Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

# Automatic discovery of stateful variables in network protocol software based on replay analysis*#

Jianxin HUANG[1], Bo YU‡[1], Runhao LIU[1], Jinshu SU‡[1,2]

[1]*College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China*
[2]*Academy of Military Science, Beijing 100091, China*

E-mail: jxin8585@nudt.edu.cn; yubo0615@nudt.edu.cn; runhaoliu@nudt.edu.cn; sjs@nudt.edu.cn

Received June 25, 2022; Revision accepted Nov. 9, 2022; Crosschecked Feb. 2, 2023

**Abstract:** Network protocol software is usually characterized by complicated functions and a vast state space. In this type of program, a massive number of stateful variables that are used to represent the evolution of the states and store some information about the sessions are prone to potential flaws caused by violations of protocol specification requirements and program logic. Discovering such variables is significant in discovering and exploiting vulnerabilities in protocol software, and still needs massive manual verifications. In this paper, we propose a novel method that could automatically discover the use of stateful variables in network protocol software. The core idea is that a stateful variable features information of the communication entities and the software states, so it will exist in the form of a global or static variable during program execution. Based on recording and replaying a protocol program's execution, varieties of variables in the life cycle can be tracked with the technique of dynamic instrument. We draw up some rules from multiple dimensions by taking full advantage of the existing vulnerability knowledge to determine whether the data stored in critical memory areas have stateful characteristics. We also implement a prototype system that can discover stateful variables automatically and then perform it on nine programs in ProFuzzBench and two complex real-world software programs. With the help of available open-source code, the evaluation results show that the average true positive rate (TPR) can reach 82% and the average precision can be approximately up to 96%.

**Key words:** Stateful variables; Network protocol software; Program analysis technology; Network security
https://doi.org/10.1631/FITEE.2200275 **CLC number:** TP311

## 1 Introduction

Nowadays, various network protocols exist on the Internet. A protocol can be viewed as a collection of communication rules, and it usually gets obvious stateful states by running network functions and providing online services. The state of a protocol can be typically measured by a state machine (Lee et al., 2018; Pham et al., 2020; Natella, 2022). Due to a lot of protocol implementations, i.e., network protocol software, a program may generate a vast number of states that are quite different from the states of the protocol in its running process. Because of this, variables and data structures used to hold the information related to the state in the program also have a degree of stateful characteristics. Thus, we call this kind of variable a "stateful variable." For example, OpenSSL, an open-source implementation of the secure sockets layer (SSL) protocol, is one of the most versatile SSL tools in the real world. It uses struct ssl_ctx_st to describe and store some session contexts and struct cert_st to hold the server

‡ Corresponding authors
* Project supported by the National Natural Science Foundation of China (Nos. 61902416 and 61902412) and the Natural Science Foundation of Hunan Province, China (No. 2019JJ50729)
# Electronic supplementary materials: The online version of this article (https://doi.org/10.1631/FITEE.2200275) contains supplementary materials, which are available to authorized users
ORCID: Jianxin HUANG, https://orcid.org/0000-0002-8643-7355; Bo YU, https://orcid.org/0000-0001-6576-5555; Jinshu SU, https://orcid.org/0000-0001-9273-616X

certificate used; these are single variables that would be stateful. Moreover, struct session_id_context is used to ensure that sessions are reused only in the appropriate context, and then an unsigned integer variable sid_ctx_length is used to specify the length of the struct session_id_context. There is an obvious correlation of the length constraint between the two variables that belong to the correlated stateful variables. We can find many similar cases in this software.

Compared with ordinary variables, stateful variables often present some special features during the execution of the program. They are closely related to the normal operation of protocol services, the correct processing of protocol data, and the safe execution of the protocol software. However, the ambiguities of specifications, the software deviations in implementations, and the coding errors or improper uses of stateful variables by program developers will lead to an abnormal state, which will leave vulnerable points in protocol software. Logical errors that result from variables being inappropriately used in protocol software might be triggered in a particular scenario involving interoperation between clients and the sever, causing unexpected consequences.

