**EBF: A Hybrid Verification Tool for Finding Software Vulnerabilities in IoT Cryptographic Protocols**

Fatimah Aljaafari¹, Lucas C. Cordeiro¹, Mustafa A. Mustafa¹, and Rafael Menezes²

¹ The University of Manchester, UK
² Federal University of Amazonas, Brazil

**Abstract.** Internet of Things (IoT) consists of a large number of smart devices connected through a network, which creates a vast amount of data communication, thereby posing new security, privacy, and trust issues. One way to address these issues is ensuring data confidentiality using lightweight encryption algorithms for IoT protocols. However, the design and implementation of such protocols is an error-prone task; flaws in the implementation can lead to devastating security vulnerabilities. Here we propose a new verification approach named Encryption-BMC and Fuzzing (EBF), which combines Bounded Model Checking (BMC) and Fuzzing techniques to verify software and detect security vulnerabilities exploited by an attacker concerning users’ privacy and integrity. EBF models IoT protocols as a client and server using POSIX threads, thereby simulating both entities’ communication. It also employs static and dynamic verification to cover the system’s state-space exhaustively. We evaluate EBF using the concurrency benchmarks from SV-COMP and show that it outperforms other state-of-the-art tools such as ESCBMC, AFL, Lazy-CSeq, and TSAN w.r.t. bug finding. We also evaluate an open-source implementation called WolfMQTT. It is an MQTT client implementation that uses the WolfSSL library. We show that EBF detects a data race, which other approaches are unable to identify.

1 Introduction

An Internet of Things (IoT) system usually comprises a large number of smart devices and objects, such as RFID tags, sensors, actuators, and smartphones, which communicate with each other (usually via Wifi, Bluetooth, and RFID) with minimum human interventions [33]. IoT covers different areas and applications, such as smart homes, cities, and health care [22]. According to Mehavarunan [29], from 2020 to 2030, IoT devices will grow from 75 billion to more than 100 billion, the upgrade from 4G to 5G playing an important part in this growth. This large number of devices will create a massive and complex network with an exceedingly high volume of data communicated over it [15, 52]. The existence of such a vast network of connected devices will inevitably pose new security, privacy and trust issues that can put users at high risk [33].
To address these issues, achieving *data confidentiality* is paramount. A natural way to do so is protecting the data in transit by designing bespoke lightweight encryption algorithms for IoT devices [13]. Due to limitations in IoT devices such as limited power supply, low memory life, and low processing speed [10,31], lightweight encryption algorithms have been developed. For example, WolfSSL is a library targeted at resource-constrained devices due to its small size, speed, and feature set [1]. It provides lightweight implementations that support TLS/SSL, which support various cryptographic algorithms, including lightweight encryption algorithms. However, designing and implementing such algorithms for IoT protocols is an error-prone task [36]; flaws in the implementation can lead to devastating security vulnerabilities [36].

Generally, there exist various techniques for finding security vulnerabilities [20,27]. One of them is Bounded Model Checking (BMC) [17]. The basic idea of BMC is to search for violation in bounded executions of length k. If no bug is detected, then k is increased until a bug is detected, the verification problem becomes intractable, or a pre-set upper bound is reached. Some examples of BMC tools include C Bounded Model Checker CBMC [26] and Efficient SMT-based Bounded Model Checker ESBMC [2]. Another popular technique is fuzzing [27]. It is an automated software testing technique that involves providing invalid values as inputs to a program. Then, the system behavior is checked for abnormalities, such as crashes or failures [35]. American Fuzzing Lop (AFL) [3] and *LibFuzzer* [5] are some of the state-of-the-art tools that implement fuzzing.

In contrast to Ognawala et al. [30], who combine symbolic execution and fuzzing and apply it to general-purpose software, EBF starts with BMC. It then uses fuzzing by considering intricate security properties in IoT protocols. We model (simulate) the client and server in the protocol’s communication as two threads and instrument the program to exploit different thread interleavings. Besides, the nature of cryptographic algorithms involves math operations on a vast space state that makes it non-trivial for SMT solvers to solve [36]. EBF employs techniques, such as constant folding, bound k, and induction techniques, to reduce the number of states needed to be verified. Lastly, some cryptographic libraries rely on UNIX sockets and file operations to encode and decode text. Since BMC approaches require models for the environment, EBF relies on dynamic techniques such as fuzzing to explore these paths in the deployed environment.

