual coding to perform the syntactic analysis, it is impractical to include all the implementations of lower-layer libraries for the verification as this will enlarge the exploration states and go beyond our scope. In addition, most of the system libraries and programming language libraries come along as compiled binary files instead of source files. In order to remain focused on the implementation of s2n, we create a set of stubs to model the behaviour of those underlying software packages and libraries (Highlighted in Green in Figure 4.1) for which s2n is standing on. The details of stubbing are described in Section 4.2.5.

4.2.2 Choosing the Suitable Approach

Regarding the verification of a software package, most of the research papers has not documented their verification approach or just simply verified a stand-alone function to demonstrate their proposed techniques. Hence, there is no conventional approach that could be adopted to this project. We believe that there are numerous verification approaches on various verification purposes and implementations of the targeted software. During our studies, we have considered and tried out three different approaches. We will discuss each of them as follows and we found that *Bottom-up module-based approach* is the best fit for the s2n verification.

Function-based approach This approach verifies every single function independently, which does not take the function usage contracts, such as the execution environment and the order of function calls, into account. It is only application for verifying stand-alone functions that are supposed to handle all exceptions themselves. However, this approach will produce a large number of false alarms while being applied to verify an object-oriented software, such as s2n. We constructed a *bash* program to go through all the CBMC built-in checkers over every s2n function, and the verification results of using this approach is listed as follows:

Property checked	BC [1]	DBZC [2]	PC [3]	MLC [4]	SOC [5]	UOC [6]	FOC [7]	TO [8]
No. of alarms	27	9	227	3	32	96	3	4

[1] Array Bounds Check
[2] Divided By Zero Check
[3] Pointer Check
[4] Memory Leak Check
[5] Signed Overflow Check
[6] Unsigned Overflow Check
[7] Float Overflow Check
[8] Timeout (5 hours)

Table 4.1: Number of false alarms generated by using function-based approach

According to the s2n development guide, the caller functions are responsible for providing all well-defined function parameters and doing all error handling. These numbers of false alarms are generated regardless of the function usage contracts. Therefore, this approach is not applicable and not reliable in this case as it requires a significant amount of human effort in reviewing each report.

Top-down module-based approach This approach verifies the software package from the API level using some verification harnesses with respect to the function usage contracts. With the verification environments being the same as the realistic usage scenarios, it can minimise both the number of false alarms being generated and the verification effort for examining all of them. In the case, a counterexample indicates the absolute certainty of the presence of a bug and no false alarms would be generated. However, the workload can be extremely heavy for verifying such a bulky function call, which would easily lead to a state explosion and an indefinite verification. For example, we constructed a verification harness for verifying the initialisation of a `s2n_connection`, which involves more than 50 different sub-function calls. The harness and its execution command are as shown below:

```
s2n_connection
```

```
s2n-harness/top-down/s2n_connection_harness.c
```

```c
int main(){
    struct s2n_connection *conn;
    conn = s2n_connection_new(S2N_SERVER);
    return 0;
}
```

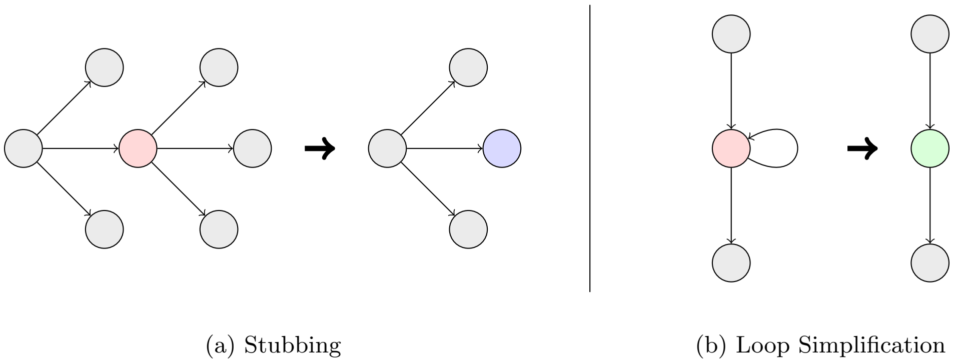
Listing 1: A harness for verifying `s2n_connection` with the execution command

This process runs for more than 5 hours for state space exploration and is even killed by the terminal. Thus, this is also not an applicable approach for verifying s2n.

Bottom-up module-based approach This approach divides the software package into modules and then verifies one module at a time from the bottom level with respect to the function usage contracts. Once a module is verified, it can be replaced by an over-approximated model in order to simplify the complexity of its caller functions. Hence, the verification is able to scale up for the entire software package. This is the approach we will use for our verification. Regarding the over-approximation, we will use two different ways to reduce the functions' complexity as shown in Figure 4.2 below. The details of stubbing and loop simplification are described in Section 4.2.5 and Section 4.2.7 respectively.

4.2.3 Capturing the Function Usage Contracts

After choosing a proper approach, we need to classify those functions into modules in order to study their usage contracts and then be able to prepare appropriate harnesses for the verification. This is different from most of the research studies in that those studies already have a given set of function properties whereas we have to extract them from the limited documentations. By scanning all the functions in s2n, we found 318 functions implemented



(a) Stubbing

(b) Loop Simplification

Figure 4.2: The two possible ways used for over-approximation.

in 50 different C implementation files. However, there is no official document describing the usage contract of each of them. Only a brief usage guide [1] for the API functions is available on the s2n Github repository. Although s2n is designed with readability in mind and most of the usages are guessable from the function name, the dependencies between functions are not clear, which is an obstacle to our verification. Thanks to **Doxygen**[2], a documentation generator, which allows us to have a detailed overview of all the functions and data structures in s2n.

Besides analysing the dependency graphs generated by Doxygen, 31 unit test cases that come along with the s2n repository are beneficial to the study of the function usage contracts as they are simulating the real scenario for using those functions. We can also transform those unit test cases into our verification harnesses by introducing non-determinism to such environments in order to extend the verification coverage, which is similar to the abstract testing approach in [33]. The harnesses we created in this verification are available in Appendix E.

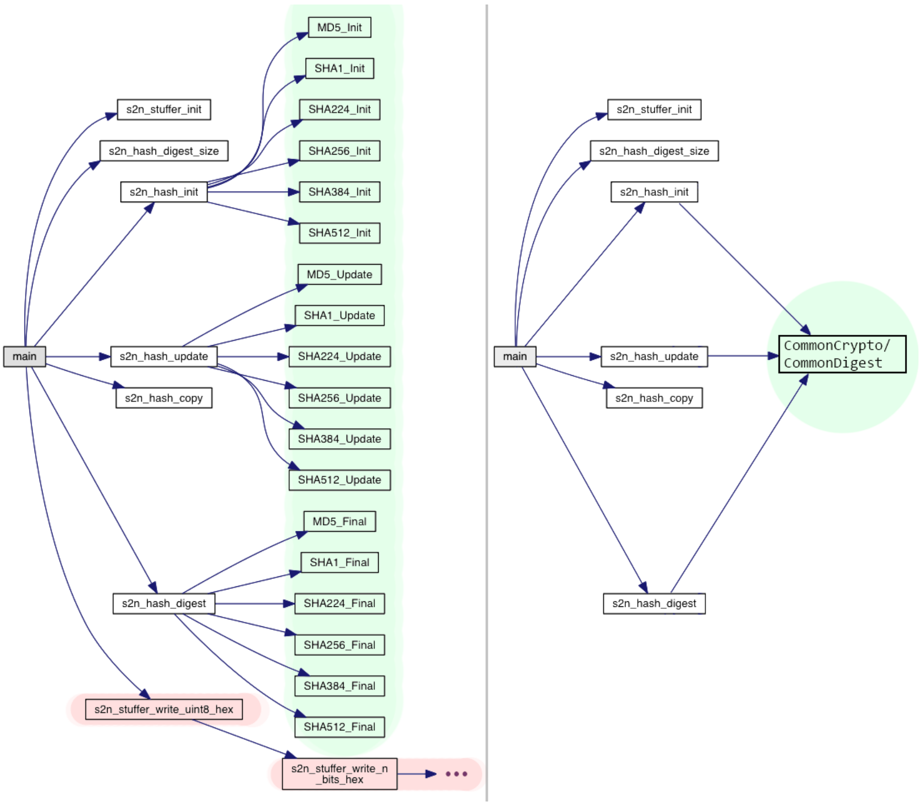
4.2.4 Function Call Graphs Analysis

As mentioned in the previous section, a set of dependency graphs, for example, function call graphs, are generated by using Doxygen. A clear overview of all functions helps us determine which function to begin with the verification and which functions are attainable or possible to be replaced by stubs. We use the analysis of `s2n_hash_test.c` as an example, which is shown in Figure 4.3 below. The left-hand side of the figure shows the original function call graph of the `s2n_hash_test`. The green highlight indicates that those external function calls, such as `SHA1_Init`, `SHA1_Update`, `SHA1_Final`, provided by the Common Crypto library of Mac OS X, are considered as beyond our verification scope. Hence, they

[1] s2n Usage Guide: https://github.com/awslabs/s2n/blob/master/docs/USAGE-GUIDE.md

[2] The Doxygen Main Page: http://www.stack.nl/ dimitri/doxygen/

are replaced by a corresponding model of `CommonCrypto/CommonDigest` implementations. Moreover, the red highlight indicates that the functions calls can be removed by non-determinism. As a result, the function dependencies are simplified as shown in the right-hand side of the figure.



(a) The function call graph of s2n_hash_test. (b) The simplified dependency of s2n_hash_harness.

Figure 4.3: An example of function call graph analysis

4.2.5 Stubbing

A stub method is used to model the behaviour of a targeted function, like its input and output behaviour, without performing the actual computation and it would return a simple and valid result, but may not necessarily be correct. It is useful for simplifying the verification processes. We continue the case above to demonstrate how the models of functions, `SHA1_Init()` and `SHA1_Final()`, are created. Note that the following implementations of `SHA1_Init()` and `SHA1_Final()` are referenced from the document of Apple Open Source[3].

Consider the model of `SHA1_Init()` shown in Listing 2. The assignments of the variable `context->state` (line 4-9) are not in our concern, and these can be replaced by a non-

[3] The Open Source Apple: http://opensource.apple.com/

```
s2n-lib/CommonCrypto/CommonDigest.c
1  /* SHA1 initialization. Begins a SHA1 operation, writing a new context. */
2  int CC_SHA1_Init(SHA1_CTX *context)
3  {
4      // context->bcount[0] = context->bcount[1] = 0; context->count = 0;
5  
6      /* Load magic initialization constants.  */
7      // context->state[0] = 0x67452301UL; context->state[1] = 0xefcdab89UL;
8      // context->state[2] = 0x98badcfeUL; context->state[3] = 0x10325476UL;
9      // context->state[4] = 0xc3d2e1f0UL;
10 
11     /* ensure the pointer of c is valid */
12     __CPROVER_assert(c != NULL, "ERROR: CC_SHA1_Init c is null");
13 
14     /* assign a non-deterministic value to the variable c. */
15     *c = nondet_CC_SHA1_CTX();
16 
17     return 1;
18 }
```

Listing 2: The model of `CC_SHA1_Init()` implementation

deterministic assignment (line 15). The only interest of the function is the memory safety of the pointer of the input parameter, therefore an assertion is needed for checking the validity of the pointer (line 12). Note that the definition of a valid pointer is always debatable as sometimes it can refer to a pointer that must be pointing to a particular range of memory addresses. In this case, a valid pointer means that it is not pointing to `NULL`.

Consider the model of `SHA1_Final()` shown in Listing 3. Similar to modelling of the `CC_SHA1_Init()` above, we are not interested in the actual computation of SHA1 (line 5-21), thus the method calls of `Encode()` and `SHA1Update()` can be ignored. However, we have to ensure the pointer of `digest` and `context` are valid (line 24-25) and also model the value change to the variables (line 28). Concerning the array bounds safety that no memory access should exceed the allocated boundary of an array, we can model this behaviour by accessing the last byte of the array (line 31).

4.2.6 Loop Bound Analysis

An appropriate bound is essential to bounded model checking, since it determines the number of loop unwinding, as mentioned in Section 3.4.1, as well as the coverage of the analysis. In this project, we are not only aiming at finding bugs, but also showing the memory safety of the s2n implementation. Therefore, the Loop Unwinding Bound (LUB) is defined by the maximum number of loop iterations in order to make sure the verification can cover the entire execution paths. Normally, such LUBs can be detected by CBMC automatically via a syntactic analysis of loop structures, for example, the left side of Table 4.2. In the s2n-lib/CommonCrypto/CommonDigest.c

```c
/* SHA1 finalization. Ends an SHA1 message-digest operation, writing the
 * the message digest and zeroizing the context. */
int CC_SHA1_Final(void *digest, SHA1_CTX *context)
{
    // unsigned char bits[8];
    // u_int32_t index = (context->bcount[1] >> 3) & 0x3f;

    /* Save number of bits */
    // Encode(bits, context->bcount, 8);

    /* Pad out to 56 mod 64. */
    // SHA1Update(context, PADDING, ((index < 56) ? 56 : 120) - index);

    /* Append length (before padding) */
    // SHA1Update(context, bits, 8);

    /* Store state in digest */
    // Encode(digest, context->state, 20);

    /* Zeroize sensitive information. */
    // memset(context, 0, sizeof (*context));

    /* ensure the pointers are valid */
    __CPROVER_assert(md != NULL, "ERROR: CC_SHA1_Final md is null");
    __CPROVER_assert(c != NULL, "ERROR: CC_SHA1_Final c is null");

    /* try to write something to the variable */
    *c = nondet_CC_SHA1_CTX();

    /* try to access the output pointer */
    md[SHA1_DIGEST_SIZE - 1];

    return 1;
}
```

Listing 3: The model of `CC_SHA1_Final()` implementation

case that the syntactic analysis fails, the right side of Table 4.2, an iterative algorithm can be used to determine a suitable bound. At a first attempt, guessing the number of loop iterations with the help of the unwinding assertions, the `--unwinding-assertions` option of CBMC, a violation can indicate any excess of the guessed bound, hence a larger one is needed [10,17]. Note that, according to the CPROVER manual, the number of unwindings is defined by the number of backjumps. In the example below, the condition `i<10` has at most 11 times of evaluation before terminating the loop. Therefore, the LUB of the loop should be 11 rather than 10.

```c
int depth=10;
for(int i=0; i<depth; i++) {
    /* do something */
}
```

(a) A bound can be detected

```c
int depth=nondet_int();
__CPROVER_assume(depth>0 && depth<10);
for(int i=0; i<depth; i++) {
    /* do something */
}
```

(b) A bound cannot be detected

Table 4.2: An example of loop bound detection

Apart from guessing the bound of loops, a more precise loop analysis can be performed manually. By using the `--show-loops` option in CBMC, all loops in the included source files will be listed with a unique identifier. Then, each LUB can be determined by studying the function usage over the function caller graphs and they can be specified for some particular loops by using the `--unwindset` option with the loop ID, as shown below:

```
Loop main.0:
  file demo.c line 5 function main

$ cbmc --unwindset main.0:10 demo.c
```

4.2.7 Simplifying the Computations

As functions can have dynamic loop bounds, such as string manipulation functions, their complexities are one of the major obstacles in scaling up our verification. To verify the memory safety of these functions, it is not necessary to check all iterations. Only the last iteration is needed since it is able to verify all the memory safety properties listed in Section 4.1. Consider the following situations:

1. If a violation of any one of them can occur in any iteration, it can also happen and be detected in the last iteration.

2. If a buffer overflow only occurs in the last iteration of a monotone function, it will not happen and can not be detected in other iterations.

Therefore, those iterations simply repeating the same computation are unnecessary to the verification except for the last iteration. In this case, we can simplify the loop by removing those redundant iterations with respect to the original control flow and error handling.