In view of this question, previous work in related areas is largely insufficient at present. Most methods for describing and discovering logical errors are concerned with variables in common binary, mainly based on static analysis (Ye et al., 2014; Garmany et al., 2019). Due to a lack of effective analysis methods for detecting the transitions of protocol states and the conversions of multiple sessions, such techniques could be relatively limited when they are applied to the network protocol software for debugging and testing. Meanwhile, traditional vulnerability mining technology, such as fuzzing, focuses mainly on analysis by coverage-guided tracing and exploring. The obvious drawback of this approach is that it may take too much time to generate such inputs to explore a specific state for testing protocol software, since its ability is so constrained by the session states that would be difficult to locate and identify quickly. Further more, as a consequence of some management and control functions provided by network protocol software, vulnerabilities possibly appear in many ways or forms, not only a crash, but also privacy disclosure, authentication bypass, RCE, etc. (Song et al., 2019; Yu et al., 2019). Under such circumstances, common methods for error detection and assessment used in testing binary files may not be as effective as before. The technological challenges can be summarized in two points. First, there is low automaticity. Stateful variables need to be differentiated by analyzing a series of network protocol software behaviors. This requires extensive manual work in testing and verifying a vulnerability caused by misused stateful variables and lack of automatic approaches. Second, they are hard to discover. Stateful variables are closely related to the protocol software states and network inputs. To observe the characteristics of stateful variables, it is necessary to use a specific sequence of data packages to reach the state that can cause errors. This conduct makes it difficult to cover deeper states, so the variables mistaken in deeper paths cannot be found eventually.

In this paper, we address this challenge and propose an automatic approach that infers and discovers stateful variables with a rich amount of information about the evolution of the protocol software states and the life cycle of various variables. The key idea is to establish certain principles for inspection according to the characteristics of the stateful variables in timing, spacing, and operating sequences, using a dynamic analysis technique based on recording and replaying the running traces of protocol software as the input to achieve our purpose.

In summary, our main contributions are as follows:

1. An automatic technique that is novel and can work on binaries of different protocol software programs is proposed to discover stateful variables.

2. A stateful variable discovery algorithm is proposed from three dimensions, about timing, spacing, and operating sequence. It is suitable for a deeper analysis of complicated states in protocol software by recording and replaying the executed trace.

3. The prototype system implemented by our approach to validate our idea is applied to 11 protocol software programs, with an average true positive rate (TPR) of 82% and an average precision up to 96%.

## 2 Overview

### 2.1 Stateful network protocols

During the course of running functions and providing services, network protocols continuously receive and send data packets that contain a certain degree of logically progressive or causal relationship through several rounds. Concurrently, the statuses of multiple client interactions of clients need to be maintained and stored by a server. Such protocols are known as stateful network protocols.

A state of a network protocol represents the current stage consisting of the numbers of operations, descriptions, and information in the protocol's state machine. The state of the network protocol software reflects this program's running state, which is not only affected by input from the network, but also related to client sessions and configurations from administrators in the network.

### 2.2 Recording and replaying technology

A virtualized platform is usually used in dynamic program analysis to record and replay its execution process, so the execution environment can be saved and efficiency analysis can be improved. Some crucial data, such as the value of registers, memory block, and CPU flags, are captured in a fixed manner without affecting the execution flow. PANDA, built upon QEMU, is an open-source platform for architecture-neutral dynamic analysis (Dolan-Gavitt et al., 2015). It adds a feature to record and replay executions, enabling iterative, deep, whole system analyses. Further, the replay log files are compact and shareable, allowing for repeatable experiments. Our system leverages the primary abilities offered by PANDA to make a record of protocol software execution, and we then conduct our analyses by replaying the execution trace.

### 2.3 Real-world examples

Now we will briefly introduce a motivating example, CVE-2015-0291, a NULL pointer dereference vulnerability of OpenSSL (www.openssl.org), which typically deduces the root cause. Through this example, we will show the limitations of existing tools.

The SSL protocol, which was designed to establish a secure network connection between a client and a server, ensures that all the data passed between the client and the server are private. Before a session is established, some controlling parameters and security policies need to be negotiated by a series of handshake packets. A standard SSL handshake process is generally as follows: (1) ClientHello, (2) ServerHello, (3) authentication and pre-master secret, (4) decryption and master secret, (5) generating session keys, and (6) encryption with the session key.