Here we develop a novel verification method named Encryption-BMC and Fuzzing (EBF), which exploits BMC and fuzzing to detect security vulnerabilities in IoT protocols’ implementations. Since client and server in communication protocols behave like different threads and some implementations of encryption libraries are using POSIX threads in their implementation [1], we develop our verification method to detect security vulnerabilities in multi-thread programs. In particular, we exploit BMC techniques to provide valuable seeds to our fuzzing approach to discover different thread interleavings, which make fuzzing detect vulnerabilities more efficiently [37]. We also extend existing fuzzing methods, such as AFL [3], that have difficulties of detecting bugs in multi-threaded im-
implementations due to the various thread interleavings by employing a program’s instrumentation using the LLVM pass \cite{6} to introduce random delays.

We evaluate EBF on five subcategories of the SV-COMP concurrency benchmarks\cite{7}. We compare the results with ESBMC \cite{2}, AFL \cite{3}, Lazy Sequentialization (LazyCSeq) \cite{25}, and Thread Sanitizer (TSAN) \cite{4} on the same benchmarks. We also examine the EBF tool by verifying the WolfMQTT implementation. The experimental results show that EBF outperforms the state-of-the-art verification tools as it detects vulnerabilities in more programs than the other tools. We make the following two significant contributions towards the verification of cryptographic protocols targeted for IoT devices and concurrency implementations:

- We propose a new hybrid verification method, named EBF, that combines BMC and fuzzing in an unprecedented manner to increase code coverage and detect both memory corruption and concurrency vulnerabilities of IoT cryptographic protocols and concurrency implementations.
- We implement EBF to verify cryptographic IoT protocols and concurrency implementations. We show that EBF can find vulnerabilities that other existing tools, such as ESBMC \cite{2}, AFL \cite{3}, LazyCSeq \cite{25}, and TSAN \cite{4}, are unable to identify. EBF can also detect a data race in the open-source library WolfMQTT \cite{7} and a thread leak in one example of WolfSSL implementation \cite{1}, where other approaches are unable to identify it.

\section{Verification Methods for Cryptographic Protocols and Concurrent programs}

Many IoT applications depend on cryptographic protocols for secure communication. Mostly the C programming language is used in writing cryptographic software \cite{12}. However, implementation of cryptographic primitives such as encryption and hash functions are error-prone; flaws in the implementation can lead to devastating security vulnerabilities, which may be exploited by an attacker \cite{33}. Recent years have seen a real development in software verification of cryptographic protocols and concurrency applications, as witnessed by the development of different tools using either BMC or fuzzing techniques \cite{12}. However, there still exists a need for further development of these tools. Although the obligation to verify cryptographic protocols and concurrent programs is now well identified, only a few recent studies suggest solutions.

One of the attempts to verify cryptographic primitives using symbolic execution is suggested by Vanhoef and Piessens \cite{35} who modified the KLEE tool \cite{19} to efficiently handle cryptographic protocol by simulating their behavior under the Dolev-Yao model. Similarly, Given-Wilson et al. \cite{24} proposed a process using model checking to detect fault injection vulnerabilities in the PRESENT cipher binary. In the authors’ framework, they used MC-Sema, which supports only some of the X86 architecture; they combined LLBMC with MC-Sema, which does not generate a trace to understand vulnerabilities and analyze results.

\footnote{https://github.com/sosy-lab/sv-benchmarks}
Another tool used for cryptographic primitives verification, based on fuzzing, is CDF [14]. It is used to achieve security verification, and in particular, to find logic bugs with standard specifications. It uses differential fuzzing technique to find inconsistencies between two implementations of the same primitive, e.g., of the RSA cipher [14]. Thus, it is only useful when different implementations of the same algorithm containing the same bug are available. ESPIKE [18] is another fuzzing tool, an extension of SPIKE [11], designed to handle secure protocols by sending all the SPIKE data through the SSL layer [18]. Its limitation is that it is only valid for the already compatible protocols with SPIKE.