For example, the function `s2n_stuffer_read_base64()` converts a base64 string into ASCII string 3 bytes at a time. The detailed implementation of the function is shown in Listing 4 below. Regarding the memory safety verification described above, the computation of the conversion and the correctness of the output values are not our concern here. Therefore, the concrete value of the inverse array and those assignments that would not affect the control flow can be omitted (line 28, 54, 58, 64 & 69). Moreover, the `do` ... `while()` loop is simply repeating the conversion until the end of the input string, which is redundant for verifying the memory safety (line 36). We skip those iterations by using the `s2n_stuffer_skip_read()` provided in the same module (line 40). As a result, we have modelled the function in a loop-free manner, which can significantly reduce the workload in the loop unwinding process.

Furthermore, during the verification, we observed that s2n's functions involve a huge number of memory manipulation function calls, such as `memset()`, `memcpy()`, `memcmp()`. In the same sense, we further simplify these functions in the built-in ANSI-C library provided by CBMC. The simplified models are listed in Listing 5 below. Note that after modifying any CBMC built-in libraries, it is necessary to rebuild CBMC in order to make it support the changes.

s2n-lib/s2n/s2n_stuffer_base64.c

```c
static const uint8_t b64_inverse[256];

int s2n_stuffer_read_base64(struct s2n_stuffer *stuffer, struct s2n_stuffer *out)
{
    uint8_t value1, value2, value3, value4;
    uint8_t pad[4]; int bytes_this_round = 3;
    struct s2n_blob o = {.data = pad,.size = sizeof(pad) };

//    do {
        if (s2n_stuffer_data_available(stuffer) < 4) { return 0; }

        /* skip the unnecessary computations */
        s2n_stuffer_skip_read(stuffer, s2n_stuffer_data_available(stuffer)/4*4);

        GUARD(s2n_stuffer_read(stuffer, &o));

        value1 = b64_inverse[o.data[0]]; value2 = b64_inverse[o.data[1]];
        value3 = b64_inverse[o.data[2]]; value4 = b64_inverse[o.data[3]];

        if (value1 == 255) { stuffer->read_cursor -= 4; return 0; }

        if (value1 == 64 || value2 == 64 || value2 == 255 || value3 == 255 || value4 == 255) { S2N_ERROR(S2N_ERR_INVALID_BASE64); }

        if (o.data[2] == '=') {
            if (o.data[3] != '=' || value2 & 0x0f) { S2N_ERROR(S2N_ERR_INVALID_BASE64); }
            bytes_this_round = 1;
//            value3 = 0; value4 = 0;
        } else if (o.data[3] == '=') {
            if (value3 & 0x03) { S2N_ERROR(S2N_ERR_INVALID_BASE64); }
            bytes_this_round = 2;
//            value4 = 0;
        }

        uint8_t c; GUARD(s2n_stuffer_write_uint8(out, c));

        if (bytes_this_round > 1) {
//            c = ((value2 << 4) & 0xf0) | ((value3 >> 2) & 0x0f);
            GUARD(s2n_stuffer_write_uint8(out, c));
        }

        if (bytes_this_round > 2) {
//            c = ((value3 << 6) & 0xc0) | (value4 & 0x3f);
            GUARD(s2n_stuffer_write_uint8(out, c));
        }
//    } while (bytes_this_round == 3);
    return 0;
}
```

Listing 4: The simplified model for `s2n_stuffer_read_base64()`

```
cbmc/src/ansi-c/library/string.c
```

```c
inline void *memcpy(void *dst, const void *src, size_t n)
{
    /* detailed implementation is omitted */
//    for(__CPROVER_size_t i=0; i<n ; i++)
//        ((char *)dst)[i]=((const char *)src)[i];

    __CPROVER_assert(n > 0, "memcpy n is least or equal to zero");
    ((char *)dst)[n-1];
    ((const char *)src)[n-1];
      return dst;
}

inline void *memset(void *s, int c, size_t n)
{
    /* detailed implementation is omitted */
    //  char *sp=s;
    //  for(__CPROVER_size_t i=0; i<n ; i++) sp[i]=c;

    __CPROVER_assert(n > 0, "memset n is least or equal to zero");
    ((char *)s)[n-1];
    return s;
}

inline int memcmp(const void *s1, const void *s2, size_t n)
{
    /* detailed implementation is omitted */
    _Bool res;
    //  int res=0;
    //  const unsigned char *sc1=s1, *sc2=s2;
    //  for(; n!=0; n--)
    //  {
    //    res = (*sc1++) - (*sc2++);
    //    if (res != 0)
    //      return res;
    //  }

     __CPROVER_assert(n > 0, "memcmp n is least or equal to zero");
    ((const char *)s1)[n-1];
    ((const char *)s2)[n-1];
    return (res)? 0 : 1;
}
```

Listing 5: The implementation of `strlen()` provided by CBMC

Chapter 5

Experiment Setup

5.1 Tool Setup and Configuration

5.1.1 CBMC Installation

Please follow the instructions to install CBMC in Mac OS X environment. For other environments, please refer to the instructions on the CBMC Github repository.

```
# -1. Install the Xcode from App store
# 0. Install C/C++ compiler, Flex and Bison, and GNU make
$ xcode-select --install

# 1. Download the latest version of CBMC from CBMC Github repository
$ git clone https://github.com/diffblue/cbmc.git

# 2. Move to cbmc/src directory
$ cd cbmc/src

# 3. Download the source code of MiniSat2
$ make minisat2-download

# 4. Download the latest version of libzip from libzip Github repository
$ git clone https://github.com/nih-at/libzip.git

# 5. Download latest version of zlib from zlib Github repository
$ git clone https://github.com/madler/zlib.git

# 6. Build CBMC
$ make

# 7. Move the CBMC executable to /usr/local/bin directory
$ mv cbmc/cbmc /usr/local/bin

# 8. Now you are ready to use cbmc
$ cbmc -h
```

Listing 6: The instructions to install CBMC in Mac OS X environment.

5.1.2 s2n Installation

Please follow the instructions to install s2n in Mac OS X environment.

```
# 1. Download the latest version of s2n from s2n Github repository
$ git clone https://github.com/awslabs/s2n.git

# 2. Move to s2n/libcrypto-build directory
$ cd s2n/libcrypto-build

# 3. Building s2n with OpenSSL-1.0.2 and download the latest version of OpenSSL
$ curl -LO https://www.openssl.org/source/openssl-1.0.2-latest.tar.gz
$ tar -xzvf openssl-1.0.2-latest.tar.gz

# 4. Build openssl's libcrypto
$ cd openssl-1.0.2h
$ ./Configure darwin64-x86_64-cc -fPIC no-shared no-libunbound no-gmp         \
              no-jpake no-krb5 no-md2 no-rc5 no-rfc3779 no-sctp no-ssl-trace  \
              no-store no-zlib no-hw no-mdc2 no-seed no-idea                  \
              enable-ec-nist_64_gcc_128 no-camellia no-bf no-ripemd no-dsa    \
              no-ssl2 no-ssl3 no-capieng -DSSL_FORBID_ENULL -DOPENSSL_NO_DTLS1\
              -DOPENSSL_NO_HEARTBEATS --prefix=`pwd`/../../libcrypto-root/
$ make depend
$ make
$ make install

# 5. Move to the main s2n directory
$ cd ../../

# 6. Build s2n
$ make

# 7. Relink the dynamic library for the s2n's examples
$ install_name_tool -change libs2n.dylib @executable_path/../lib/libs2n.dylib bin/s2nc
$ install_name_tool -change libs2n.dylib @executable_path/../lib/libs2n.dylib bin/s2nd
```

Listing 7: The instructions to install s2n in Mac OS X environment.

5.2 Execution Environment

The machine configuration for the experiment is shown below:

Item	Value
Processor	2.5 GHz Intel Core i7
Memory	16 GB 1600 MHz DDR3
Disk	500GB Apple SSD SM0512G Media
OS	OS X 10.11.5
CBMC Version	5.4
CBMC Github Commit ID	d031ccc0f8ec82495310c3268066dcde5bc18c59
s2n Github Commit ID	42d9daf1813e60b9c8ad235096309486e9fbfa05
OpenSSL	1.0.2i-dev

Table 5.1: Machine Configuration

5.3 CBMC Execution Command Construction

After setting up the environment and creating the verification harnesses, we have to construct the CBMC execution commands for each individual harness. We will demonstrate how a command can be constructed by going through one of the cases (`s2n_hmac_harness.c`) in detail:

First, we can run CBMC on the prepared harness directly, and it will report any missing header files that are required as shown in the follows:

```
$ cbmc ./s2n-harness/module/s2n_hmac_harness.c

CBMC version 5.4 64-bit x86_64 macos
Parsing ./s2n-harness/module/s2n_hmac_harness.c
file ./s2n-harness/module/s2n_hmac_harness.c line 16:
 ↪   ./s2n-harness/module/s2n_hmac_harness.c:16:10: fatal error: 'error/s2n_errno.h'
 ↪   file not found
file ./s2n-harness/module/s2n_hmac_harness.c line 16: #include "error/s2n_errno.h"
file ./s2n-harness/module/s2n_hmac_harness.c line 16:          ^
file ./s2n-harness/module/s2n_hmac_harness.c line 16: 1 error generated.
file ./s2n-harness/module/s2n_hmac_harness.c line 16: GCC preprocessing failed
PARSING ERROR
```

Listing 8: Extracting the missing header file

By repeating the above step, we can extract all the required header files and then include each of them into the execution command by using the `-I` option. Next, we can use the help of the `grep` command to capture any missing function body as shown below:

```
$ cbmc -I ./s2n -I ./s2n/api -I ./s2n/libcrypto-build/openssl/include
 ↪  ./s2n-harness/module/s2n_hmac_harness.c --unwind 1 2>&1 | grep "WARNING"

**** WARNING: no body for function s2n_hmac_digest_size
**** WARNING: no body for function s2n_hmac_init
**** WARNING: no body for function s2n_hmac_update
**** WARNING: no body for function s2n_hmac_digest
**** WARNING: no body for function s2n_hmac_copy
```

Listing 9: Extracting the missing function bodies

Since CBMC will display the warning messages in the `stderr` channel, therefore we have to redirect the warning messages into the `stdout` channel in order to capture them by using the `grep` command. Moreover, we also use `--unwind 1` to limit the number of iterations as one so that it can also cover those functions called in a loop. The above steps are repeated until all the required source files are included in the execution command. Then, we can extract all the loops involved in this execution with their loop IDs given by CBMC as shown in the following:

```
$ cbmc -I ./s2n -I ./s2n/api -I ./s2n/libcrypto-build/openssl/include
 ↪  ./s2n-harness/module/s2n_hmac_harness.c ./s2n/crypto/s2n_hmac.c
 ↪  ./s2n/crypto/s2n_hash.c ./s2n/stuffer/s2n_stuffer.c
 ↪  ./s2n-lib/CommonCrypto/CommonDigest.c --unwind 1 | grep "loop"

...
Not unwinding loop s2n_sslv3_mac_init.0 iteration 1 (1 max) file
 ↪  ./s2n/crypto/s2n_hmac.c line 51 function s2n_sslv3_mac_init thread 0
Not unwinding loop s2n_sslv3_mac_init.1 iteration 1 (1 max) file
 ↪  ./s2n/crypto/s2n_hmac.c line 59 function s2n_sslv3_mac_init thread 0
Not unwinding loop s2n_hmac_init.6 iteration 1 (1 max) file ./s2n/crypto/s2n_hmac.c
 ↪  line 156 function s2n_hmac_init thread 0
Not unwinding loop s2n_hmac_init.7 iteration 1 (1 max) file ./s2n/crypto/s2n_hmac.c
 ↪  line 159 function s2n_hmac_init thread 0
Not unwinding loop s2n_hmac_init.8 iteration 1 (1 max) file ./s2n/crypto/s2n_hmac.c
 ↪  line 166 function s2n_hmac_init thread 0
Not unwinding loop s2n_sslv3_mac_digest.0 iteration 1 (1 max) file
 ↪  ./s2n/crypto/s2n_hmac.c line 73 function s2n_sslv3_mac_digest thread 0
```

Listing 10: Extracting all the loops involved the execution

With the above information, we can perform loop bounds analysis for all the involved loops as explained in Section 4.2.6. After all the LUBs have been determined, each of them can be specified in the execution command by using the `--unwindset` option. Then, we can test the termination of the constructed execution command as shown as follows:

```
$ cbmc -I ./s2n -I ./s2n/api -I ./s2n/libcrypto-build/openssl/include
 ↪  ./s2n-harness/module/s2n_hmac_harness.c ./s2n/crypto/s2n_hmac.c
 ↪  ./s2n/crypto/s2n_hash.c ./s2n/stuffer/s2n_stuffer.c
 ↪  ./s2n-lib/CommonCrypto/CommonDigest.c --unwindset
 ↪  s2n_sslv3_mac_init.0:49,s2n_sslv3_mac_init.1:49,s2n_hmac_init.6:129,s2n_hmac_init.7:129,s2n_hmac_init.8:12

CBMC version 5.4 64-bit x86_64 macos
Parsing ./s2n-harness/module/s2n_hmac_harness.c
...
size of program expression: 12072 steps
simple slicing removed 545 assignments
Generated 306 VCC(s), 256 remaining after simplification
Passing problem to propositional reduction
converting SSA
Running propositional reduction
Post-processing
Solving with MiniSAT 2.2.1 with simplifier
1340716 variables, 5211423 clauses
SAT checker: instance is UNSATISFIABLE
Runtime decision procedure: 12.026s

** Results:
[__builtin___memcpy_chk.assertion.1] memcpy n is least or equal to zero: SUCCESS
...
[CC_SHA512_Final.assertion.2] ERROR: CC_SHA512_Final c is null: SUCCESS

** 0 of 38 failed (1 iteration)
VERIFICATION SUCCESSFUL
```

Listing 11: Testing the termination of the constructed execution command

The above verification result shows that the CBMC execution terminates normally, which means all involved loops are well bounded by the given LUBs. Moreover, the absence of warning messages means all the required files, both header files and source files, are included in the execution. Hence, the CBMC execution command for `s2n_hmac_harness.c` has been successfully constructed.