As we know, a NULL pointer dereference flaw with high severity has been found when OpenSSL handles a renegotiation request that contains incorrect or maliciously crafted data; it affects OpenSSL version 1.0.2, and has been fixed in version 1.0.2a. Remote attackers could use this flaw to crash an OpenSSL server by sending malicious ClientHello packages. By analyzing the source code of OpenSSL 1.0.2, we found that vulnerability exists in two functions, tls1_set_server_sigalgs() and tls1_set_shared_sigalgs(), within the ssl\t1_lib.c file. When a client connects to an OpenSSL 1.0.2 server in a conventional way, ClientHello is resolved followed by protocol specification. A data struct (SSL *s->cert) that stores the signature algorithm to be negotiated is passed to tls1_set_server_sigalgs(), where the variable shared_sigalgs will be assigned NULL first, and then passed to tls1_set_shared_sigalgs(), where the variable shared_sigalgs will then be assigned to point to the struct that contains the identification of the signature algorithm in ClientHello. A co-related variable named shared_sigalgs_len that specifies the length of the struct that shared_sigalgs points to will be set to a specified value. In this condition, if the client resends a ClientHello and renegotiates with an invalid signature algorithm extension, shared_sigalgs() will be clear again in tls1_set_server_sigalgs. However, shared_sigalgs_len would not be set to 0 accordingly. So, a NULL pointer dereference will occur when the execution path goes into a loop with shared_sigalgs_len as the judgment condition in tls1_set_shared_sigalgs(). This can be exploited in a denial-of-service (DoS) attack against the server.

In this case, shared_sigalgs and shared_sigalgs_len are co-related stateful variables. The vulnerability is caused by improper handling of this group of stateful variables, which are assigned and used at the same time but not cleared simultaneously, resulting in inconsistent states between them.

To achieve these states, we need to construct specific packages or large numbers of prefix messages that can reach the corresponding state according to the behavior of the protocol software. Because such errors are triggered by a more complicated path and a more in-depth state than a regular binary file, this kind of vulnerability, which occurs in protocol software, can hardly be disclosed by the state-of-the-art static tools.

## 3 Design

### 3.1 Definitions

#### 3.1.1 Stateful network protocols

In view of the features of stateful protocols, using the finite state machine (FSM) theory to describe the protocols has been proved to have excellent performance and outstanding scalability. An FSM abstracted by a network protocol represents a mathematical model of a set of states, the events that can trigger the state machine, the actions to be executed when certain events occur in a specific format in received packages, and the transitions from the current state to a successive one.

**Definition 1** Let $M$ be an FSM of a stateful network protocol that consists of $n$ states and $m$ transitions ($n$, $m$>0). $M$ is a quintuple of the following form:

$$M = (Q, \xi, \delta, q_0, F), \tag{1}$$

where $Q$ is a non-empty finite set of states. $Q = \bigcup_{i=0}^{n} q_i$. $\forall q \in Q$, $q$ is a state of $M$. $\xi$ is a set of input messages from network. The elements of $\xi$ follow a finite sequence $\langle \xi_1, \xi_2, ..., \xi_m \rangle$. $\delta: Q \times \xi \to Q$ is a state transition function. $q_0 \in Q$ is the initial state of $M$. $F$ is a non-empty finite set of final states. $F \subseteq Q$, $\forall q \in F$, where $q$ is a final state of $M$.

#### 3.1.2 State of variables

**Definition 2** Let $W$ be a stateful protocol software program, and let $\mathrm{var}_1$, $\mathrm{var}_2$, ..., $\mathrm{var}_n$ ($n$>0) be the variables of $W$. Then $V$ is the set of these variables. $V$={$\mathrm{var}_1$, $\mathrm{var}_2$, ..., $\mathrm{var}_n$}, and the set $V|_{t_i}$={$\mathrm{var}_1|_{t_i}$, $\mathrm{var}_2|_{t_i}$, ..., $\mathrm{var}_n|_{t_i}$} is the combination of variable values at one point of a timeline, where $t_i$ is any time in the execution process of $W$.

**Definition 3** Let $W$ be a stateful protocol software program, and let $\mathrm{var}_1$, $\mathrm{var}_2$, ..., $\mathrm{var}_n$ ($n$>0) be the variables of $W$.

1. If an initialized variable $\mathrm{var}_x$ is written at $t_i$, then it is used (accessed or modified) at $t_j$ by the condition of $\mathrm{var}_x|_{t_i}$; we say the value of $\mathrm{var}_x|_{t_j}$ is related to $\mathrm{var}_x|_{t_i}$ ($i < j$), and $\mathrm{var}_x$ is a stateful variable. So, $\mathrm{var}_x|_{t_i}$ logically implies that $\mathrm{var}_x|_{t_j}$ can be marked as $\mathrm{var}_x|_{t_i} \to \mathrm{var}_x|_{t_j}$, or for brevity, just $(\mathrm{var}_x)_{\mathrm{state}}$.