Concerning concurrent programs, a few attempts have been proposed to detect security vulnerabilities. The challenge with a multi-threaded program is that it contains different thread interleavings, which may introduce bugs (e.g., data race) that are difficult to detect. MUZZ [21] is a recent tool suggested for fuzzing concurrent programs. It is a grey box fuzzing tool that detects bugs in a multi-threaded program using thread-aware instrumentation. MUZZ, similarly to EBF, instruments the code using LLVM pass to detect concurrency bugs. However, EBF uses a BMC technique to analyze the code and generate inputs, which can help the fuzzer trigger intricate execution paths. ConAFL [28] is another thread-aware grey box fuzzer that focuses on user-space multi-threaded programs. It also uses heavy thread-aware static and dynamic analysis, which causes scalability issues. ConAFL employs static analysis to locate sensitive concurrent operations to determine the execution order, focusing on three types of vulnerabilities: buffer-overflow, double-free, or use-after-free. In addition to ConAFL, EBF focuses on detecting memory corruption bugs and concurrency bugs for maximum detection, i.e., for more inclusiveness.

3 EBF Design and Implementation

We develop a novel verification method to detect memory corruption vulnerabilities such as buffer overflow and memory leak and concurrency bugs such as data races and thread leak using BMC and fuzzing techniques. We build the EBF verification method on top of two tools, ESBMC and AFL to implement our methodology; these tools have been chosen based on the comparison by Beyer et al. [16] between model checking and testing. Figure 1 illustrates the EBF verification method, which consists of three phases: initial inputs generation, instrumentation, and fuzzing.

Overview. In the beginning, EBF uses ESBMC for initial state exploration to search for memory leaks and buffer overflows. We provide ESBMC with the Program under Test (PUT) and specific properties (P). If ESBMC detects a property violation and generates a counterexample, EBF extracts the assumption values and feeds them to the fuzzer as inputs to find unexpected paths that may expose a vulnerability. In case ESBMC fails to detect a violation, then EBF generates random inputs to feed it to the fuzzer. As a second phase, we instrument the PUT using a custom LLVM pass to track the active threads and
inject a delay function after each instruction at runtime. **EBF** feeds the inputs and the instrumented program to **AFL** with **TSAN**’s help to search for additional bugs, especially concurrency bugs. Thus, **EBF** analyzes the results; if there is a bug, it states "Verification Failed"; otherwise, it states "Verification Successful".

**Input Generation phase.** This phase builds on top of **ESBMC**. The user feeds the **EBF** with the source code that needs to be tested with the specified properties (i.e., **unreach-call**, **valid-memsafety** and **no-overflow**). When the user sets **unreach-call** property, it means there is a particular function call in the code that must be unreachable. When **valid-memsafety** is chosen, a specific memory safety property must hold in the code. Also, **no-overflow** propriety means there is a certain kind of undefined behavior (i.e., overflows of signed integer) that must not exist in the code. Algorithm 1 describes the BMC workflow in the **EBF** verification method. Note that **ESBMC** takes the two parameters, the PUT and P, which need to be checked. From line 1 to 4, **ESBMC** simplifies the PUT to a control flow (GOTO program). Then, it converts the GOTO program to a single static assignment (SSA) form. In line 5, it converts SSA into quantifier-free formula ($C \land \neg P$), where $C$ denotes constraints and $P$ denotes properties. Then, the SMT solver checks the formula satisfiability. From line 7 to 10, **ESBMC** checks if $T$ is satisfiable; if so, it converts the result into a counterexample. If the result is unsatisfiable, then it returns verification successfully. We focus on the part when **ESBMC** finds a violation of the properties and generate a counterexample. **EBF** extracts the assumptions values from the counterexamples and saves them for the fuzzer.

**Instrumentation phase.** In this phase, we developed a custom LLVM Pass using LLVM version 10, which is used to perform the transformations and optimizations in the program. In **EBF**, we developed a custom LLVM pass to instrument the PUT. Specifically, it instruments the PUT by injecting a random delay function after each instruction at the LLVM intermediate representation.
Algorithm 1 BMC

Input: Program Under Test (PUT) and properties (P) to be checked.
Output: FAIL (SMT) with counterexample (Property violations detected by ES-BMC).