5.4 CBMC Parameter Setting

After the CBMC execution commands have been constructed, we can perform the verification by enabling those built-in checkers in CBMC according to our verification interest. Since we are interested in checking all the memory safety properties that are supported by CBMC as listed in Section 4.1. We will simply enable all the built-in checkers for our verification. In order to prevent an indefinite verification, set 5 hours as the timeout for the experiments. However, CBMC does not support timeout feature. Hence, we have created a helper tool to enable all the built-in checkers on verification, terminate the process on timeout and export the verification result into an output file. The usage and the implementation of the helper tool are listed below:

The usage of verifier.sh

```
$ ./verifier.sh [output filename] [cbmc execution command]
```

helper-tools/verifier.sh

```bash
#!/bin/bash
buildInChecks="--bounds-check --div-by-zero-check --pointer-check --memory-leak-check --signed-overflow-check --unsigned-overflow-check --float-overflow-check"

filename="$1"

prefix="s2n-verification-result/module-based/"
dir="$prefix"
mkdir $dir;
shift

if [ "$1" != "cbmc" ]
then
    echo "ERROR: missing output file name";
    exit 1
fi

start_ns=$(gdate +%s%N);
result=$(gtimeout 5h $@ $buildInChecks);
end_ns=$(gdate +%s%N);
elapsed_ms=$(((end_ns - start_ns) / 1000000));
echo $elapsed_ms;

echo "> gtimeout 1h $@ $buildInChecks" > $dir/$filename.txt;
echo "$result" >> $dir/$filename.txt;
echo "Time taken: $elapsed_ms" >> $dir/$filename.txt
```

Listing 12: The script usage and the implementation of the helper tool for assisting our verification

Chapter 6

Experimental Evaluation

6.1 Achievement

After the verification, we have successfully found two arithmetic overflow bugs in the s2n implementation and thus provided the corresponding solutions. The details are described in Section 6.2 . A pull request[1] was submitted to the s2n Github repository and accepted by the Amazon s2n developers. They showed interest in our findings and wish to include the use of our techniques into their project. The details of the pull request and the conversation are shown in the screenshot in Figure 6.1 below:

6.2 Detected Failures

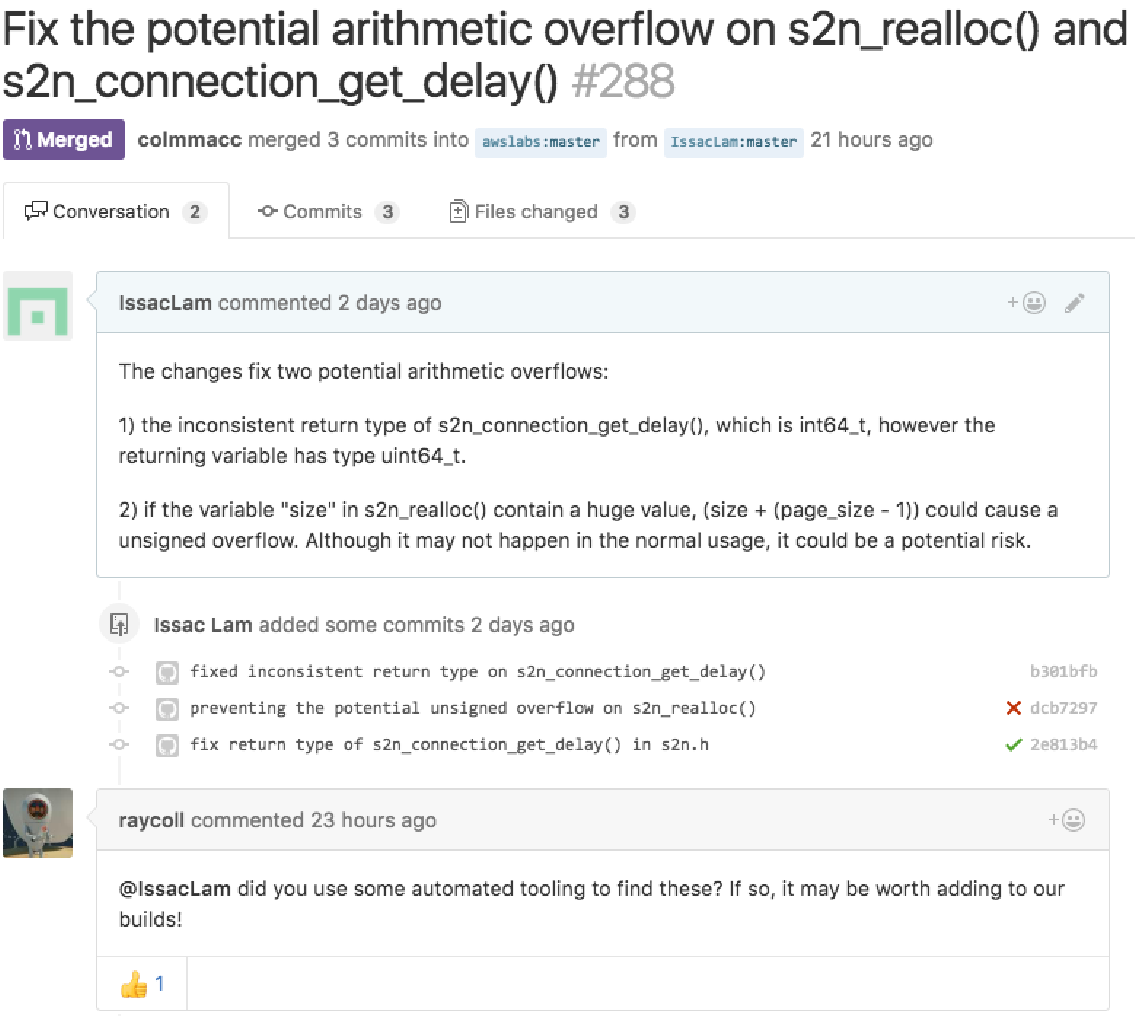
In this verification, CBMC reports 14 failures in seven different harnesses with all built-in checkers enabled, as mentioned in Section 5.4. By tracing the cause of the failures, we have discovered the similarity between the causes of failures and they can be concluded into four key issues: two bugs, one expected behaviour and one false alarm. Each key issue will be discussed in detail below:

6.2.1 Unsigned Overflow on Memory Allocation

An unsigned overflow failure is detected in the implementation of `s2n_realloc()` (line 56). The error trace and the implementation are listed in Listing 13 below:

Regarding the implementation of `s2n_realloc()`, we can see that there is no upper bounds check on the variable `size` before line 56. Since the value of `size` is given by the caller functions and it can be any possible value of type `uint32_t`, an unsigned overflow will occur at `(size + (page_size - 1))` operation (line 56) when the value of `size` is greater than `4,294,963,201`. Note that the value of `page_size` is commonly 4096 bytes (4KB),

[1] https://github.com/awslabs/s2n/pull/288



Fix the potential arithmetic overflow on s2n_realloc() and s2n_connection_get_delay() #288

colmmacc merged 3 commits into `awslabs:master` from `IssacLam:master` 21 hours ago

IssacLam commented 2 days ago

The changes fix two potential arithmetic overflows:

1) the inconsistent return type of s2n_connection_get_delay(), which is int64_t, however the returning variable has type uint64_t.

2) if the variable "size" in s2n_realloc() contain a huge value, (size + (page_size - 1)) could cause a unsigned overflow. Although it may not happen in the normal usage, it could be a potential risk.

Issac Lam added some commits 2 days ago

```
fixed inconsistent return type on s2n_connection_get_delay()       b301bfb
preventing the potential unsigned overflow on s2n_realloc()      ✗ dcb7297
fix return type of s2n_connection_get_delay() in s2n.h           ✓ 2e813b4
```

raycoll commented 23 hours ago

@IssacLam did you use some automated tooling to find these? If so, it may be worth adding to our builds!

Figure 6.1: The screen capture of the accepted pull request and related conversations

```
detected-failures/s2n_mem_harness-unsigned_overflow.txt
1   /* detail error trace is omitted */
2
3   State 70 file ./s2n/utils/s2n_mem.c line 46 function s2n_alloc thread 0
4   ----------------------------------------------------
5     size=4294963204u (11111111111111111111000000000100)
6
7   State 73 file ./s2n/utils/s2n_mem.c line 61 function s2n_realloc thread 0
8   ----------------------------------------------------
9     allocate=0u (00000000000000000000000000000000)
10
11  Violated property:
12    file ./s2n/utils/s2n_mem.c line 61 function s2n_realloc
13    arithmetic overflow on signed to unsigned type conversion in (unsigned int)(page_size
     ↪   * (((signed long int)size + (page_size - (signed long int)1)) / page_size))
14    ((-1l + page_size + (signed long int)size) / page_size) * page_size >= 0l && !(((-1l
     ↪   + page_size + (signed long int)size) / page_size) * page_size >= 4294967296l)
15
16  VERIFICATION FAILED
```

```
s2n/utils/s2n_mem.c
50  int s2n_realloc(struct s2n_blob *b, uint32_t size)
51  {
52      if (size == 0) { return s2n_free(b); }
53
54      if (size < b->allocated) { b->size = size; return 0; }
55
56      uint32_t allocate = page_size * ((size + (page_size - 1)) / page_size);
```

Listing 13: The error trace and the implementation of `s2n_realloc()`

which is the default in s2n. This unsigned overflow can be prevented by rearranging the sequence of the calculation as shown below: (as compared to line 56 of Listing 13)

```
s2n/utils/s2n_mem.c
58      // preventing unsigned overflow occurs at (size + (page_size - 1))
59      uint32_t allocate = page_size * (((size - 1) / page_size) + 1);
```

Listing 14: A fix for the implementation `s2n_realloc()`

6.2.2 Signed Overflow on Type Conversion

Another arithmetic overflow failure is detected on a type conversion in `s2n_connection_get_delay()` (line 379). The error trace and the detailed implementation are listed below:

```
detected-failures/s2n_connection_gs_harness-signed_overflow.txt
/* detail error trace is omitted */

State 248 file ./s2n/tls/s2n_config.c line 41 function get_nanoseconds_since_epoch thread 0
----------------------------------------------------
  current_time=6423702844750168064ul (0101100100100101100100111011001111111100000000000000000000000000)

State 254 file ./s2n/utils/s2n_timer.c line 35 function s2n_timer_elapsed thread 0
----------------------------------------------------
  elapsed=4227858432ul (0000000000000000000000000000000011111100000000000000000000000000)

Violated property:
  file ./s2n/tls/s2n_connection.c line 385 function s2n_connection_get_delay
  arithmetic overflow on unsigned to signed type conversion in (signed long long int)(conn->delay - elapsed)
  !(conn->delay + -elapsed >= 9223372036854775808ull)

VERIFICATION FAILED
```

```c
s2n/tls/s2n_connection.c
int64_t s2n_connection_get_delay(struct s2n_connection *conn)
{
    if (!conn->delay) { return 0; }

    uint64_t elapsed;
    GUARD(s2n_timer_elapsed(conn->config, &conn->write_timer, &elapsed));

    if (elapsed > conn->delay) { return 0; }

    return conn->delay - elapsed;
}
```

Listing 15: The error trace and the implementation of `s2n_connection_get_delay()`

This arithmetic overflow is due to the return type (`int64_t`) and the type of return variable (`uint64_t`) being inconsistent. Moreover, a signed integer overflow is an undefined behaviour in ANSI-C. As this is an API level function, the non-deterministic return value can lead to an unexpected behaviour in a third-party server implementation. This problem can be fixed with a correct return type of the function as shown below:

```c
s2n/tls/s2n_connection.c
uint64_t s2n_connection_get_delay(struct s2n_connection *conn)
```

Listing 16: A fix for the implementation `s2n_connection_get_delay()`

6.2.3 Memory Leaks on Error Exit

Among 14 of the failures, five of them indicate memory leaks have occurred when the s2n process exit on error. According to the development guide, the error handling in s2n is done by the macros defined in `s2n_error.h` and `s2n_safety.h`, which return `-1` or `NULL` value directly in order to terminate the process immediately when an error exists. As a result, the process does not have a chance to deallocate the already allocated dynamic memory properly, and all of them remain in the heap after the process has been terminated. The suggested practice for error handling in s2n and the macros are shown in the following code snippets:

s2n/utils/s2n_safety.h

```
/**
 *  s2n suggested practice for error handling:
 *  if (s2n_do_something(with_something_else) < 0) { return -1; }
 */

#define GUARD( x )      if ( (x) < 0 ) return -1
#define GUARD_PTR( x )  if ( (x) < 0 ) return NULL
```

s2n/error/s2n_errno.h

```
#define S2N_ERROR( x )      _S2N_ERROR( ( x ) ); return -1
#define S2N_ERROR_PTR( x )  _S2N_ERROR( ( x ) ); return NULL
```

Listing 17: The error handling and safety checking macros of s2n

The intention of the process is to guarantee its termination and in the case of error handling that is what the above macros are doing. Since most of the mainstream operating systems are able to reclaim all the heap memory allocated to the terminated process, therefore, this error handling behaviour is reasonable and does not count as a bug.

6.2.4 Buffer over-read on `strlen()`

In order to ensure maximum coverage of the harness, we allow all `char` arrays to carry every possible combination of values, even of those without any terminating character `'\0'`. However, three spurious warnings were detected on `strlen()` due to this approximation. Regarding the implementation of `strlen()`, it requires the input `char` array to be a string, which means an array must contain a terminating character `'\0'`.

This is the only criterion of using the function `strlen()` and we were not aware of that at the beginning. After a manual review on the verification result, we were able to trace this missing condition. A simple fix is done on the corresponding harnesses listed below. By introducing an assumption that the `char` array must carry at least one `'\0'` value,

```
cbmc/src/ansi-c/library/string.c
inline size_t strlen(const char *s)
{
  /* detailed implementation is omitted */
  __CPROVER_size_t len=0;
  while(s[len]!=0) len++;
  return len;
}
```

Listing 18: The implementation of `strlen()` provided by CBMC

`s2n_stuffer_base64_harness.c` (line 15) and `s2n_connection_gs_harness.c` (line 16), the false alarms are removed.

```
s2n-harness/module/s2n_stuffer_base64_harness.c
14      char message[message_size];
15      __CPROVER_assume(message[message_size-1] == '\0');
```

```
s2n-harness/module/s2n_connection_gs_harness.c
14      const char server_name[256];
15      uint8_t i;
16      __CPROVER_assume(i < 256 && server_name[i] == '\0');
```

Listing 19: Fixed the missing condition on 2 verification harnesses

6.3 Loop Bounds Analysis Results

There are over 50 different loops involved in the functions covered by the verification. 26 of whose LUBs are not able to be detected by CBMC automatically through syntactic analysis. We found most of the LUBs to be related to several particular sizes of data block, therefore there are some repeating LUBs, such as 17, 33, 49, 129 and 257. In Table 6.1, it lists out the analysis results of all 26 loops in detail. The first four columns show the loop IDs given by CBMC, the filenames, the line numbers of where the loops are located, and LUBs of the loops respectively. The descriptions of the loops and the reasons for determining their corresponding LUBs are shown in the last column.

For example, the loop `s2n_sslv3_mac_init.1` (the second row of Table 6.1) assigns the standard padding value over the data block. Since the SSLv3 MAC function only supports two hash algorithms, SHA1 and MD5, their block sizes are 40 bytes and 48 bytes respectively. This loop iterates at most 49 times before being terminated and therefore its LUB is 49.

Loop ID	File	Line	LUB	Description (bytes)
s2n_sslv3_mac_init.0	s2n_hmac.c	51	49	The loop iterates the block size of `SHA1` (40) or `MD5` (48).
s2n_sslv3_mac_init.1	s2n_hmac.c	59	49	The loop iterates the block size of `SHA1` (40) or `MD5` (48).
s2n_hmac_init.6	s2n_hmac.c	156	129	The loop iterates the `MAX_DIGEST_LENGTH` (64) or `MAX_BLOCK_SIZE` (128).
s2n_hmac_init.7	s2n_hmac.c	159	129	The loop iterates the `MAX_BLOCK_SIZE`(128).
s2n_hmac_init.8	s2n_hmac.c	166	129	The loop iterates the `MAX_BLOCK_SIZE`(128).
s2n_sslv3_mac_digest.0	s2n_hmac.c	73	49	The loop iterates the block size of `SHA1` (40) or `MD5` (48).
s2n_stuffer_alloc_ro_from_string.0	s2n_stuffer_text.c	76	50[§]	The loop exists in `strlen()` (49).
s2n_stuffer_write_base64.0	s2n_stuffer_base64.c	153	17[§]	The loop iterates over a `stuffer` for `s2n_stuffer_data_available(in)/3` times. It is only used for testing purpose (49/3=16).
s2n_stuffer_read_base64.0	s2n_stuffer_base64.c	141	17	The loop iterates over a `stuffer` for `s2n_stuffer_data_available(in)/4` times. It is only used for reading pem file line by line (65[†]/4=16).
s2n_stuffer_read_token.0	s2n_stuffer_text.c	56	66	The loop iterates over a `stuffer` until it reaches the given taken or the end. It is only used for reading pem file line by line (65[†])
s2n_stuffer_data_from_pem.1	s2n_stuffer_pem.c	46	17	The loop exists in `strlen()` for counting the keyword length of pem[‡](16), (12) or (14).
s2n_stuffer_data_from_pem.3	s2n_stuffer_pem.c	48	17	The loop exists in `strlen()` for counting the keyword length of pem[‡](16), (12) or (14).
s2n_stuffer_data_from_pem.5	s2n_stuffer_pem.c	75	14	The loop iterates over the pem `stuffer` until it reaches a pem line or the end of the file. Minimum number of lines in pem[‡](8), (13) or (3).
s2n_stuffer_data_from_pem.7	s2n_stuffer_pem.c	84	17	The loop exists in `strlen()` for counting the keyword length of pem[‡](16), (12) or (14).
s2n_stuffer_data_from_pem.9	s2n_stuffer_pem.c	86	17	The loop exists in `strlen()` for counting the keyword length of pem[‡](16), (12) or (14).
s2n_drbg_bits.0	s2n_drbg.c	44	4	The loop iterates the data each drbg block(16). Different sizes of data are given for various usages: (8), (28), (32), (36), (48). i.e. (48/16=3)
s2n_drbg_update.1	s2n_drbg.c	74	33	The loop iterates over the provided data and `xor` each byte. It is only used by `s2n_drbg_generate()` (32) and `s2n_drbg_seed()` (32).
s2n_drbg_seed.0	s2n_drbg.c	100	33	The loop iterates over the personalised string provided by `s2n_drbg_generate()` (32) or `s2n_drbg_instantiate()` (32).
s2n_get_urandom_data.0	s2n_random.c	113	33	The loop is used to read a user-defined random data for `s2n_drbg_seed()` (32).
_read.0	unistd.c	193	33	The loop models the behaviour of reading from a file provided by the CBMC ANSI-C Library.
s2n_config_set_cipher_preferences.0	s2n_config.c	225	8	The loop iterates over the available cipher preferences (7)
s2n_config_free_cert_chain_and_key.0	s2n_config.c	175	11	The loop iterates over the certificate chains. The default maximum chain length in OpenSSL is 10.
s2n_ecc_find_supported_curve.0	s2n_ecc.c	286	2	The loop iterates over the number of `IANA IDs/2`. (1)
s2n_set_server_name.0	s2n_connection.c	335	257	The loop exists in `strlen()`. (256)
s2n_get_server_name.0	s2n_connection.c	347	257	The loop exists in `strlen()`. (256)
s2n_get_application_protocol.0	s2n_connection.c	356	257	The loop exists in `strlen()`. (256)

[†] the maximum length per line in a pem file.