2. If the value of $\mathrm{var}_y$ is affected by the value of $\mathrm{var}_x$ (e.g., $\mathrm{var}_y$ records the current length of $\mathrm{var}_x$), or the writing operation for $\mathrm{var}_y$ is conditional on the value of $\mathrm{var}_x$, when $\mathrm{var}_x$ changes, $\mathrm{var}_y$ will change accordingly; we say $\mathrm{var}_y$ is associated with $\mathrm{var}_x$ and use tuple ($\mathrm{var}_x$,$\mathrm{var}_y$) for representation. If the value of $\mathrm{var}_y|_{t_i}$ is associated with $\mathrm{var}_x|_{t_i}$, and $\mathrm{var}_x|_{t_j}$, $\mathrm{var}_y|_{t_j}$ are related to $\mathrm{var}_x|_{t_i}$, $\mathrm{var}_y|_{t_i}(i < j)$, $\mathrm{var}_x$ and $\mathrm{var}_y$ are correlated stateful variables and we write $(\mathrm{var}_x|_{t_i}$,$\mathrm{var}_y|_{t_i}) \to (\mathrm{var}_x|_{t_j}$,$\mathrm{var}_y|_{t_j})$, or for brevity, just $(\mathrm{var}_x$,$\mathrm{var}_y)_{\mathrm{state}}$.

#### 3.1.3 State of protocol software

The state of a protocol software program is jointly determined by the runtime values of all internal variables and registers during the execution of the program. In the protocol software state machine, an input (or package) will cause the software to perform a series of transitions from the previous state.

**Definition 4** Let $W$ be a stateful protocol software program. Let $V$ be the set of variables and Reg be the set of registers that $W$ uses. $S = V \cup \mathrm{Reg}$, and the set $S|_{t_i} = V|_{t_i} \cup \mathrm{Reg}|_{t_i}$ is the combination of variables and register values at one point on the timeline, where $t_i$ is any time during the execution process of $W$.

#### 3.1.4 State space of protocol software

Due to the progress of control flow, the values of the program's internal variables and registers also change along with external inputs. The state space of a protocol software program is all possible states of variables, and is a Cartesian product formed by the states of all variables within the software.

**Definition 5** Let $W$ be a stateful protocol software program, and let $S$ be a state of $W$ at one point on the timeline. Then $S^*$ is the state space of $W$. $S^* = S|_{t_1} \times S|_{t_2} \times ... \times S|_{t_n} (n > 0)$, where $t_i$ ($1 \le i \le n$) is any time during the execution process of $W$.

### 3.2 Depiction of stateful variables

Note that vulnerabilities caused by stateful variables may not be disclosed sufficiently under limited conditions, especially without source code or with software using proprietary protocols, because stateful variables are not thoroughly searched or some states are not fully explored. However, automatic discovery of such stateful variables in protocol software is a good way to provide candidates for stateful errors. We will depict stateful variables through a range of multi-dimensional analyses in the following.

3.2.1 Stateful variables can reflect the transitions of protocol states

Network protocols stipulate the behaviors of entities participating in communication at a certain time, as well as the responses to specific events or packages. Consequently, protocol states are usually determined by these network events and messages because interactions between communication entities have become the driving force for continual transitions of protocol states. In communication progress of protocol software, contents stored in the variables that represent the protocol state and its transitions are often related to the entity of the current session, and there has been a great certainty of values between the same entities and within the same session. From the life cycle perspective, such variables are usually generated and assigned when the two communicating parties establish a connection after the protocol service starts, and their lives can often be persistent in the runtime. To this end, it is of great significance to determine whether a variable has a stateful characteristic by recording the timestamp when it is assigned a value and tracking its life cycle by means of dynamic analysis.