1: \( \text{PUT} \leftarrow \text{Specify source code} \)
2: \( P \leftarrow \text{Specify properties} \)
3: \( C \leftarrow \text{Convert (PUT) to control flow} \)
4: \( S \leftarrow \text{Convert (C) into (SSA) form} \)
5: \( E \leftarrow \text{Convert (S) into quantifier free formula (C } \land \lnot P \text{)} \)
6: \( T \leftarrow \text{Solve(E) with SMT solver} \)
7: if \( T \) satisfiable then
8: Convert(T) into counterexample
9: else
10: return Verification Successful;

( LLVM-IR) level. We implement the functions (e.g., delay, Pthread_add, and Pthread_release) as a run-time library in C and link it with the program at compile time. It keeps tracking the active threads by counting (Pthread_create) and switches to the delay function to be executed. However, note that if it encounters (Pthread_join), no active threads are running, then switch to no delay. This approach keeps the instrumentation lightweight and helps the fuzzer detect any unsynchronization between threads. Therefore, with the appropriate inputs, our fuzzer can detect vulnerabilities in concurrency programs.

Algorithm 2 illustrates the steps behind the LLVM pass. From line 1 to 3, it iterates over each Function’s Basic Blocks, then after each instruction \( I \) inside Basic Block \( BB \), a call to the delay function \( D_f \) is inserted. From line 5-9, the pass iterates over each Function’s Basic Blocks; then, if it encounters a call to (Pthread_create), it inserts a call to a function (Pthread_add), whose purpose is to count the active threads and then switch delay function to continue to do the delay. In line 10-12, if it encounters a call to (Pthread_join), it inserts a call to a function (Pthread_release), which reduces the number of active threads by one and switches the delay function to return without delay.

Fuzzing phase. Algorithm 3 shows the standard workflow of a grey-box fuzzer such as AFL [3]. It takes a target program PUT and initial seeds \( M \). It uses its instrumentation to track code coverage \( P_f \) and then starts the loop in line 1; from line 3-5, it selects the seeds and schedules the seed by applying how many mutations \( N \) are applied to \( t \) to generate the mutated seed \( t' \). From line 6, the fuzzer repeatedly executes \( N \) times, for each new seed \( t' \), to get the execution statistics. In line 9, the inputs from \( t' \) are evaluated based on the statistics and coverage feedback from the instrumentation \( p_f \). If the input triggers a crash, it saves it in the crash directory and marks it as a unique crash or if it covers a new branch, it saves it in the seed queue. We build our fuzzing phase on top of AFL and TSAN. EBF feeds AFL with the inputs generated from Input Generation phase with the instrumented code generated from Instrumentation
Algorithm 2 LLVM Custom Pass

Input: Program Under Test (PUT).
Output: Instrumented program.

INITIALIZE FUNCTIONS: (Delay function ($D_f$), pthread_add function ($A_f$) and pthread_release function ($R_f$)).

INITIALIZE TARGET: (pthread_create and pthread_join).

1: for all Function $F \in PUT$ do
2: for Basic blocks $BB$ in $F$ do
3: if $I$ is a branch Instruction then
4: $M \leftarrow$ insert $D_f$ ◁ insert a call to delay function after each instruction,
5: for all Function $F \in PUT$ do
6: for Basic blocks $BB$ in $F$ do
7: if $I =$ pthread_create then ◁ Track each pthread_create
8: $C \leftarrow$ insert $A_f$ ◁ Count as an active thread
9: switch $D_f$ to delay ◁ Continue to run delay
10: else if $I =$ pthread_join then ◁ Track each pthread_join
11: $J \leftarrow R_f$ ◁ Count as NOT an active thread
12: switch $D_f$ return ◁ No delay’s running

phase: with the help of TSAN. AFL then feeds the code with mutation inputs to execute different paths. In this phase, we aim to detect and report the memory corruption errors in concurrent programs such as buffer overflow and memory leak using AFL and detecting concurrency bugs such as data race and thread leak using TSAN. Grey-box Fuzzing approach as AFL is illustrated in Algorithm 3. However, AFL suffers from understanding all possible schedule interleavings [21].