[‡] the three numbers refer to the property of *RSA Private key*, *Certificate* and *DH Parameters* in a pem file respectively.

[§] the bound depends on the size of string (48bytes), used to convert to a base64 string (64bytes) for the `s2n_stuffer_read_base64()` verification.

Table 6.1: Loop Bounds Analysis Result of s2n

6.4 Model Checking Results

Note that, unfortunately, the complete verification reports generated by CBMC and the detailed error traces are not attached in this report due to the page limitation. For more information, please refer to the supplementary files or visit our Github repository[2].

This verification covers 67.4% of all API-level functions and 52.3% of all public functions. This limited coverage is due to the difficulty in verifying some of the core functions and the remaining functions that depend on them. For example, the verification of `s2n_connection_harness.c` (the 16^{th} row of Table 6.2) is killed by the terminal during post-processing. Since the function `s2n_connection_new()` cannot be verified, we are not able to prepare a valid `s2n_connection` for verifying the functions depending on it. Moreover, the verification of `s2n_config_server_harness.c` (the 13^{th} of Table 6.2) has a problem during the conversion to SSA form, which is related to another core function `s2n_config_add_cert_chain_and_key()` for initialising a `s2n_config` with a certificate chain and keys.

The experimental results are summarised in Table 6.2 below. The first three columns are the names of verification harnesses, the verification results of the corresponding iterations, and the numbers of failed tests over the total numbers of tests respectively. The next three columns are the times used in generating SAT formulas, solving the formulas and the overall execution time in seconds. The seventh and the eighth columns are the numbers of verification conditions generated by CBMC and the number of remaining verification conditions after simplification. The last two columns show the number of variables and clauses in the SAT formula.

The complexity of s2n and the coverage of the verification are visualised in Figure 6.2 below. Due to the limited paper space, a simplified overview of s2n that only contains public functions is shown. Different colours are used in the figure and the colours correspond to:

1. Orange: API-level functions
2. Green: Public functions covered in the verification
3. Purple: External functions modelled by stubs

6.5 Threats to Validity

6.5.1 Repeatability

All materials involved in this experiment, including the file structure of this repository, the executions, and all verification harnesses are attached in Appendix A, D and E. However, due to the limited page space, all stubs we have created in this project are not attached

[2] The Github repository of this project: https://github.com/IssacLam/securing-the-web

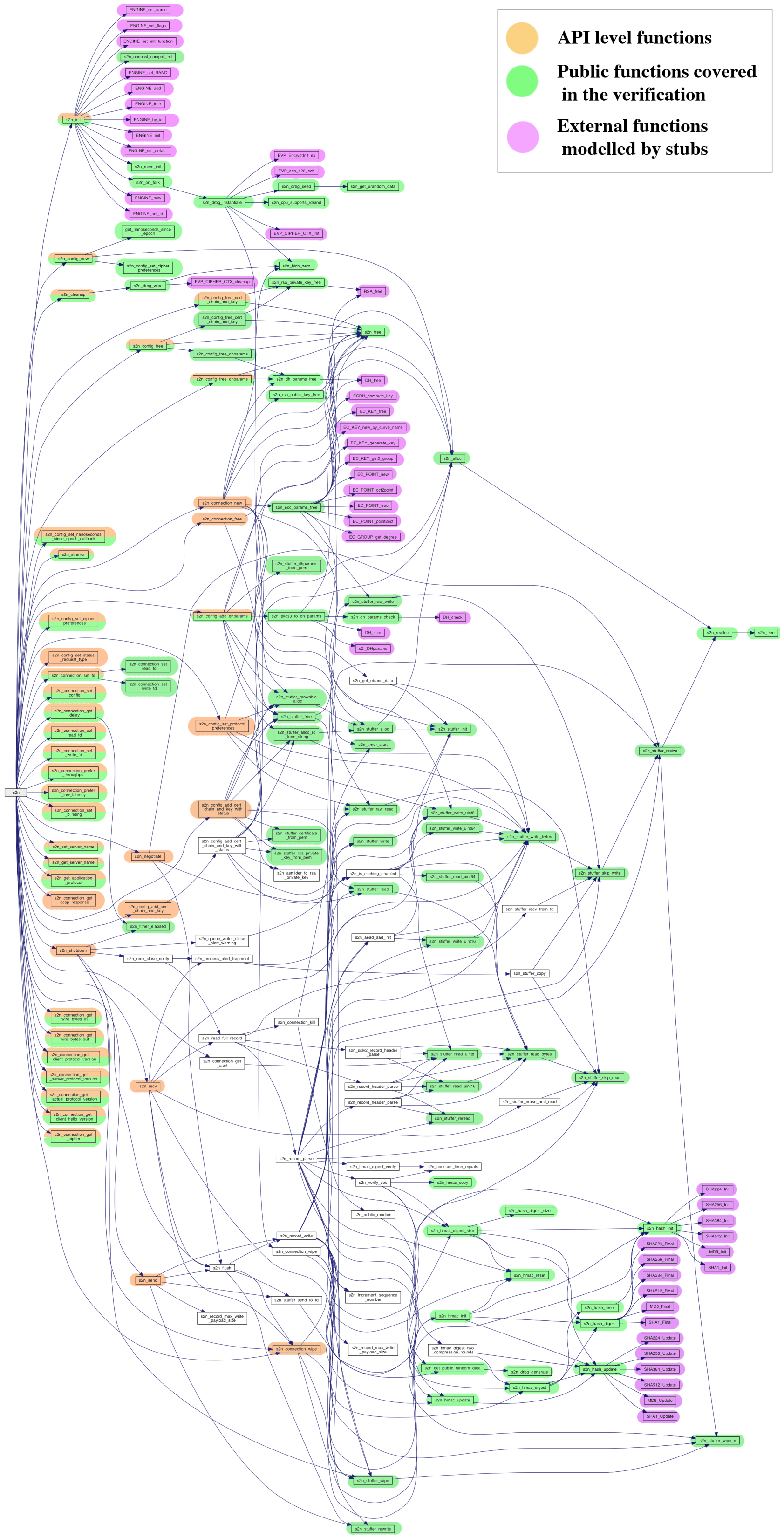


Figure 6.2: The Verification Coverage & The Overview of s2n Public Function Calls

Harness	SAT	#of failed Results	Time (s)			VCC[§]		SAT formula statistics	
			Formula generation	Solving	Total	Gen	Simp	#of variables	#of clauses
s2n_blob	SAT	1/42	0.418	0.007	0.415	29	11	1450	2823
	UNSAT							1450	1043
s2n_mem	SAT	2/302	1.058	0.099	1.157	403	168	25021	62481
	UNSAT							25012	14801
s2n_timer	UNSAT	0/398	1.322	0.380	1.762	275	125	55254	281492
s2n_hash	UNSAT	0/312	0.841	0.400	1.241	340	60	98319	267059
s2n_hmac	UNSAT	0/1015	14.356	16.137	30.493	7495	1275	1445978	5606727
s2n_stuffer	UNSAT	0/704	1722.768	564.44	2287.208	67540	38730	13431597	34789455
s2n_stuffer_base64[¶]	SAT	3/901	915.566	464.398	1379.964	25176	14286	3181447	14472455
	SAT							3181447	1802115
	UNSAT							3181447	1604792
s2n_stuffer_text[¶]	UNSAT	0/702	3.346	78.278	81.624	1827	1075	4854780	28002070
s2n_override _openssl_random	SAT	2/1516	226.482	93.505	319.987	21305	12126	2650201	11191936
	SAT							2650201	412966
	UNSAT							2650201	403524
s2n_drbg	UNSAT	0/1167	114.91	103.009	217.919	182338	92342	44585234	140834724
s2n_ecc	SAT	3/874	10.012	8.481	18.493	7518	4235	779809	2200597
	SAT							779809	52580
	UNSAT							779809	48486
s2n_config_client	UNSAT	0/1186	6.827	696.328	703.115	1536	1073	55583317	180060732
s2n_config_server[†]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
s2n_config_dhparams	UNSAT	0/1896	435.212	412.252	847.464	29601	16636	70985517	232832304
s2n	SAT	1/555	7.995	1355.71	1363.705	21947	15820	2572719	14221606
	UNSAT							2572719	2158242
s2n_connection[‡]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
s2n_connection_gs	SAT	2/3011	29.241	3.525	32.766	6709	1750	410718	556807
	SAT							410718	113608
	UNSAT							410718	43716

[¶] The chosen LUB does not cover the entire loop due to the limited scalability of BMC.
[§] Gen: Generated Verification Condition; Simp: Simplified Verification Condition.
[†] Error occurred during the verification.
[‡] The verification process is killed by the terminal.

Table 6.2: Model Checking Result of s2n

in this report. For more information, please refer to the supplementary files or visit our Github repository[3]. This experiment can simply be repeated by following the instructions described in Chapter 5. In this sense, the repeatability of this experiment would not be a concern.

6.5.2 Execution Environment Dependency

Regarding the execution environment, as described in Section 5.1 and Section 5.2, OpenSSL is chosen as the *libcrypto* cryptography library provider and this experiment is performed on Mac OS X. Some system dependent functions in s2n will behave differently since various underlying software packages or system libraries are being used. Therefore, the experimental results of those functions can be different from ours. Otherwise, the same results can be obtained under the same experiment setup, except the times.

[3] The Github repository of this project: https://github.com/IssacLam/securing-the-web

6.5.3 Verification Tool Dependency

Concerning the reliability of CBMC, bugs could possibly exist in its implementation, and it might miss some bugs, but would not affect the result of those bugs we have found. Since the given violation traces locate exactly where the bugs are, and more importantly it is confirmed by Amazon s2n developers, this is solid evidence for supporting our hypothesis that CBMC is able to hunt down hidden bugs in the s2n implementation not found by the other common testing techniques used in the industry. However, it should be noted that any bugs missed by CBMC could be a threat to the claiming of the memory safety of the implementation since there is no detailed report for showing how the properties are verified.

6.5.4 Correctness of the Verification Harnesses

As mentioned in Section 4.2.3, harnesses are used to prepare the environment for the verification with respect to the function usage contracts. If a harness has any conflicts with the contracts, false alarms will be generated. Since CBMC has traces for any violation, everything can be traced back to the cause. By examining the verification results, we are able to tell whether the conflict is in the harness or the targeted implementation, as described in Section 6.2. Therefore, the correctness of verification harnesses is not a potential threat to the verification result as all conflicts can be determined by reviewing the error traces.

Chapter 7

Discussion and lessons learnt

7.1 Memory safety of the s2n implementation

Consider that BMC guarantees all bugs are detected within the given bound. More importantly, with a bound that is large enough to cover the function usage, it can prove the absence of all bugs in the implementation. As mentioned in Section 4.2.6, we chose the maximum number of loop iterations as the LUBs in our verification. It can ensure complete coverage in the function verification. Therefore, our verification results can also show the memory safety of those verified implementations, except for two verification harnesses, `s2n_stuffer_base64_harness.c` and `s2n_stuffer_text_harness.c`. As their chosen LUBs do not cover the entire loop iterations due to the limited scalability of BMC, as marked in the verification results in Section 6.4.

7.2 Trade-offs in Software Verification

During this project, we made some trade-offs regarding the properties of the triangle model mentioned in Section 2.1 to obtain this successful result. First of all, we make use of over-approximation in order to scale up the verification capability of model checking. However, the drawback is the false alarms to the verification. The use of human assistance at this point can help minimise the number of false alarms.

Trade-off between Under- and Over- Approximation Model checking is an under-approximation verification technique that would produce no false alarms. In order to achieve this result, CBMC requires that the detailed implementations of all functions involved to be included, including those underlying software packages and system libraries. However, in this case, s2n is not a stand-alone software package, to include the implementations of external functions would enlarge the exploration space and exceed our verification scope. Moreover, sometimes the documentations only provide some descriptions of the functions rather than the detail implementations. Hence, we have to always create stubs for those functions with over-approximations as mentioned in Section 4.2.5. In addition, we further simplified some functions depending on the verification needs by using over-approximation as mentioned in Section 4.2.7.

Trade-off between Automation and Human-assisted To remain a low false alarm rate while using over-approximation, human effort is needed. Besides the use of over-approximation as described above, without considering the function usage contracts, the verification can also produce false alarms. Since s2n is not a single-function software package, an appropriate execution environment is essential to the verification. Therefore, human assistance is required to capture the function usage contracts for preparing proper harnesses with suitable LUBs as mentioned in Section 4.2.3 and Section 4.2.6.

7.3 Manual Loop Bounds Analysis Effort for BMC

As mentioned in Section 4.2.6, loop bounds analysis is performed on all functions involved in the verification and the maximum numbers of iterations are chosen as the LUBs. This can ensure the coverage of the verification in order to detect bugs and more importantly to show the memory safety of the implementations. Our approach is slightly different from the study conducted by Y. Kim et al. [28]. Instead of showing the memory safety of the implementation, they were focused on detecting bugs only. They chose the minimal number of iterations to exit a loop as their loop unwinding bounds. By increasing the number in each iteration, hopefully, a witness to the property violation will exist. In this case, a sound upper bound is good enough for detecting bugs, but not enough for showing the memory safety. The authors further concluded that without an accurate loop bounds analysis, there are possibilities to make the verification fail to detect bugs even if a large LUB is used. Since the choices of LUBs directly affect the quality of verification results and the effectiveness of BMC, the manual effort of loop bounds analysis is indispensable.

7.4 Dealing with the Scalability of BMC

During this project, we observed that the key limitation of BMC is the lack of scalability. This is a well-known limitation of model checking due to state explosions, which makes it hard to be applicable to industrial software [28]. Since industrial software always consists of numerous loops, sometimes even be nested, and also with a huge number of iterations, this is the main cause of the exponential complexity as well as state explosions. For example, some string manipulation functions in s2n, such as `s2n_stuffer_read_base64()`, will iterate over the entire memory space of a given `char` array. It is impractical to use the maximum values of `uint64_t` as the LUBs of such functions. As an example to demonstrate this impracticality, it takes more than 5 hours to convert explored space into SSA form when 49 is chosen as its LUB. In this case, we have chosen a smaller number as the LUB in order to reduce the exploration space and keep up with the verification. Due to the insufficient LUB coverage, the memory safety of those functions are not guaranteed, but nevertheless, the verification is still useful for detecting bugs in that limited coverage. Besides choosing a smaller LUB, simplifying some complex computations and those verified functions is also a viable way to scale up the verification further, as mentioned in Section 4.2.7.