According to the previous statement, the states of protocol software can be represented by the stateful variables in the program. The changes in variable values probably occur when transitions of the corresponding software states take place, and we introduce a symbol "⇒" to represent such a causal relationship. Therefore, based on Definitions 1 and 3, we develop the following proposition:

**Proposition 1** Let $M$ be an FSM representing a stateful protocol. $Q$ is the state set of $M$. $Q = \bigcup_{i=0}^{n} q_i$ $(n > 0)$, $\forall q \in Q$, $q$ is a state of $M$, and $\delta: Q \times \xi \to Q$ is a state transition function. $W$ is a software program of $M$. $V$ is the set of variables in $W$. If $\delta: q_s \times \xi \to q_t$ $(0 < s < t < n)$ happens at $t_i$, where $t_i$ is any time during the execution process of $W$,

1. $\exists \mathrm{Var} \in V$, $q_s \Rightarrow \mathrm{Var}|_{t_{i-u}}$, and $q_t \Rightarrow \mathrm{Var}|_{t_{i+v}}$ $(1 < u, v < n)$, then $(q_s \to q_t) \Rightarrow \mathrm{Var}_{\mathrm{state}}$, or

2. $\exists \mathrm{Var}_1, \mathrm{Var}_2 \in V$, $q_s \Rightarrow (\mathrm{Var}_1|_{t_{i-u}} \to \mathrm{Var}_2|_{t_{i-u}})$, and $q_t \Rightarrow (\mathrm{Var}|_{t_{i+v}} \to \mathrm{Var}_2|_{t_{i+v}})$ $(1 < u, v < n)$, then $(q_s \to q_t) \Rightarrow (\mathrm{Var}_1, \mathrm{Var}_2)_{\mathrm{state}}$.

3.2.2 Stateful variables can reflect the transitions of software states

Protocol software programs are implementations of network protocols, and the states of protocol software are closely associated with the protocol states. The states of software usually have something to do with the numerical values or data structures as global variables or static variables inside. When a binary is loaded into memory, static variables are usually mapped to the data segment (for initialized static variables) or block started by symbol (BSS, for uninitialized static variables), and global variables are dynamically allocated and reclaimed in the heap area by the operating system (OS). For this reason, it is helpful to judge whether a variable has a stateful characteristic by tracking operations of the core areas such as heap, BSS, and data segment in the memory address space of the program using dynamic analysis.

**Proposition 2** Let $W$ be a software program of stateful protocol $M$. $S^*$ is the state space of $W$. If $\exists S|_{t_a}, S|_{t_b} \in S^*$, the transition from $S|_{t_a}$ to $S|_{t_b}$ $S|_{t_a} \to S|_{t_b}$ happens at $t_i$, where $t_a, t_i, t_b$ are time points in the execution process of $W$,

1. $\exists \mathrm{Var} \in V$, $\mathrm{Var}_{\mathrm{addr}} \in (\mathrm{Mem}_{\mathrm{heap}_{\mathrm{addr}}} \cup \mathrm{Mem}_{\mathrm{bss}_{\mathrm{addr}}} \cup \mathrm{Mem}_{\mathrm{data}_{\mathrm{addr}}})$, $(S|_{t_a} \to S|_{t_b}) \Rightarrow (\mathrm{Var}|_{t_a} \to \mathrm{Var}|_{t_b})$, then $(S|_{t_a} \to S|_{t_b}) \Rightarrow \mathrm{Var}_{\mathrm{state}}$, or

2. $\exists \mathrm{Var}_1, \mathrm{Var}_2 \in V$, $\mathrm{Var}_{1_{\mathrm{addr}}}, \mathrm{Var}_{2_{\mathrm{addr}}} \in (\mathrm{Mem}_{\mathrm{heap}_{\mathrm{addr}}} \cup \mathrm{Mem}_{\mathrm{bss}_{\mathrm{addr}}} \cup \mathrm{Mem}_{\mathrm{data}_{\mathrm{addr}}})$, $(S|_{t_a} \to S|_{t_b}) \Rightarrow (\mathrm{Var}_1|_{t_a} \to \mathrm{Var}_2|_{t_a}) \to (\mathrm{Var}_1|_{t_b} \to \mathrm{Var}_2|_{t_b})$, then $(S|_{t_a} \to S|_{t_b}) \Rightarrow (\mathrm{Var}_1, \mathrm{Var}_2)_{\mathrm{state}}$.