To overcome this limitation, we inject random delays in Instrumentation phase to help AFL detect different thread interleavings. This way, it has more possibilities to detect concurrency bugs, as demonstrated in our experimental evaluation. In the end, EBF analyzes the results and generates the bug report with all the bugs detected, which are either memory corruption bugs or concurrency bugs.

4 Experimental Evaluation

4.1 Description of the benchmarks and setup

We build the EBF tool using python and C++ programming languages. We evaluate it over various benchmarks, specifically the open-source implementation wolfMQTT [7] and SV-COMP subcategories: Pthread, Pthread-atomic, Pthread-divine, Pthread-complex, and Pthread-lit from the concurrency safety category [34], which include 81 verification tasks. We run EBF on each of these verification tasks and compare its results to ESBMC v6.4.0, Lazy-CSeq v2.1, AFL v2.5b, and TSAN clang version 10.0.0.

AFL is a grey-box fuzzer that uses evolutionary genetic algorithms and runtime instrumentation to discover new interesting inputs that trigger new internal states in the targeted binary [37]. ESBMC is context-bounded model...
Algorithm 3 Grey-box Fuzzing

**Input:** Program Under Test (PUT) and CORPUS directory that contains the test cases (M).

**Output:** final seed queue ($Q_S$), vulnerable inputs file ($S_I$).

1: $P_f$ ← instrument (PUT) \hspace{1cm} \triangleright \text{Fuzzer instrument the source code}
2: $S_I$ ← $\phi$ \hspace{1cm} \triangleright \text{Seed selection}
3: while true do
4: \hspace{1cm} $t$ ← select next seed ($Q_S$) \hspace{1cm} \triangleright \text{Seed selection}
5: \hspace{1cm} $N$ ← get mutation chance ($P_f$, $t$) \hspace{1cm} \triangleright \text{Seed scheduling}
6: \hspace{1cm} for all $i \in 1 \cdots N$ do
7: \hspace{1cm} \hspace{1cm} $t'$ ← mutated input ($t$) \hspace{1cm} \triangleright \text{Seed mutation}
8: \hspace{1cm} \hspace{1cm} $rep$ ← Run ($P_f$, $t'$, $M_c$) \hspace{1cm} \triangleright \text{Repeated execution}
9: \hspace{1cm} \hspace{1cm} if is crash($rep$) then
10: \hspace{1cm} \hspace{1cm} \hspace{1cm} $S_I$ ← $S_I$ $\cup$ $t'$ \hspace{1cm} \triangleright \text{Vulnerable seed}
11: \hspace{1cm} \hspace{1cm} else if cover new trace ($t'$, $rep$) then
12: \hspace{1cm} \hspace{1cm} \hspace{1cm} $Q_i$ ← $Q_i$ $\oplus$ $t'$ \hspace{1cm} \triangleright \text{Maintain “effective” seeds}

checker based on SMT solvers to verify of single and multi-threaded C/C++ programs [23]. TSAN is a data race detector for C/C++ programs [4]. It employs compile-time instrumentation to examine all non-race-free memory access at runtime. Lazy-CSeq is a context-bounded verification tool that translates a multi-threaded C program into a sequential consistent C program [25].

All experiments were conducted on an idle Intel Core i7 2.7Ghz processor, with 8 GB of RAM and running Ubuntu 18.04.5 LTS. Our experiments are based on a set of publicly available benchmarks. All tools, benchmarks and results of our evaluation are available on GitHub.

4.2 Goals

Our main experimental goal is to check the EBF method’s performance and effectiveness to verify cryptographic protocols, mostly concurrent implementations of such protocols. Our experimental evaluation has the following goals:

**EG1 Bug detection:** To demonstrate that EBF can detect more bugs in multi-threaded programs than other state-of-the-art verifiers.

**EG2 Cryptographic protocol:** To demonstrate that EBF can be employed to detect bugs in real-world cryptographic protocols.