7.5 Limitations on Functional Correctness Verification

Due to the use of non-determinism, stubs only return a valid result but not necessarily a correct one, as described in Section 4.2.5. This makes the correctness of several functional properties unverifiable. For example, the functional properties of the hash function are as follows:

1. The same input string must have the same hash value.
2. Similar inputs result in very different hash values.
3. There is fixed output size for variable input size.

Only the 3^{rd} property is able to be verified while other properties require deterministic values to verify the functional correctness. In contrast, deductive verification could be a better choice to verify such properties. Since it performs the verification over the dynamic execution instead of static execution, it does not require the source code of the underlying software packages and a large number of memory space. However, this would heavily rely on human efforts for capturing the specifications and introducing invariants for each function to support the verification.

7.6 Inconvenience Syntax to Loop Bounds Analysis

During the loop bounds analysis, we found two kinds of loop declaration that require extra effort to review the actual loop implementation manually.

while(1) loops This syntax is used for string manipulations and listening to the connections. However, it makes the unwinding assertions always fail no matter how large the value of LUB is, as shown below. Therefore, the iterative algorithm for guessing LUBs cannot be applied to `while(1)` loops and manual review is needed.

```
__CPROVER_assert(!1, "unwinding assertion loop.0");
```

Listing 20: A failure condition for unwinding assertions

do . . . while(0) loops This syntax is commonly used for convenience to declare a multi-statement macro correctly. The details are explained in the example below: Suppose we have two macros declared, where `A()` should be followed by `B()`:

```
#define F(x) A(x); B(x);
#define G(x) do { A(x); B(x); } while(0)
```

```
if(cond)
    F(x);

/* is equivalent to */
if(cond)
    A(x);
B(x);
```

(a) Incorrect nested statements as expected

```
if(cond)
    G(x);

/* is equivalent to */
if(cond) {
    A(x); B(x);
}
```

(b) Correct nested statements as expected

A set of safety checks in s2n is declared using `do ... while(0)` loop macros, listed in Appendix C, and they are applied to the entire s2n implementation. This syntax will burden the workload on loop bounds analysis when using the `--show-loop` option in CBMC. Since all of them will be listed in the result, an extra manual effort is needed to filter them out from the loop bounds analysis.

Chapter 8

Conclusion and Future Work

In this project, we applied CBMC to verify the memory safety of Amazon's s2n TLS implementation and successfully found two arithmetic overflow bugs in it. The corresponding solutions are reported to the s2n Github repository, and the Amazon's s2n developers acknowledged the bugs and accepted our changes to the implementation according to our findings. Through this project, we have demonstrated that CBMC is capable of detecting hidden bugs in heavily tested critical industrial software such as s2n. In addition, our verification also shows the memory safety of s2n implementation by determining proper LUBs, which cover a sufficient number of loop iterations according to the function usage. In order to obtain this result, we have presented our verification approach with several key steps that require substantial human effort to scale up the verification and improve the precision of the verification result, which is one of the contributions of this project. However, due to the limitation of project time and the lack of scalability in BMC, our verification can only cover 67.4% of all API-level functions and 52.3% all of public functions in s2n.

There is plenty of room for improvement and much need for continual work. We have come up with the following three suggestions:

Extend the Verification Coverage Due to the importance of securing the s2n implementation, a full coverage verification is needed. Besides covering all the s2n functions, the verification also needs to cover all execution environment setups, which support s2n, such as being built with the `libcrypto` library from different providers, running on 32-bit or 64-bit machines, and being executed with different security policies.

Automated Loop Bounds Analysis Concerning the necessity of loop bounds analysis, the required human effort could be one of the obstacles in applying BMC techniques to industrial practice. Therefore by reducing the verification effort, an automated loop bounds analysis heuristic is essential to making BMC widely used in the industry.

Github Integration Tool As Github is an extensively used platform for hosting most open source projects, an integration tool that can disseminate software verification, such as BMC, to Continuous Integration would not only support build automation and test automation, but also verification automation. As a result, BMC can even be universally adopted in non-critical software as well as improving software quality levels.

Appendix A

File Structure of the Repository

Note that we only list the core files used in this verification as the followings. Those files generated by third parties and not being used in our verification are omitted.

```
/
├── dissertation.pdf
├── dissertation-source
│   └── ...
└── securing-the-web
    ├── detected-failures
    │   ├── s2n_blob_harness-bounds_pointer.txt
    │   ├── s2n_connection_gc_harness-bounds_pointer-fixed_harness.txt
    │   ├── s2n_connection_gs_harness-bounds_pointer.txt
    │   ├── s2n_connection_gs_harness-signed_overflow.txt
    │   ├── s2n_ecc_harness-memory_leak.txt
    │   ├── s2n_ecc_harness-pointer.txt
    │   ├── s2n_harness-memory_leak.txt
    │   ├── s2n_mem_harness-memory_leak.txt
    │   ├── s2n_mem_harness-unsigned_overflow.txt
    │   ├── s2n_override_openssl_random_harness-memory_leak.txt
    │   ├── s2n_override_openssl_random_harness-pointer.txt
    │   ├── s2n_stuffer_base64-bounds_pointer-fixed_harness.txt
    │   ├── s2n_stuffer_base64_harness-bounds_pointer.txt
    │   └── s2n_stuffer_base64_harness-memory_leak.txt
    ├── doxygen-overview
    │   └── ...
    └── helper-tools
        ├── all_functions.txt
        ├── openssl_functions.txt
        ├── external_function_scanner.sh
        ├── public_function_scanner.sh
        ├── external_functions.txt
        ├── public_functions.txt
        └── function_based_analyzer.sh
```

```
|
|   |
|   └── verifier.sh
|   └── function_scanner.sh
└── instructions
|   └── cbmc-installation.txt
|   └── s2n-installation.txt
|   └── s2n-complie-and-execution.txt
└── s2n
|   └── ...
└── s2n-harness
|   └── bottom-up
|   |   └── ...
|   └── module
|   |   └── execution.txt
|   |   └── s2n_harness.c
|   |   └── s2n_blob_harness.c
|   |   └── s2n_hash_harness.c
|   |   └── s2n_config_client_harness.c
|   |   └── s2n_hmac_harness.c
|   |   └── s2n_config_dhparams_harness.c
|   |   └── s2n_mem_harness.c
|   |   └── s2n_config_server_harness.c
|   |   └── s2n_override_openssl_random_harness.c
|   |   └── s2n_connection_gs_harness.c
|   |   └── s2n_stuffer_base64_harness.c
|   |   └── s2n_connection_harness.c
|   |   └── s2n_stuffer_harness.c
|   |   └── s2n_drbg_harness.c
|   |   └── s2n_stuffer_text_harness.c
|   |   └── s2n_ecc_harness.c
|   |   └── s2n_timer_harness.c
|   └── top-down
|       └── ...
└── s2n-lib
|   └── CommonCrypto
|   |   └── CommonDigest.c
|   └── fcntl.c
|   └── mach_time.c
|   └── mman.c
|   └── openssl
|   └── pthread.c .4 s2n
|   |   └── s2n_stuffer_base64.c
|   |   └── s2n_stuffer_text.c
|   └── stdlib.c
|   └── unistd.c
└── s2n-verification-result
    └── function-based
    |   └── ...
    └── module-based
```

```
├── s2n.txt
├── s2n_hmac.txt
├── s2n_blob.txt
├── s2n_mem.txt
├── s2n_config.txt
├── s2n_override_openssl_random.txt
├── s2n_config_client.txt
├── s2n_stuffer.txt
├── s2n_config_dhparams.txt
├── s2n_stuffer_base64.txt
├── s2n_config_server.txt
├── s2n_stuffer_base64_1hr_timeout.txt
├── s2n_connection_gs.txt
├── s2n_stuffer_base64_5hr_timeout.txt
├── s2n_drbg.txt
├── s2n_stuffer_base64_Wrong_Bounds.txt
├── s2n_ecc.txt
├── s2n_stuffer_text.txt
├── s2n_hash.txt
└── s2n_timer.txt
```

Appendix B

External Functions Used in s2n

openssl/bn.h
BN_bin2bn
BN_bn2bin
BN_free
BN_is_zero
BN_num_bytes

openssl/dh.h
DH_compute_key
DH_free
DH_generate_key
DH_new
DH_size
DHparams_dup
d2i_DHparams

openssl/ec.h
EC_GROUP_get_degree
EC_KEY_free
EC_KEY_generate_key
EC_KEY_get0_group
EC_KEY_get0_public_key
EC_KEY_new_by_curve_name
EC_KEY_set_public_key
EC_POINT_free
EC_POINT_new
EC_POINT_oct2point
EC_POINT_point2oct

openssl/ecdh.h
ECDH_compute_key

openssl/engine.h
ENGINE_add
ENGINE_by_id
ENGINE_free
ENGINE_init
ENGINE_new
ENGINE_set_RAND
ENGINE_set_default
ENGINE_set_flags
ENGINE_set_id
ENGINE_set_init_function
ENGINE_set_name

openssl/evp.h
EVP_CIPHER_CTX_cleanup
EVP_CIPHER_CTX_ctrl
EVP_CIPHER_CTX_init
EVP_CIPHER_CTX_set_padding
EVP_DecryptFinal_ex
EVP_DecryptInit_ex
EVP_DecryptUpdate
EVP_EncryptFinal_ex
EVP_EncryptInit_ex
EVP_EncryptUpdate
EVP_PKEY_free
EVP_PKEY_get1_RSA
EVP_aes_128_cbc
EVP_aes_128_ecb
EVP_aes_128_gcm
EVP_aes_256_cbc
EVP_aes_256_gcm
EVP_des_ede3_cbc

openssl/md5.h
MD5_Final
MD5_Init
MD5_Update

openssl/rc4.h
RC4
RC4_set_key

openssl/rsa.h
RSA_free
RSA_private_decrypt
RSA_public_encrypt
RSA_sign
RSA_size
d2i_RSAPrivateKey

openssl/sha.h
SHA1_Final
SHA1_Init
SHA1_Update
SHA224_Final
SHA224_Init
SHA224_Update
SHA256_Final
SHA256_Init
SHA256_Update
SHA384_Final
SHA384_Init
SHA384_Update
SHA512_Final
SHA512_Init
SHA512_Update

openssl/x509.h
X509_free
X509_get_pubkey
d2i_X509

sys/mman.h
madvise
minherit
mlock
mmap
munlock

sys/param.h
MIN

sys/stat.h
fstat

mach/mach_time.h
mach_absolute_time
mach_timebase_info

fcntl.h
open

pthread.h
pthread_atfork

stdint.h
INT64_C

stdlib.h
free
getenv
posix_memalign
rand

string.h
memcmp
memset
strlen

strings.h
strcasecmp

time.h
clock_gettime
nanosleep
time

unistd.h
close
getpid
read
sleep
write
syscall
sysconf

Appendix C

Error Handling and Safety Checking in s2n

```
/* NULL check a pointer */
#define notnull_check( ptr )           do { if ( (ptr) == NULL ) { S2N_ERROR(S2N_ERR_NULL); } } while(0)

/* Check memcpy and memset's arguments, if these are not right, log an error
 */
#define memcpy_check( d, s, n )     do { if ( (n) ) { notnull_check( (d) ); memcpy( (d), (s), (n)); } } while(0)
#define memset_check( d, c, n )     do { if ( (n) ) { notnull_check( (d) ); memset( (d), (c), (n)); } } while(0)

/* Range check a number */
#define gte_check(n, min)  do { if ( (n) < min ) { S2N_ERROR(S2N_ERR_SAFETY); } } while(0)
#define lte_check(n, max)  do { if ( (n) > max ) { S2N_ERROR(S2N_ERR_SAFETY); } } while(0)
#define gt_check(n, min)  do { if ( (n) <= min ) { S2N_ERROR(S2N_ERR_SAFETY); } } while(0)
#define lt_check(n, max)  do { if ( (n) >= max ) { S2N_ERROR(S2N_ERR_SAFETY); } } while(0)
#define eq_check(a, b)  do { if ( (a) != (b) ) { S2N_ERROR(S2N_ERR_SAFETY); } } while(0)
#define ne_check(a, b)  do { if ( (a) == (b) ) { S2N_ERROR(S2N_ERR_SAFETY); } } while(0)
#define inclusive_range_check( low, n, high )  gte_check(n, low); lte_check(n, high)
#define exclusive_range_check( low, n, high )  gt_check(n, low); lt_check(n, high)
```

Listing 21: The safety check macros of s2n implementation.