3.2.3 Stateful variables can reflect the security and reliability of key data operations

In view of the particularity and importance of the data stored in the stateful variable, operations on reading, modifying, and clearing must comply with

the protocol specifications, conform to the logic of program state transitions, and be restricted by the security mechanism of the OS. From the perspective of the definition-usage relationship of variables, considering one single variable itself, any operation, reading or writing, should not only abide by the system's safe operation sequence to avoid UAF, DF, and other similar vulnerabilities, but also maintain the consistency of the variable with its previous state and avoid violating the particular constraints concerning any other variable associated with it. In this connection, it is feasible to differentiate whether a variable has a stateful characteristic by recording all operations of the memory blocks allocated with global and static variables in the life span, to reverse the definition-usage chain of these variables and check constraints and consistency based on the definition-usage relationship based on dynamic analysis.

**Definition 6** Let $W$ be a stateful protocol software program and $V$ be the set of variables in $W$. Var is a variable that belongs to $V$. If an instruction Ins, located in the basic block BB, is an assignment instruction to Var in $W$, then we say the value of Var is defined at the instruction Ins corresponding to the basic block BB and write def(Var, Ins, BB).

**Definition 7** Let $W$ be a stateful protocol software program and $V$ be the set of variables in $W$. Var is a variable that belongs to $V$. If an instruction Ins, located in the basic block BB, is a reference instruction to Var in $W$, then we say the value of Var is used at the instruction Ins corresponding to the basic block BB and write use(Var, Ins, BB).

**Definition 8** Let $W$ be a stateful protocol software program, $V$ be the set of variables in $W$, and $P$ be the set of execution paths of $W$. Var is a variable that belongs to $V$. If there is a path $p$ from $\mathrm{def}(\mathrm{Var}, \mathrm{Ins}_1, \mathrm{BB}_1)$ to $\mathrm{use}(\mathrm{Var}, \mathrm{Ins}_2, \mathrm{BB}_1)$ or to $\mathrm{use}(\mathrm{Var}, \mathrm{Ins}_2, \mathrm{BB}_2)$ $(\mathrm{Ins}_1 \prec \mathrm{Ins}_2)$ in $P$, where $\mathrm{Ins}_1 \prec \mathrm{Ins}_2$ represents that $\mathrm{Ins}_1$ precedes $\mathrm{Ins}_2$ in $P$, then we say it is a definition-usage path or definition-usage chain about Var in $P$ and write $\mathrm{du}(\mathrm{Var}, \mathrm{Ins}_1, \mathrm{BB}_1, \mathrm{Ins}_2, \mathrm{BB}_1)$ or $\mathrm{du}(\mathrm{Var}, \mathrm{Ins}_1, \mathrm{BB}_1, \mathrm{Ins}_2, \mathrm{BB}_2) \in P$.

**Proposition 3** Let $M$ be an FSM representing a stateful protocol. $W$ is a software program of $M$, $V$ is the set of variables in $W$, and $P$ is the set of execution paths of $W$. Two cases are considered:

1. If $\exists \mathrm{Var} \in V$, $\exists \mathrm{du}(\mathrm{Var}, \mathrm{Ins}_1, \mathrm{BB}_1, \mathrm{Ins}_2, \mathrm{BB}_2) \in P$, $\mathrm{Ins}_1$ is executed at $t_i$, $\mathrm{Ins}_2$ is executed at $t_j$ $(0 < i < j)$, where $t_i, t_j$ are time points in the execution process of $W$, $\mathrm{du}(\mathrm{Var}, \mathrm{Ins}_1, \mathrm{BB}_1, \mathrm{Ins}_2, \mathrm{BB}_2) \Rightarrow (\mathrm{Var}|_{t_i} \rightarrow \mathrm{Var}|_{t_j})$, and $\exists \mathrm{du}(\mathrm{Var}, \mathrm{Ins}_1, \mathrm{BB}_1, \mathrm{Ins}_2, \mathrm{BB}_2, \mathrm{Ins}_3, \mathrm{BB}_3) \in P$, $\mathrm{Ins}_3$ is executed at $t_k$ $(0 < i < j < k)$, $\mathrm{du}(\mathrm{Var}, \mathrm{Ins}_1, \mathrm{BB}_1, \mathrm{Ins}_2, \mathrm{BB}_2, \mathrm{Ins}_3, \mathrm{BB}_3) \Rightarrow (\mathrm{Var}|_{t_j} = \mathrm{Var}|_{t_k})$, then $\mathrm{du}(\mathrm{Var}, \mathrm{Ins}_1, \mathrm{BB}_1, \mathrm{Ins}_2, \mathrm{BB}_2, \mathrm{Ins}_3, \mathrm{BB}_3) \Rightarrow \mathrm{Var}_{\mathrm{state}}$.