4.3 Results

The SV-COMP benchmarks provide a wide range of source code to verify, including some TLS examples. Thus, we have achieved concurrent and cryptographic
programs. Note that we set \texttt{unreach\_call} property for all the verification tasks since the concurrency safety category required this specification. We compare the results with \texttt{ESBMC}, \texttt{AFL}, \texttt{TSAN} and \texttt{Lazy-\texttt{CSeq}} in isolation. In this evaluation, we check how many bugs were detected with \texttt{EBF} compared with the other tools. Table 1 illustrates the five subcategories (e.g., \texttt{Pthread}, \texttt{Pthread-atomics}, \texttt{Pthread-divine}, \texttt{Pthread-complex}, and \texttt{Pthread-lit}) with the number of verification tasks and lines of code (LOC). Bold numbers indicate the best results for each set of benchmarks. It also shows the tools evaluated and compared with \texttt{EBF}; note that \texttt{Lazy-\texttt{CSeq}} is the SV-COMP 2021 winner.

Overall, \texttt{EBF} consistently outperforms all evaluated tools. For the \texttt{Pthread} subcategory that consists of 38 tasks, we significantly outperform \texttt{ESBMC}, \texttt{AFL}, \texttt{Lazy-\texttt{CSeq}}, and \texttt{TSAN} by two verification tasks. \texttt{EBF} also outperforms all evaluated tools in the \texttt{Pthread-divine} and \texttt{Pthread-lit} subcategories, and \texttt{TSAN} by one verification task in each subcategory. However, in the \texttt{Pthread-atomic} and \texttt{Pthread-complex} subcategories, \texttt{EBF} solves the same number of tasks as \texttt{TSAN}, but it outperforms all other evaluated tools.

Note that none of the evaluated tools is consistently better than the other. Each verifier has advantages and disadvantages, where \texttt{EBF} scalability depends mainly on its capability of detecting both memory corruption and concurrency bugs, which maximize the chance of catching more bugs. \texttt{ESBMC} can find both types of vulnerabilities except that it does not perform well in concurrency implementations, as shown in Table 1. \texttt{ESBMC} either detects a violation or exhausts time or memory limits, while \texttt{Lazy-\texttt{CSeq}} performed well except in \texttt{Pthread-lit} subcategory. Also, since \texttt{AFL} is not effectively detecting concurrency vulnerabilities, it performed worse than all other tools in most subcategories. Lastly, \texttt{TSAN} performs relatively well in most subcategories compared to the other tools.

Overall, \texttt{EBF} detected bugs in 51 out of 81, with four verification tasks more in total than \texttt{TSAN} (the best competing tool). \texttt{EBF} shows encouraging results compared with the state-of-the-art verification tools, which answers \texttt{EG1}.

To verify the WolfMQTT implementation, we used \texttt{EBF} over its API functions (e.g., \texttt{MqttSocket\_Write}, \texttt{MqttSocket\_Read}) by verifying the file \texttt{multi\_thread.c}, which contains a set of tests for WolfMQTT in multi-threads contexts. \texttt{EBF} could detect a data race in the “\texttt{Mqtt\_Client\_Wait\_Type}” function. We execute this experiment with a mosquito server \cite{mosquito} running on the same machine as the client. We created a GitHub issue for the developers to confirm it.\footnote{https://github.com/wolfSSL/wolfMQTT/issues/198}

We also run \texttt{EBF} over an example of WolfSSL. \texttt{EBF} detected a thread leak in \texttt{memory-tls.c}, which was reported in GitHub as an issue to the WolfSSL developers.\footnote{https://github.com/wolfSSL/wolfssl-examples/issues/242}


Table 1. Experimental Results.

| Concurrency Safety | Tasks | LOC  | ESBMC | AFL | TSAN | Lazy-CSeq | EBF |
|--------------------|-------|------|-------|-----|------|-----------|-----|
| Pthread            | 38    | 3674 | 8     | 3   | 17   | 15        | 19  |
| Pthread-atomic     | 11    | 1669 | 0     | 2   | 6    | 2         | 6   |
| Pthread-divine     | 16    | 333  | 2     | 0   | 13   | 7         | 14  |
| Pthread-complex    | 5     | 1633 | 0     | 0   | 3    | 2         | 3   |
| Pthread-lit        | 11    | 595  | 1     | 1   | 8    | 3         | 9   |
| Total              | 81    | 7904 | 11    | 6   | 47   | 29        | 51  |

EBF could detect a data race in one open-source implementation for cryptographic protocols, which thus answers EG2.