Appendix D

CBMC Execution Commands

```
s2n_blob_harness.c
cbmc -I ./s2n -I ./s2n/api -I ./s2n/libcrypto-build/openssl/include
 ↪  ./s2n-harness/module/s2n_blob_harness.c ./s2n/utils/s2n_blob.c

s2n_mem_harness.c
cbmc -I ./s2n -I ./s2n/api -I ./s2n/libcrypto-build/openssl/include
 ↪  ./s2n-harness/module/s2n_mem_harness.c ./s2n/utils/s2n_blob.c ./s2n/utils/s2n_mem.c
 ↪  ./s2n-lib/mman.c ./s2n-lib/stdlib.c ./s2n-lib/unistd.c

s2n_timer_harness.c
cbmc -I ./s2n -I ./s2n/api -I ./s2n/libcrypto-build/openssl/include -I
 ↪  ./s2n/libcrypto-build/ ./s2n-harness/unit/s2n_timer_harness.c
 ↪  ./s2n/module/s2n_timer.c ./s2n/tls/s2n_config.c ./s2n-lib/mach_time.c

s2n_hash_harness.c
cbmc -I ./s2n -I ./s2n/api -I ./s2n/libcrypto-build/openssl/include
 ↪  ./s2n-harness/unit/s2n_hash_harness.c ./s2n/crypto/s2n_hash.c
 ↪  ./s2n-lib/CommonCrypto/CommonDigest.c

s2n_hmac_harness.c
cbmc -I ./s2n -I ./s2n/api -I ./s2n/libcrypto-build/openssl/include
 ↪  ./s2n-harness/module/s2n_hmac_harness.c ./s2n/crypto/s2n_hmac.c
 ↪  ./s2n/crypto/s2n_hash.c ./s2n/stuffer/s2n_stuffer.c
 ↪  ./s2n-lib/CommonCrypto/CommonDigest.c --unwindset
 ↪  s2n_sslv3_mac_init.0:49,s2n_sslv3_mac_init.1:49,s2n_hmac_init.6:129,s2n_hmac_init.7:129,s2n_hmac_init.8:129

s2n_stuffer_harness.c
cbmc -I ./s2n -I ./s2n/api -I ./s2n/libcrypto-build/openssl/include
 ↪  ./s2n-harness/module/s2n_stuffer_harness.c ./s2n/stuffer/s2n_stuffer.c
 ↪  ./s2n/stuffer/s2n_stuffer_text.c ./s2n/utils/s2n_blob.c ./s2n/utils/s2n_random.c
 ↪  ./s2n/utils/s2n_mem.c ./s2n-lib/stdlib.c ./s2n-lib/mman.c --unwind 49

s2n_stuffer_base64_harness.c
cbmc -I ./s2n -I ./s2n/api -I ./s2n/libcrypto-build/openssl/include
 ↪  ./s2n-harness/module/s2n_stuffer_base64_harness.c ./s2n/stuffer/s2n_stuffer.c
 ↪  ./s2n/stuffer/s2n_stuffer_base64.c ./s2n/stuffer/s2n_stuffer_text.c
 ↪  ./s2n/utils/s2n_random.c ./s2n/utils/s2n_blob.c ./s2n/utils/s2n_mem.c
 ↪  ./s2n-lib/stdlib.c ./s2n-lib/mman.c --unwindset
 ↪  s2n_stuffer_alloc_ro_from_string.0:50,s2n_stuffer_write_base64.0:17,s2n_stuffer_read_base64.0:17

s2n_stuffer_text_harness.c
```

```
cbmc -I ./s2n -I ./s2n/api -I ./s2n/libcrypto-build/openssl/include
 ↪   ./s2n-harness/module/s2n_stuffer_text_harness.c ./s2n/stuffer/s2n_stuffer.c
 ↪   ./s2n/stuffer/s2n_stuffer_text.c ./s2n/stuffer/s2n_stuffer_text.c
 ↪   ./s2n/utils/s2n_random.c ./s2n/utils/s2n_blob.c ./s2n/utils/s2n_mem.c
 ↪   ./s2n-lib/stdlib.c ./s2n-lib/mman.c --unwindset s2n_stuffer_read_token.0:66

s2n_override_openssl_random_harness.c
cbmc -I ./s2n -I ./s2n/api -I ./s2n/libcrypto-build/openssl/include
 ↪   ./s2n-harness/module/s2n_override_openssl_random_harness.c
 ↪   ./s2n/stuffer/s2n_stuffer.c ./s2n/utils/s2n_random.c ./s2n/utils/s2n_mem.c
 ↪   ./s2n/utils/s2n_blob.c ./s2n/crypto/s2n_dhe.c ./s2n/crypto/s2n_drbg.c
 ↪   ./s2n/stuffer/s2n_stuffer_pem.c ./s2n-lib/openssl/dh.c ./s2n-lib/mman.c
 ↪   ./s2n-lib/stdlib.c ./s2n-lib/s2n/s2n_stuffer_base64.c
 ↪   ./s2n-lib/s2n/s2n_stuffer_text.c --unwindset
 ↪   s2n_stuffer_data_from_pem.1:17,s2n_stuffer_data_from_pem.3:17,s2n_stuffer_data_from_pem.5:14,s2n_stuf

s2n_drbg_harness.c
cbmc -I ./s2n -I ./s2n/api -I ./s2n/libcrypto-build/openssl -I
 ↪   ./s2n/libcrypto-build/openssl/crypto -I ./s2n/libcrypto-build/openssl/crypto/engine
 ↪   -I ./s2n/libcrypto-build/openssl/include ./s2n-harness/module/s2n_drbg_harness.c
 ↪   ./s2n/stuffer/s2n_stuffer.c ./s2n/crypto/s2n_drbg.c ./s2n/utils/s2n_blob.c
 ↪   ./s2n/utils/s2n_mem.c ./s2n/tls/s2n_config.c ./s2n/utils/s2n_random.c
 ↪   ./s2n/crypto/s2n_sequence.c ./s2n-lib/openssl/evp.c ./s2n-lib/mman.c
 ↪   ./s2n-lib/stdlib.c ./s2n-lib/fcntl.c --unwindset
 ↪   s2n_drbg_bits.0:4,s2n_drbg_update.1:33,s2n_drbg_seed.0:33

s2n_harness.c
cbmc -I ./s2n -I ./s2n/api -I ./s2n/libcrypto-build/openssl -I
 ↪   ./s2n/libcrypto-build/openssl/crypto -I ./s2n/libcrypto-build/openssl/crypto/engine
 ↪   -I ./s2n/libcrypto-build/openssl/include ./s2n-harness/module/s2n_harness.c
 ↪   ./s2n/utils/s2n_random.c ./s2n/crypto/s2n_drbg.c ./s2n/utils/s2n_mem.c
 ↪   ./s2n-lib/unistd.c ./s2n-lib/openssl/engine.c ./s2n-lib/fcntl.c
 ↪   ./s2n/utils/s2n_blob.c ./s2n-lib/openssl/evp.c ./s2n/crypto/s2n_sequence.c
 ↪   ./s2n-lib/pthread.c --unwindset _read.0:33,s2n_get_urandom_data.0:33

s2n_ecc_harness.c
cbmc -I ./s2n -I ./s2n/api -I ./s2n/libcrypto-build/openssl -I
 ↪   ./s2n/libcrypto-build/openssl/crypto -I ./s2n/libcrypto-build/openssl/crypto/engine
 ↪   -I ./s2n/libcrypto-build/openssl/include ./s2n-harness/module/s2n_ecc_harness.c
 ↪   ./s2n/utils/s2n_mem.c ./s2n/utils/s2n_blob.c ./s2n/stuffer/s2n_stuffer.c
 ↪   ./s2n/crypto/s2n_ecc.c ./s2n-lib/mman.c ./s2n-lib/stdlib.c ./s2n-lib/openssl/ec.c
 ↪   --unwindset s2n_ecc_find_supported_curve.0:2

s2n_config_client_harness.c
cbmc -I ./s2n -I ./s2n/api -I ./s2n/libcrypto-build/openssl -I
 ↪   ./s2n/libcrypto-build/openssl/crypto -I ./s2n/libcrypto-build/openssl/crypto/engine
 ↪   -I ./s2n/libcrypto-build/openssl/include
 ↪   ./s2n-harness/module/s2n_config_cllient_harness.c ./s2n/tls/s2n_config.c
 ↪   ./s2n/utils/s2n_mem.c ./s2n/utils/s2n_blob.c ./s2n/crypto/s2n_rsa.c
 ↪   ./s2n/crypto/s2n_dhe.c ./s2n-lib/openssl/rsa.c ./s2n-lib/openssl/dh.c
 ↪   ./s2n-lib/mman.c ./s2n-lib/stdlib.c --unwindset
 ↪   s2n_config_free_cert_chain_and_key.0:11

s2n_config_server_harness.c
```

```
cbmc -I ./s2n -I ./s2n/api -I ./s2n/libcrypto-build/openssl -I
 ↪   ./s2n/libcrypto-build/openssl/crypto -I ./s2n/libcrypto-build/openssl/crypto/engine
 ↪   -I ./s2n/libcrypto-build/openssl/include
 ↪   ./s2n-harness/module/s2n_config_server_harness.c ./s2n/tls/s2n_config.c
 ↪   ./s2n/utils/s2n_mem.c ./s2n/utils/s2n_blob.c ./s2n/crypto/s2n_rsa.c
 ↪   ./s2n/crypto/s2n_dhe.c ./s2n/stuffer/s2n_stuffer.c ./s2n-lib/s2n/s2n_stuffer_base64.c
 ↪   ./s2n-lib/s2n/s2n_stuffer_text.c ./s2n/stuffer/s2n_stuffer_pem.c
 ↪   ./s2n-lib/openssl/rsa.c ./s2n-lib/openssl/dh.c ./s2n-lib/mman.c ./s2n-lib/stdlib.c
 ↪   --unwindset
 ↪   s2n_config_free_cert_chain_and_key.0:11,s2n_stuffer_data_from_pem.1:17,s2n_stuffer_data_from_pem.3:17,s2n_s
 ↪   --unwind 1

s2n_connection_gs_harness.c
cbmc -I ./s2n -I ./s2n/api -I ./s2n/libcrypto-build/openssl -I
 ↪   ./s2n/libcrypto-build/openssl/crypto -I ./s2n/libcrypto-build/openssl/crypto/engine
 ↪   -I ./s2n/libcrypto-build/openssl/include
 ↪   ./s2n-harness/module/s2n_connection_gs_harness.c ./s2n/tls/s2n_connection.c
 ↪   ./s2n/utils/s2n_timer.c ./s2n/crypto/s2n_rsa.c ./s2n/crypto/s2n_dhe.c
 ↪   ./s2n/crypto/s2n_ecc.c ./s2n/utils/s2n_mem.c ./s2n/utils/s2n_blob.c
 ↪   ./s2n/stuffer/s2n_stuffer.c ./s2n/crypto/s2n_hmac.c ./s2n/crypto/s2n_hash.c
 ↪   ./s2n-lib/openssl/rsa.c ./s2n-lib/openssl/ec.c ./s2n-lib/openssl/dh.c
 ↪   ./s2n-lib/mman.c ./s2n-lib/stdlib.c ./s2n-lib/CommonCrypto/CommonDigest.c
 ↪   ./s2n/tls/s2n_config.c ./s2n-lib/mach_time.c ./s2n/tls/s2n_cipher_suites.c
 ↪   --unwindset
 ↪   s2n_set_server_name.0:257,s2n_get_server_name.0:257,s2n_get_application_protocol.0:257

s2n_connection_harness.c
cbmc -I ./s2n -I ./s2n/api -I ./s2n/libcrypto-build/openssl -I
 ↪   ./s2n/libcrypto-build/openssl/crypto -I ./s2n/libcrypto-build/openssl/crypto/engine
 ↪   -I ./s2n/libcrypto-build/openssl/include
 ↪   ./s2n-harness/module/s2n_connection_harness.c ./s2n/tls/s2n_connection.c
 ↪   ./s2n/utils/s2n_mem.c ./s2n/utils/s2n_blob.c ./s2n/stuffer/s2n_stuffer.c
 ↪   ./s2n/crypto/s2n_rsa.c ./s2n/crypto/s2n_dhe.c ./s2n/crypto/s2n_ecc.c
 ↪   ./s2n/utils/s2n_timer.c ./s2n-lib/CommonCrypto/CommonDigest.c ./s2n-lib/openssl/rsa.c
 ↪   ./s2n-lib/openssl/dh.c ./s2n-lib/openssl/ec.c ./s2n-lib/mman.c ./s2n-lib/stdlib.c
 ↪   --unwind 1
```

Appendix E

Verification Harnesses

E.1 s2n_blob_harness.c

s2n-harness/module/s2n_blob_harness.c

```c
#include "error/s2n_errno.h"
#include "utils/s2n_safety.h"

#include "utils/s2n_blob.h"

uint32_t nondet_uint32();

int main(int argc, char **argv)
{
    uint32_t data_size = nondet_uint32();
    __CPROVER_assume(data_size > 0);

    uint8_t data[data_size];
    struct s2n_blob b = {.data = data, .size = sizeof(data)};

    /* no function actully use this method */
    //    s2n_blob_init(&b, &data, sizeof(data));

    GUARD(s2n_blob_zero(&b));
    return 0;
}
```

E.2 s2n_mem_harness.c

s2n-harness/module/s2n_mem_harness.c

```c
#include "error/s2n_errno.h"
#include "utils/s2n_safety.h"

#include "utils/s2n_mem.h"
#include "utils/s2n_blob.h"

uint32_t nondet_uint32();

int main(int argc, char **argv)
{
    uint32_t blob_size = nondet_uint32();
    __CPROVER_assume(blob_size > 0);

    struct s2n_blob b;

    GUARD(s2n_mem_init());
    GUARD(s2n_alloc(&b, blob_size));
    GUARD(s2n_realloc(&b, nondet_uint32()));
    GUARD(s2n_free(&b));
    return 0;
}
```

E.3 s2n_timer_harness.c

s2n-harness/module/s2n_timer_harness.c

```
#include <stdint.h>
#include "utils/s2n_timer.h"
#include "tls/s2n_config.h"

uint64_t nondet_uint64();

int mock_clock(void *in, uint64_t *out)
{
    *out = *(uint64_t *)in;
    return 0;
}

int main(int argc, char **argv)
{
    struct s2n_config config;
    struct s2n_timer timer;
    uint64_t nanoseconds;
    uint64_t mock_time;
    uint64_t elapsed;

    s2n_config_set_nanoseconds_since_epoch_callback(&config, mock_clock, &mock_time);

    mock_time = 0;
    s2n_timer_start(&config, &timer);

    elapsed = nondet_uint64();
    mock_time += elapsed;

    s2n_timer_reset(&config, &timer, &nanoseconds);
    __CPROVER_assert(nanoseconds == mock_time, "ERROR: s2n_timer_reset");

    elapsed = nondet_uint64();
    __CPROVER_assume(elapsed <= (UINT64_MAX - mock_time));
    mock_time += elapsed;

    s2n_timer_elapsed(&config, &timer, &nanoseconds);
    __CPROVER_assert(nanoseconds == elapsed, "ERROR:s2n_timer_reset");
}
```

E.4 s2n_hash_harness.c

s2n-harness/module/s2n_hash_harness.c

```c
/*
 * Copyright 2014 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License.
 * A copy of the License is located at
 *
 *  http://aws.amazon.com/apache2.0
 *
 * or in the "license" file accompanying this file. This file is distributed
 * on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
 * express or implied. See the License for the specific language governing
 * permissions and limitations under the License.
 */

//#include "s2n_test.h"

#include "error/s2n_errno.h"
#include "utils/s2n_safety.h"

#include <string.h>

//#include "testlib/s2n_testlib.h"
#include "stuffer/s2n_stuffer.h"
#include "crypto/s2n_hash.h"
#include "utils/s2n_blob.h"

uint32_t nondet_uint32();
s2n_hash_algorithm nondet_s2n_hash_algorithm();

//typedef enum { S2N_HASH_NONE, S2N_HASH_MD5, S2N_HASH_SHA1, S2N_HASH_SHA224, S2N_HASH_SHA256, S2N_HASH_SHA384, S2N_HASH_SHA512, S2N_HASH_MD5_SHA1 } s2n_hash_algorithm;

int main(int argc, char **argv)
{
    struct s2n_hash_state hash, copy;
    uint8_t output_pad[64];

    uint32_t length = nondet_uint32();
    __CPROVER_assume(length > 0);

    uint8_t msg[length];

    s2n_hash_algorithm alg = nondet_s2n_hash_algorithm();
    __CPROVER_assume(alg >= 0 && alg <= 7);

    uint8_t digestSize = s2n_hash_digest_size(alg);

    // normal hash flow
    GUARD(s2n_hash_digest_size(alg));
    GUARD(s2n_hash_init(&hash, alg));
    GUARD(s2n_hash_update(&hash, msg, length));
    // only the hash state is changed
    GUARD(s2n_hash_digest(&hash, output_pad, digestSize));
    GUARD(s2n_hash_copy(&copy, &hash));
    return 0;
}
```

E.5 s2n_hmac_harness.c

s2n-harness/module/s2n_hmac_harness.c

```
/*
 * Copyright 2014 Amazon.com, Inc. or its affiliates. All Rights Reserved.
 *
 * Licensed under the Apache License, Version 2.0 (the "License").
 * You may not use this file except in compliance with the License.
 * A copy of the License is located at
 *
 *  http://aws.amazon.com/apache2.0
 *
 * or in the "license" file accompanying this file. This file is distributed
 * on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
 * express or implied. See the License for the specific language governing
 * permissions and limitations under the License.
 */

#include "error/s2n_errno.h"
#include "utils/s2n_safety.h"
//#include "s2n_test.h"

//#include "testlib/s2n_testlib.h"
#include "stuffer/s2n_stuffer.h"
#include "crypto/s2n_hash.h"
#include "crypto/s2n_hmac.h"
#include <string.h>

uint8_t nondet_uint8();
uint32_t nondet_uint32();
s2n_hmac_algorithm nondet_s2n_hmac_algorithm();

int main(int argc, char **argv)
{
    uint8_t output_pad[256];

    uint32_t sekritSize = nondet_uint32();
    __CPROVER_assume(sekritSize > 0 && sekritSize < 1 << 16);

    uint32_t msgSize = nondet_uint32();
    __CPROVER_assume(msgSize > 0);

    uint8_t sekrit[sekritSize];
    uint8_t msg[msgSize];
    struct s2n_hmac_state hmac, copy;

    s2n_hmac_algorithm alg = nondet_s2n_hmac_algorithm();
    __CPROVER_assume(alg >= 0 && alg < 8);

    uint8_t digestSize = s2n_hmac_digest_size(alg);
    GUARD(s2n_hmac_init(&hmac, alg, sekrit, sekritSize));
    GUARD(s2n_hmac_update(&hmac, msg, msgSize));
    GUARD(s2n_hmac_digest(&hmac, output_pad, digestSize));

    GUARD(s2n_hmac_copy(&copy, &hmac));
    return 0;
}
```

E.6 s2n_stuffer_harness.c

s2n-harness/module/s2n_stuffer_harness.c

```
#include "error/s2n_errno.h"
#include "utils/s2n_safety.h"

#include "stuffer/s2n_stuffer.h"

uint32_t nondet_uint32();
uint64_t nondet_uint64();
_Bool nondet_bool();

int main(int argc, char **argv)
{
    uint8_t u8; uint16_t u16; uint32_t u32;

    uint32_t stuffer_size = nondet_uint32();
    __CPROVER_assume(stuffer_size > 0);

    uint32_t entropy_size = nondet_uint32();
    __CPROVER_assume(entropy_size > 0);

    uint8_t entropy[entropy_size];
    struct s2n_stuffer stuffer;

    GUARD(s2n_stuffer_alloc(&stuffer, stuffer_size));

    struct s2n_blob in = {.data = entropy,.size = entropy_size};
    GUARD(s2n_stuffer_write(&stuffer, &in));

    GUARD(s2n_stuffer_wipe(&stuffer));
    while(nondet_bool()) {
        GUARD(s2n_stuffer_write_uint8(&stuffer, nondet_uint64()));
    }

    while(nondet_bool()) {
        GUARD(s2n_stuffer_read_uint8(&stuffer, &u8));
    }

    GUARD(s2n_stuffer_wipe(&stuffer));
    while(nondet_bool()) {
        GUARD(s2n_stuffer_write_uint16(&stuffer, nondet_uint64()));
    }

    while(nondet_bool()) {
        GUARD(s2n_stuffer_read_uint16(&stuffer, &u16));
    }

    GUARD(s2n_stuffer_wipe(&stuffer));
    while(nondet_bool()) {
        GUARD(s2n_stuffer_write_uint24(&stuffer, nondet_uint64()));
    }

    while(nondet_bool()) {
        GUARD(s2n_stuffer_read_uint24(&stuffer, &u32));
    }

    GUARD(s2n_stuffer_wipe(&stuffer));
    while(nondet_bool()) {
        GUARD(s2n_stuffer_write_uint32(&stuffer, nondet_uint64()));
    }

    while(nondet_bool()) {
        GUARD(s2n_stuffer_read_uint32(&stuffer, &u32));
    }

    GUARD(s2n_stuffer_free(&stuffer));
}
```

E.7 s2n_stuffer_base64_harness.c

s2n-harness/module/s2n_stuffer_base64_harness.c

```
#include "error/s2n_errno.h"
#include "utils/s2n_safety.h"

#include "stuffer/s2n_stuffer.h"

#define S2N_PEM_LINE_LENGTH 64
uint32_t nondet_uint32();

int main(int argc, char **argv)
{
    uint32_t message_size = nondet_uint32();
    __CPROVER_assume(message_size > 0 && message_size < 49);

    char message[message_size];
    __CPROVER_assume(message[message_size-1] == '\0');

    uint32_t base64_message_size = message_size * 4 / 3;
    uint32_t scratch_size = S2N_PEM_LINE_LENGTH + 1;

    struct s2n_stuffer stuffer, data, scratch;

    GUARD(s2n_stuffer_alloc(&stuffer, base64_message_size));

    GUARD(s2n_stuffer_alloc_ro_from_string(&data, message));

    GUARD(s2n_stuffer_write_base64(&stuffer, &data));
    GUARD(s2n_stuffer_free(&data));

    GUARD(s2n_stuffer_alloc(&scratch, scratch_size));
    GUARD(s2n_stuffer_read_base64(&stuffer, &scratch));

    GUARD(s2n_stuffer_free(&stuffer));
    GUARD(s2n_stuffer_free(&scratch));

    return 0;
}
```

E.8 s2n_stuffer_text_harness.c

s2n-harness/module/s2n_stuffer_text_harness.c

```
#include "error/s2n_errno.h"
#include "utils/s2n_safety.h"

#include "stuffer/s2n_stuffer.h"

uint32_t nondet_uint32();
_Bool nondet_bool();

#define S2N_PEM_LINE_LENGTH 64

int main(int argc, char **argv)
{
    uint32_t text_size = nondet_uint32();
    __CPROVER_assume(text_size > 0 && text_size < 5120);

    uint32_t read_size = nondet_uint32();
    __CPROVER_assume(text_size > 0 && text_size < 5120);

    char c;
    char pad[5120], out[5120], text[text_size];
    char linepad[S2N_PEM_LINE_LENGTH + 1];

    struct s2n_blob line_blob = {.data = linepad, .size = sizeof(linepad)};
    struct s2n_blob pblob = {.data = pad, .size = sizeof(pad)};

    struct s2n_stuffer stuffer, line;

    GUARD(s2n_stuffer_init(&line, &line_blob));
    GUARD(s2n_stuffer_init(&stuffer, &pblob));

    GUARD(s2n_stuffer_write_text(&stuffer, text, sizeof(text)));
    GUARD(s2n_stuffer_peek_char(&stuffer, &c));
    GUARD(s2n_stuffer_read_text(&stuffer, out, read_size));
    GUARD(s2n_stuffer_read_char(&stuffer, &c));
    GUARD(s2n_stuffer_read_token(&stuffer, &line, '\n'));

    return 0;
}
```

E.9 s2n_override_openssl_random_harness.c

s2n-harness/module/s2n_override_openssl_random_harness.c

```
#include "crypto/s2n_dhe.h"

#include "error/s2n_errno.h"
#include "utils/s2n_safety.h"

```

```c
#include "utils/s2n_random.h"
#include "utils/s2n_blob.h"

#include <openssl/engine.h>
#include <openssl/dh.h>
#include <s2n.h>

static uint8_t dhparams[] =
    "-----BEGIN DH PARAMETERS-----\n"
    "MIIBCAKCAQEAy1+hVWCfNQoPB+NA733IVOONl8fCumiz9zdRRu1hzVa2yvGseUSq\n"
    "Bbn6k0FQ7yMED6w5XWQKDC0z2m0FI/BPE3AjUfuPzEYGqTDf9zQZ2Lz4oAN90Sud\n"
    "luOoEhYR99cEbCn0T4eBvEf9IUtczXUZ/wj7gzGbGG07dLfT+CmCRJxCjhrosenJ\n"
    "gzucyS7jt1bobgU66JKkgMNm7hJY4/nhR5LWTCzZyzYQh2HM2Vk4K5ZqILpj/n0S\n"
    "5JYTQ2PVhxP+Uu8+hICs/8VvM72DznjPZzufADipjC7CsQ4S6x/ecZluFtbb+ZTv\n"
    "HI5CnYmkAwJ6+FSWGaZQDi8bgerFk9RWwwIBAg==\n"
    "-----END DH PARAMETERS-----\n";

int main(int argc, char **argv)
{
    struct s2n_stuffer dhparams_in, dhparams_out;
    struct s2n_dh_params dh_params;
    struct s2n_blob b = {.data=dhparams, .size=sizeof(dhparams)};

    __CPROVER_assert(s2n_get_private_random_bytes_used() == 0, "ERROR:
 ↪  s2n_get_private_random_bytes_used()");

    /* Parse the DH params */
    GUARD(s2n_stuffer_alloc(&dhparams_in, sizeof(dhparams)));
    GUARD(s2n_stuffer_alloc(&dhparams_out, sizeof(dhparams)));
    GUARD(s2n_stuffer_write(&dhparams_in, &b));
    GUARD(s2n_stuffer_dhparams_from_pem(&dhparams_in, &dhparams_out));

    b.size = s2n_stuffer_data_available(&dhparams_out);
    b.data = s2n_stuffer_raw_read(&dhparams_out, b.size);
    GUARD(s2n_pkcs3_to_dh_params(&dh_params, &b));
    GUARD(s2n_dh_generate_ephemeral_key(&dh_params));

    /* Verify that our DRBG is called and that over-riding works */
    __CPROVER_assert(s2n_get_private_random_bytes_used() != 0, "ERROR:
 ↪  s2n_get_private_random_bytes_used()");

    GUARD(s2n_dh_params_free(&dh_params));
    GUARD(s2n_stuffer_free(&dhparams_out));
    GUARD(s2n_stuffer_free(&dhparams_in));
}
```

E.10 s2n_drbg_harness.c

s2n-harness/module/s2n_drbg_harness.c

```c
#include "error/s2n_errno.h"
#include "utils/s2n_safety.h"

#include <inttypes.h>
#include <fcntl.h>
#include <s2n.h>
```

```c

#include <openssl/aes.h>

#include "crypto/s2n_drbg.h"

#include "utils/s2n_safety.h"
#include "utils/s2n_random.h"
#include "utils/s2n_timer.h"

#include "tls/s2n_config.h"

/* Test vectors are taken from
 ↪  http://csrc.nist.gov/groups/STM/cavp/documents/drbg/drbgtestvectors.zip
 * - drbgvectors_no_reseed/CTR_DRBG.txt :
 * [AES-128 no df]
 * [PredictionResistance = False]
 * [EntropyInputLen = 256]
 * [NonceLen = 0]
 * [PersonalizationStringLen = 256]
 * [AdditionalInputLen = 0]
 * [ReturnedBitsLen = 512]
 */

struct s2n_stuffer nist_reference_entropy;
const char nist_reference_entropy_hex[480];
int entropy_fd = -1;

const char nist_reference_personalization_strings_hex[480];

const char nist_reference_values_hex[720];

const char nist_reference_returned_bits_hex[960];

/* This function over-rides the s2n internal copy of the same function */
int nist_fake_urandom_data(struct s2n_blob *blob)
{
    /* At first, we use entropy data provided by the NIST test vectors */
    GUARD(s2n_stuffer_read(&nist_reference_entropy, blob));

    return 0;
}

int main(int argc, char **argv)
{
    uint8_t data[256] = { 0 };
    struct s2n_drbg drbg = {{ 0 }};
    struct s2n_blob blob = {.data = data, .size = 16 };
    struct s2n_timer timer;
    uint64_t drbg_nanoseconds;
    uint64_t urandom_nanoseconds;
    struct s2n_stuffer nist_reference_personalization_strings;
    struct s2n_stuffer nist_reference_returned_bits;
    struct s2n_stuffer nist_reference_values;
    struct s2n_config *config;

    config = s2n_config_new();
    __CPROVER_assert(config != NULL, "ERROR: s2n_config_new()");

    /* Open /dev/urandom */
    entropy_fd = open("/dev/urandom", O_RDONLY);

    /* Convert the hex entropy data into binary */
    GUARD(s2n_stuffer_alloc(&nist_reference_entropy, 480));
    GUARD(s2n_stuffer_alloc(&nist_reference_personalization_strings, 480));
    GUARD(s2n_stuffer_alloc(&nist_reference_returned_bits, 960));
    GUARD(s2n_stuffer_alloc(&nist_reference_values, 720));

    /* Check everything against the NIST vectors */
    for (int i = 0; i < 14; i++) {
        uint8_t ps[32];
        struct s2n_drbg nist_drbg = { .entropy_generator = nist_fake_urandom_data };
        struct s2n_blob personalization_string = {.data = ps, .size = 32};

        /* Read the next personalization string */
        GUARD(s2n_stuffer_read(&nist_reference_personalization_strings, &personalization_string));

        /* Instantiate the DRBG */
        GUARD(s2n_drbg_instantiate(&nist_drbg, &personalization_string));

        uint8_t nist_v[16];

        GUARD(s2n_stuffer_read_bytes(&nist_reference_values, nist_v, sizeof(nist_v)));

        /* Generate 512 bits (FIRST CALL) */
        uint8_t out[64];
        struct s2n_blob generated = {.data = out, .size = 64 };
        GUARD(s2n_drbg_generate(&nist_drbg, &generated));

        GUARD(s2n_stuffer_read_bytes(&nist_reference_values, nist_v, sizeof(nist_v)));

        /* Generate another 512 bits (SECOND CALL) */
        GUARD(s2n_drbg_generate(&nist_drbg, &generated));

        GUARD(s2n_stuffer_read_bytes(&nist_reference_values, nist_v, sizeof(nist_v)));

        uint8_t nist_returned_bits[64];
        GUARD(s2n_stuffer_read_bytes(&nist_reference_returned_bits, nist_returned_bits, sizeof(nist_returned_bits)));

        GUARD(s2n_drbg_wipe(&nist_drbg));
    }

    GUARD(s2n_drbg_instantiate(&drbg, &blob));

    /* Use the DRBG for 16MB of data */
    GUARD(s2n_timer_start(config, &timer));
    for (int i = 0; i < 1000000; i++) {
        GUARD(s2n_drbg_generate(&drbg, &blob));
    }
    GUARD(s2n_timer_reset(config, &timer, &drbg_nanoseconds));

    /* Use urandom for 16MB of data */
    GUARD(s2n_timer_start(config, &timer));
    for (int i = 0; i < 1000000; i++) {
        GUARD(s2n_get_urandom_data(&blob));
    }
    GUARD(s2n_timer_reset(config, &timer, &urandom_nanoseconds));

    /* NOTE: s2n_random_test also includes monobit tests for this DRBG */

    /* the DRBG state is 128 bytes, test that we can get more than that */
    blob.size = 129;
    for (int i = 0; i < 10; i++) {
        GUARD(s2n_drbg_generate(&drbg, &blob));
    }

    /* Move the DRBG to where it would be just before a reseed */
    drbg.bytes_used = S2N_DRBG_RESEED_LIMIT - 128;
    for (int i = 0; i < 10; i++) {
        GUARD(s2n_drbg_generate(&drbg, &blob));
    }

    GUARD(s2n_drbg_wipe(&drbg));

    GUARD(s2n_stuffer_free(&nist_reference_entropy));
    GUARD(s2n_stuffer_free(&nist_reference_personalization_strings));
    GUARD(s2n_stuffer_free(&nist_reference_returned_bits));
    GUARD(s2n_stuffer_free(&nist_reference_values));

//    END_TEST();
}
```

E.11 s2n_ecc_harness.c

s2n-harness/module/s2n_ecc_harness.c

```
#include "error/s2n_errno.h"
#include "utils/s2n_safety.h"

#include <s2n.h>

#include "crypto/s2n_ecc.h"
#include "utils/s2n_mem.h"

int main(int argc, char **argv)
{
    /* Test generate->write->read->compute_shared with all supported curves */
    for (int i = 0; i < sizeof(s2n_ecc_supported_curves) / sizeof(s2n_ecc_supported_curves[0]); i++) {
        struct s2n_ecc_params server_params, client_params;
        struct s2n_stuffer wire;
        struct s2n_blob server_shared, client_shared, ecdh_params_sent, ecdh_params_received;

        GUARD(s2n_stuffer_growable_alloc(&wire, 1024));

        /* Server generates a key for a given curve */
        server_params.negotiated_curve = &s2n_ecc_supported_curves[i];
        GUARD(s2n_ecc_generate_ephemeral_key(&server_params));
        /* Server sends the public */
        GUARD(s2n_ecc_write_ecc_params(&server_params, &wire, &ecdh_params_sent));
        /* Client reads the public */
```

```c
        GUARD(s2n_ecc_read_ecc_params(&client_params, &wire, &ecdh_params_received));

        /* Client sends its public */
        GUARD(s2n_ecc_compute_shared_secret_as_client(&client_params, &wire, &client_shared));
        /* Server receives it */
        GUARD(s2n_ecc_compute_shared_secret_as_server(&server_params, &wire, &server_shared));
        /* Shared is the same for the client and the server */
        __CPROVER_assert(client_shared.size == server_shared.size, "ERROR: Shared size are not the same.");

        int j;
        __CPROVER_assume(j >= 0 && j < client_shared.size);
        __CPROVER_assert(client_shared.data[j] == server_shared.data[j], "ERROR: Shared data are not the same." );

        /* Clean up */
        GUARD(s2n_stuffer_free(&wire));
        GUARD(s2n_free(&server_shared));
        GUARD(s2n_free(&client_shared));
        GUARD(s2n_ecc_params_free(&server_params));
        GUARD(s2n_ecc_params_free(&client_params));
    }
    return 0;
}
```

E.12 s2n_config_client_harness.c

s2n-harness/module/s2n_config_client_harness.c

```c
#include "error/s2n_errno.h"
#include "utils/s2n_safety.h"

#include <s2n.h>
#include <string.h>

int main(){
    s2n_status_request_type type;
    __CPROVER_assume(type == S2N_STATUS_REQUEST_NONE || type == S2N_STATUS_REQUEST_OCSP);

    struct s2n_config *config = s2n_config_new();
    __CPROVER_assert(config != NULL, "ERROR: s2n_config_new()");

    GUARD(s2n_config_set_status_request_type(config, type));
    GUARD(s2n_config_free(config));

    return 0;
}
```