4.4 Threats to Validity

We selected our SV-COMP benchmarks based on two factors. First, it contains a variety of vulnerable programs. Second, it employs POSIX threads. However, SV-COMP benchmarks contain specific functions, which are not following the C standard. For example, we excluded some subcategories such as “Pthread-driver-races” because some of the benchmarks contain compilation errors using clang 10 (e.g., unknown type name), which we might introduce bugs if we try to fix them. As a result, we evaluated EBF on 81 verification tasks from the concurrency safety category. The main threat is that this paper’s evaluation is subject to these benchmarks and may not be generalized to other benchmarks. Another threat to our experiment’s validity is that we only evaluated our method on one example of the WolfMQTT implementation. Still, that example calls for several functions and also one instance of the WolfSSL library. Therefore, our approach may not generalize on open-source cryptographic protocol implementations, except for the WolfMQTT and WolfSSL. In general, EBF can be further developed to be effective on all the implementation differences of cryptographic protocols.

5 Conclusions and Future Work

Inadequate implementations of cryptographic protocols can cause memory corruption vulnerabilities such as memory leaks and buffer overflows. Furthermore, it is not an easy task for the developers to discover concurrency vulnerabilities [21]; they are generally produced by shared memory access without proper synchronization between threads. This paper presented EBF, a novel software verification tool that combines BMC and fuzzing techniques to detect memory corruption and concurrency vulnerabilities in IoT cryptographic protocols and concurrency implementations. Our tool works by feeding AFL with an initial seed

https://github.com/sosy-lab/sv-benchmarks/issues/1291
generated from ESBMC counterexamples along with the instrumented program generated from our custom-developed LLVM pass. We run the tool over several SV-COMP benchmarks and an open-source application. We show that EBF outperforms other state-of-the-art software testing and verification tools such as ESBMC, AFL, LazyCSeq and TSAN in detecting vulnerabilities in more tasks. Thus, EBF contributes to the vision of fully verified trustworthy software systems. For future work, EBF will be extended to support more cryptographic protocol implementations and to improve EBF performance, including speed up the verification time.