E.13 s2n_config_server_harness.c

s2n-harness/module/s2n_config_server_harness.c

```c
#include "error/s2n_errno.h"
#include "utils/s2n_safety.h"

#include <s2n.h>
#include <string.h>

static char certificate[10];
static char private_key[10];

int main(){
    struct s2n_config *config = s2n_config_new();
    __CPROVER_assert(config != NULL, "ERROR: s2n_config_new()");

    GUARD(s2n_config_add_cert_chain_and_key(config, certificate, private_key));
    GUARD(s2n_config_free(config));
    return 0;
}
```

E.14 s2n_config_dhparams_harness.c

s2n-harness/module/s2n_config_dhparams_harness.c

```c
#include "error/s2n_errno.h"
#include "utils/s2n_safety.h"

#include <s2n.h>
#include <string.h>

static char dhparams[] =
"-----BEGIN DH PARAMETERS-----\n"
"MIIBCAKCAQEAy1+hVWCfNQoPB+NA733IVOONl8fCumiz9zdRRu1hzVa2yvGseUSq\n"
"Bbn6k0FQ7yMED6w5XWQKDC0z2m0FI/BPE3AjUfuPzEYGqTDf9zQZ2Lz4oAN90Sud\n"
"luOoEhYR99cEbCn0T4eBvEf9IUtczXUZ/wj7gzGbGG07dLfT+CmCRJxCjhrosenJ\n"
"gzucyS7jt1bobgU66JKkgMNm7hJY4/nhR5LWTCzZyzYQh2HM2Vk4K5ZqILpj/n0S\n"
"5JYTQ2PVhxP+Uu8+hICs/8VvM72DznjPZzufADipjC7CsQ4S6x/ecZluFtbb+ZTv\n"
"HI5CnYmkAwJ6+FSWGaZQDi8bgerFk9RWwwIBAg==\n"
"-----END DH PARAMETERS-----\n";

int main(){
    struct s2n_config *config = s2n_config_new();
    __CPROVER_assert(config != NULL, "ERROR: s2n_config_new()");

    GUARD(s2n_config_add_dhparams(config, dhparams));
    GUARD(s2n_config_free(config));
    return 0;
}
```

E.15 s2n_harness.c

s2n-harness/module/s2n_harness.c

```c
#include "error/s2n_errno.h"
#include "utils/s2n_safety.h"
#include <s2n.h>

int main(int arc, char * const *argv){
        GUARD(s2n_init());
        GUARD(s2n_cleanup());
    return 0;
}
```

E.16 s2n_connection_harness.c

s2n-harness/module/s2n_connection_harness.c

```c
#include <s2n.h>

int main(){
    struct s2n_connection *conn;
    conn = s2n_connection_new(S2N_SERVER);
    return 0;
}
```

E.17 s2n_connection_gs_harness.c

s2n-harness/module/s2n_connection_gs_harness.c

```c
#include "error/s2n_errno.h"
#include "utils/s2n_safety.h"

#include <s2n.h>
#include "tls/s2n_connection.h"
#include "tls/s2n_cipher_suites.h"

uint8_t nondet_uint8();
int nondet_int();

int main(int argc, char **argv)
{
    struct s2n_connection conn;
    const char server_name[256];
    uint8_t i;
    __CPROVER_assume(i < 256 && server_name[i] == '\0');

    struct s2n_blob blob = {.data=conn.alert_in_data, .size=S2N_ALERT_LENGTH};
    s2n_stuffer_alloc(&conn.alert_in, &blob);

    conn.active.cipher_suite = &s2n_null_cipher_suite;

    s2n_blinding blinding;
    __CPROVER_assume(blinding == S2N_BUILT_IN_BLINDING || blinding == S2N_SELF_SERVICE_BLINDING);

    GUARD(s2n_connection_set_config(&conn, &s2n_default_config));
    GUARD(s2n_connection_set_fd(&conn, nondet_int()));
    GUARD(s2n_connection_set_read_fd(&conn, nondet_int()));
    GUARD(s2n_connection_set_write_fd(&conn, nondet_int()));

    s2n_connection_prefer_throughput(&conn);
    s2n_connection_prefer_low_latency(&conn);

    GUARD(s2n_connection_set_blinding(&conn, blinding));
    s2n_connection_get_delay(&conn);

    GUARD(s2n_set_server_name(&conn, server_name));
    s2n_get_server_name(&conn);
    s2n_get_application_protocol(&conn);

    s2n_connection_get_wire_bytes_in(&conn);
     s2n_connection_get_wire_bytes_out(&conn);
    s2n_connection_get_client_protocol_version(&conn);
     s2n_connection_get_server_protocol_version(&conn);
     s2n_connection_get_actual_protocol_version(&conn);
     s2n_connection_get_client_hello_version(&conn);
     s2n_connection_get_cipher(&conn);
     s2n_connection_get_alert(&conn);

    return 0;
}
```

Appendix F

Helper tools

F.1 function_based_analyzer.sh

helper-tools/function_based_analyzer.sh

```bash
#!/bin/bash
if [ ! -z "$1" ]; then
        src="$1"
else
        src="."
fi

timestamp=$(date +%Y%m%d%H%M%S)
dir=./s2n-analysis/$timestamp
mkdir $dir
filename="$dir/results.csv"
regex="^(\w+ )+\*?\w+\(.*\)$"
buildInChecks=(--bounds-check --div-by-zero-check --pointer-check --memory-leak-check
 ↪   --signed-overflow-check --unsigned-overflow-check --float-overflow-check)
echo "filenName, functionName, --bounds-check, --div-by-zero-check, --pointer-check,
 ↪   --memory-leak-check, --signed-overflow-check, --unsigned-overflow-check,
 ↪   --float-overflow-check," >> $filename;
for f in $(find $src -name ’*.c’ -not -path ’*bin*’ -not -path ’*test*’);
do
        grep -E ’^(\w+ )+\*?\w+\(.*\)$’ $f | cut -d "(" -f 1 | tr -d "*" | while read
 ↪   -r line; do
        echo -n "$f," >> $filename;
        echo -n "${line##* }," >> $filename;
        for check in "${buildInChecks[@]}"
        do
                result=$(cbmc -I ./s2n -I ./s2n/api -I ./s2n/libcrypto-root/include
 ↪   $check $f --function ${line##* } --unwind 10 --no-assertions --no-assumptions)
                if echo $result | grep -q "VERIFICATION SUCCESSFUL"; then
                        echo -n "1," >> $filename;
                else
                        echo -n "0," >> $filename;
                        echo "$result" >> $dir/${line##* }$check.txt;
                fi
        done
        echo >> $filename;
done
#echo -e " " >> $filename;
```

```
done
```

F.2 public_function_scanner.sh

s2n-harness/helper-tools/public_function_scanner.sh

```
#!/bin/bash
filename="public_functions.txt"
result=""
for f in $(find ./s2n -type f -name "*.h" -not -path '*bin*' -not -path '*tests*' -not
 ↪  -path '*libcrypto-build*' -not -path '*libcrypto-root*'); do
    result="$result$(grep -Eo '\w+\(.+\);$' $f)";
done

echo $result | tr ";" "\n" | sort | uniq | while read -r line; do
        echo "- [ ] ${line%%(*}" >> $filename;
done
```

F.3 external_function_scanner.sh

s2n-harness/helper-tools/external_function_scanner.sh

```
#!/bin/bash
filename="externalFunctions.txt"
result=""
for f in $(find ./s2n -name '*.c' -not -path '*bin*' -not -path '*test*' -not -path
 ↪  '*libcrypto-build*' -not -path '*libcrypto-root*');
do
       result="$result $(grep -Eo '(\w+)\(' $f | grep -v -E
 ↪  '(s2n|S2N|CPROVER|GUARD|if|while|switch|_check|defined)' | cut -d '(' -f 1)"
done
echo $result | tr " " "\n" | sort | uniq > $filename;
```

Bibliography

[1] The s2n user manual. `https://github.com/awslabs/s2n/blob/master/docs/USAGE-GUIDE.md`.

[2] Damiano Angeletti, Enrico Giunchiglia, Massimo Narizzano, Alessandra Puddu, and Salvatore Sabina. *Automatic Test Generation for Coverage Analysis Using CBMC*, pages 287–294. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[3] Thoms Ball. The concept of dynamic analysis. *SIGSOFT Softw. Eng. Notes*, 24(6):216–234, October 1999.

[4] Bernhard Beckert, Thorsten Bormer, Michael Kirsten, Till Neuber, and Mattias Ulbrich. Automated verification for functional and relational properties of voting rules. In *Sixth International Workshop on Computational Social Choice (COMSOC 2016)*, Toulouse, France, June 2016.

[5] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[6] M. Brain and D. Kroening. The software verification pyramid. Submitted to the Royal Society Philosophical Transactions A.

[7] D. Bucur and M. Kwiatkowska. On software verification for sensor nodes. *Journal of Software and Systems*, 84(10):1693–1707, 2011.

[8] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.*, 19(1):7–34, July 2001.

[9] Edmund Clarke and Daniel Kroening. Hardware verification using ansi-c programs as a reference. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, ASP-DAC '03, pages 308–311, New York, NY, USA, 2003. ACM.

[10] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[11] Edmund M. Clarke. 25 years of model checking. chapter The Birth of Model Checking, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2008.

[12] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[13] L. Cordeiro, B. Fischer, H. Chen, and J. Marques-Silva. Semiformal verification of embedded software in medical devices considering stringent hardware constraints. In *Embedded Software and Systems, 2009. ICESS '09. International Conference on*, pages 396–403, May 2009.

[14] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. Smt-based bounded model checking for embedded ansi-c software. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 137–148, Washington, DC, USA, 2009. IEEE Computer Society.

[15] P. Cousot and R. Cousot. *A gentle introduction to formal verification of computer systems by abstract interpretation*, pages 1–29. NATO Science Series III: Computer and Systems Sciences. IOS Press, 2010.

[16] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does astrée scale up? *Formal Methods in System Design*, 35(3):229–264, December 2009.

[17] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, July 2008.

[18] Jean-Christophe Filliâtre. Deductive software verification. *International Journal on Software Tools for Technology Transfer*, 13(5):397–403, 2011.

[19] B. Fischer, O. Inverso, and G. Parlato. Cseq: A concurrency pre-processor for sequential c verification tools. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 710–713, Nov 2013.

[20] Andreas Griesmayer, Stefan Staber, and Roderick Bloem. Automated fault localization for c programs. *Electron. Notes Theor. Comput. Sci.*, 174(4):95–111, May 2007.

[21] A. Groce, I. Ahmed, C. Jensen, and P. E. McKenney. How verified is my code? falsification-driven verification (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 737–748, Nov 2015.

[22] Reiner Hähnle. Introduction deductive software verification.

[23] Antti Jääskeläinen, Mika Katara, Shmuel Katz, and Heikki Virtanen. Verification of safety-critical systems: A case study report on using modern model checking tools. In Jörg Brauer, Marco Roveri, and Hendrik Tews, editors, *6th International Workshop on Systems Software Verification*, volume 24 of *OpenAccess Series in Informatics (OASIcs)*, pages 44–56, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[24] M. Kim, Y. Kim, and H. Kim. Unit testing of flash memory device driver through a sat-based model checker. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 198–207, Sept 2008.

[25] M. Kim, Y. Kim, and H. Kim. A comparative study of software model checkers as unit testing tools: An industrial case study. *IEEE Transactions on Software Engineering*, 37(2):146–160, March 2011.

[26] Moonzoo Kim, Yunja Choi, Yunho Kim, and Hotae Kim. Formal verification of a flash memory device driver — an experience report. In *Proceedings of the 15th International Workshop on Model Checking Software*, SPIN '08, pages 144–159, Berlin, Heidelberg, 2008. Springer-Verlag.

[27] Moonzoo Kim and Yunho Kim. Formal methods: Foundations and applications. chapter Concolic Testing of the Multi-sector Read Operation for Flash Memory File System, pages 251–265. Springer-Verlag, Berlin, Heidelberg, 2009.

[28] Y. Kim and M. Kim. Sat-based bounded software model checking for embedded software: A case study. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 55–62, Dec 2014.

[29] Shrawan Kumar, Bharti Chimdyalwar, and Ulka Shrotri. Precise range analysis on large industry code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 675–678, New York, NY, USA, 2013. ACM.

[30] David Kung and Hong Zhu. Software verification and validation. *Wiley Encyclopedia of Computer Science and Engineering*, 2008.

[31] Dong-Ah Lee, Junbeom Yoo, and Jang-Soo Lee. *Equivalence Checking between Function Block Diagrams and C Programs Using HW-CBMC*, pages 397–408. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[32] Mateus Krepsky Ludwich and Antônio Augusto Fröhlich. On the formal verification of component-based embedded operating systems. *SIGOPS Oper. Syst. Rev.*, 47(1):28–34, January 2013.

[33] Florian Merz, Carsten Sinz, Hendrik Post, Thomas Gorges, and Thomas Kropf. Bridging the gap between test cases and requirements by abstract testing. *Innov. Syst. Softw. Eng.*, 11(4):233–242, December 2015.

[34] Ravindra Metta. Verifying code and its optimizations: An experience report. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 578–583, Washington, DC, USA, 2011. IEEE Computer Society.

[35] Netcraft. 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`, April 2014.

[36] Mingyu Park, Taejoon Byun, and Yunja Choi. Property-based code slicing for efficient verification of OSEK/VDX operating systems. In *Proceedings First International Workshop on Formal Techniques for Safety-Critical Systems, FTSCS 2012, Kyoto, Japan, November 12, 2012.*, pages 69–84, 2012.

[37] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 347–350, Sept 2008.

[38] H. Post, C. Sinz, A. Kaiser, and T. Gorges. Reducing false positives by combining abstract interpretation and bounded model checking. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 188–197, Sept 2008.

[39] H. Rocha, H. Ismail, L. Cordeiro, and R. Barreto. Model checking embedded c software using k-induction and invariants. In *2015 Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 90–95, Nov 2015.

[40] M. Sagiv. Lecture notes in automatic software verification: Bounded model checking, April 2015.

[41] S. Schmidt. Introducing s2n, a new open source tls implementation. `https://blogs.aws.amazon.com/security/post/TxCKZM94ST1S6Y/Introducing-s2n-a-NewOpen-Source-TLS-Implementation`.

[42] J. Toman, S. Pernsteiner, and E. Torlak. Crust: A bounded verifier for rust (n). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 75–80, Nov 2015.

[43] Mayur Khanzode Kumar Madhukar Tukaram Muske, Advaita Datar. Efficient elimination of false positives using bounded model checking. In *VALID 2013, The Fifth International Conference on Advances in System Testing and Validation Lifecycle*, pages 13–to. IARIA, 2013, 2013.

[44] Moshe Y. Vardi. *Model Checking for Database Theoreticians*, pages 1–16. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[45] Markus Voelter, Zaur Molotnikov, and Bernd Kolb. Towards improving software security using language engineering and mbeddr c. In *Proceedings of the Workshop on Domain-Specific Modeling*, DSM 2015, pages 55–62, New York, NY, USA, 2015. ACM.

[46] Kostyantyn Vorobyov and Padmanabhan Krishnan. Comparing model checking and static program analysis: A case study in error detection approaches. *Proceedings of SSV*, 2010.

[47] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. *Cross-platform verification framework for embedded systems*, pages 137–148. Lecture Notes in Computer Science. Springer, 9 2007.

[48] Frank Werner and David Faragó. Correctness of sensor network applications by software bounded model checking. In *FMICS*, 2010.

[49] C. Williams. Anatomy of openssl's heartbleed: Just four bytes trigger horror bug. `http://www.theregister.co.uk/2014/04/09/heartbleed_explained/l`.

[50] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, November 2006.

[51] Anand Yeolekar and Divyesh Unadkat. *Assertion Checking Using Dynamic Inference*, pages 199–213. Springer International Publishing, Cham, 2013.

[52] Y. Zhang, F. Xie, Y. Dong, X. Zhou, and C. Ma. Cyber/physical co-verification for developing reliable cyber-physical systems. In *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*, pages 539–548, July 2013.