References

1. https://www.wolfssl.com/ (2019)
2. https://github.com/esbmc/esbmc (2019)
3. https://github.com/google/AFL (2019)
4. https://clang.llvm.org/docs/ThreadSanitizer.html (2019)
5. https://llvm.org/docs/LibFuzzer.html (2020)
6. https://llvm.org/docs/WritingAnLLVMPass.html (2020)
7. https://github.com/wolfSSL/wolfMQTT (2020)
8. https://mosquitto.org/ (2020)
9. Collection of verification tasks (2020), https://github.com/sosy-lab/sv-benchmarks/specifications
10. Acharya, K., Saywan, M., Bhargava, S.: Analysis of cryptographic algorithms for network security. International Journal of Computer Applications Technology and Research 3, 130–135 (2013)
11. Aitel, D.: An introduction to spike, the fuzzer creation kit. presentation slides), Aug 1 (2002)
12. Aizatulin, M., Gordon, A.D., Jürjens, J.: Extracting and verifying cryptographic models from C protocol code by symbolic execution. CoRR abs/1107.1017 (2011)
13. Andrea, I., Chrysostomou, C., Hadjichristofi, G.C.: Internet of things: Security vulnerabilities and challenges. (ISCC) pp. 180–187 (2015)
14. Aumasson, J.P., Romainnier, Y.: Automated testing of crypto software using differential fuzzing. Black Hat USA 7 (2017)
15. Balte, A., Kashid, A., Patil, B.: Security issues in internet of things (iot): A survey. International Journal of Advanced Research in Computer Science and Software Engineering 5(4), 450–455 (2015)
16. Beyer, D., Lemberger, T.: Software verification: Testing vs. model checking. In: Hardware and Software: Verification and Testing. pp. 99–114. Springer (2017)
17. Biere, A.: Bounded model checking. In: Handbook of Satisfiability, pp. 457–481 (2009)
18. Biyani, A., Sharma, G., Aghav, J., Waradpande, P., Savaji, P., Gautam, M.: Extension of spike for encrypted protocol fuzzing. In: 2011 Third International Conference on Multimedia Information Networking and Security. pp. 343–347. IEEE (Nov 2011)
19. Cadar, C., Dunbar, D., Engler, D.R., et al.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. vol. 8, pp. 209–224 (2008)
20. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM 56(2), 82–90 (2013)
21. Chen, H., Guo, S., Xue, Y., Sui, Y., Zhang, C., Li, Y., Wang, H., Liu, Y.: {MUZZ}: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In: 29th {USENIX} Security Symposium ({USENIX} Security 20). pp. 2325–2342 (2020)
22. Datta, P., Sharma, B.: A survey on iot architectures, protocols, security and smart city based applications. In: (ICCCNT), pp. 1–5. IEEE, Delhi, India (July 2017)
23. Gadelha, M.R., Menezes, R.S., Cordeiro, L.C.: Esbmc 6.1: automated test case generation using bounded model checking. International Journal on Software Tools for Technology Transfer pp. 1–5 (2020)
24. Given-Wilson, T., Jafri, N., Lanet, J.L., Legay, A.: An automated formal process for detecting fault injection vulnerabilities in binaries and case study on present. In: 2017 IEEE Trustcom/BigDataSE/ICESS, pp. 293–300. IEEE (2017)
25. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Lazy-cseq: a lazy sequentialization tool for c. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 398–401. Springer (2014)
26. Kroening, D., Tautschnig, M.: Chmc–c bounded model checker. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 389–391. Springer (2014)
27. Li, Y., Ji, S., Lv, C., Chen, Y., Chen, J., Gu, Q., Wu, C.: V-fuzz: Vulnerability-oriented evolutionary fuzzing. CoRR (2019)
28. Liu, C., Zou, D., Luo, P., Zhu, B.B., Jin, H.: A heuristic framework to detect concurrency vulnerabilities. In: Proceedings of the 34th Annual Computer Security Applications Conference. pp. 529–541 (2018)
29. Mehavarunan: The future of iot: 4 predictions about the internet of things (2019), https://thriveglobal.com/stories/the-future-of-iot-4-predictions-about-the-internet-of-things/
30. Ognawala, S., Hutzelmann, T., Psallida, E., Pretschner, A.: Improving function coverage with munch: A hybrid fuzzing and directed symbolic execution approach. In: SAC. pp. 1475–1482. ACM (2018)
31. Philip, M., Vaithiyathan: A survey on lightweight ciphers for iot devices. In: 2017 (TAP Energy). pp. 1–4. IEEE (Dec 2017)
32. Razzaq, M.A., Gill, S.H., Qureshi, M.A., Ullah, S.: Security issues in the internet of things (iot): A comprehensive study. International Journal of Advanced Computer Science and Applications 8(6), 383 (2017)
33. Singh, S., Sharma, P.K., Moon, S.Y., Park, J.H.: Advanced lightweight encryption algorithms for iot devices: survey, challenges and solutions. Journal of Ambient Intelligence and Humanized Computing pp. 1–18 (May 2017)
34. 10th Competition on Software Verification (SV-COMP 2021): https://sv-comp.sosy-lab.org/2021/ (2021)
35. Tsankov, P., Dashki, M.T., Basin, D.: Seefuzz: Fuzz-testing security protocols. In: 2012 (AST). pp. 1–7. IEEE (June 2012)
36. Vanhoef, M., Piessens, F.: Symbolic execution of security protocol implementations: Handling cryptographic primitives. In: 12th (WOOT). USENIX, Baltimore, MD (2018)
37. You, W., Liu, X., Ma, S., Perry, D., Zhang, X., Liang, B.: Slf: fuzzing without valid seed inputs. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 712–723. IEEE (2019)