2. If $\exists \mathrm{Var}_a, \mathrm{Var}_b \in V$, $\exists \mathrm{du}(\mathrm{Var}_a, \mathrm{Ins}_{a_1}, \mathrm{BB}_1, \mathrm{Ins}_{a_2}, \mathrm{BB}_2)$, $\mathrm{du}(\mathrm{Var}_b, \mathrm{Ins}_{b_1}, \mathrm{BB}_1, \mathrm{Ins}_{b_2}, \mathrm{BB}_2) \in P$, $\mathrm{Ins}_{a_1}$ is executed at $t_i$, $\mathrm{Ins}_{a_2}$ is executed at $t_j$ $(0 < i < j)$, where $t_i, t_j$ are time points in the execution process of $W$, $\mathrm{Ins}_{b_1}$ is executed at $t_i + \Delta t$, $\mathrm{Ins}_{b_2}$ is executed at $t_j + \Delta t$ $(\Delta t > 0)$, $\mathrm{du}(\mathrm{Var}_a, \mathrm{Ins}_{a_1}, \mathrm{BB}_1, \mathrm{Ins}_{a_2}, \mathrm{BB}_2) \Rightarrow (\mathrm{Var}_a|_{t_i} \rightarrow \mathrm{Var}_a|_{t_j})$, $\mathrm{du}(\mathrm{Var}_b, \mathrm{Ins}_{b_1}, \mathrm{BB}_1, \mathrm{Ins}_{b_2}, \mathrm{BB}_2) \Rightarrow (\mathrm{Var}_b|_{t_i+\Delta t} \rightarrow \mathrm{Var}_b|_{t_j+\Delta t})$, and $\exists \mathrm{du}(\mathrm{Var}_a, \mathrm{Ins}_{a_1}, \mathrm{BB}_1, \mathrm{Ins}_{a_2}, \mathrm{BB}_2, \mathrm{Ins}_{a_3}, \mathrm{BB}_3)$, $\mathrm{du}(\mathrm{Var}_b, \mathrm{Ins}_{b_1}, \mathrm{BB}_1, \mathrm{Ins}_{b_2}, \mathrm{BB}_2, \mathrm{Ins}_{b_3}, \mathrm{BB}_3) \in P$, $\mathrm{Ins}_{a_3}$ is executed at $t_k$ $(0 < i < j < k)$, $\mathrm{Ins}_{b_3}$ is executed at $t_k + \Delta t$, $\mathrm{du}(\mathrm{Var}_a, \mathrm{Ins}_{a_1}, \mathrm{BB}_1, \mathrm{Ins}_{a_2}, \mathrm{BB}_2, \mathrm{Ins}_{a_3}, \mathrm{BB}_3) \Rightarrow (\mathrm{Var}_a|_{t_j} = \mathrm{Var}_a|_{t_k})$, $\mathrm{du}(\mathrm{Var}_b, \mathrm{Ins}_{b_1}, \mathrm{BB}_1, \mathrm{Ins}_{b_2}, \mathrm{BB}_2, \mathrm{Ins}_{b_3}, \mathrm{BB}_3) \Rightarrow (\mathrm{Var}_b|_{t_j+\Delta t} = \mathrm{Var}_b|_{t_k+\Delta t})$, then $\mathrm{du}(\mathrm{Var}_a, \mathrm{Ins}_{a_1}, \mathrm{BB}_1, \mathrm{Ins}_{a_2}, \mathrm{BB}_2, \mathrm{Ins}_{a_3}, \mathrm{BB}_3) \wedge \mathrm{du}(\mathrm{Var}_b, \mathrm{Ins}_{b_1}, \mathrm{BB}_1, \mathrm{Ins}_{b_2}, \mathrm{BB}_2, \mathrm{Ins}_{b_3}, \mathrm{BB}_3) \Rightarrow (\mathrm{Var}_a, \mathrm{Var}_b)_{\mathrm{state}}$.

### 3.3 Key techniques

#### 3.3.1 Analysis of transitions of protocol states

On the assumption that protocol states will change with input and output packages in the session, the current protocol state is stored mainly in certain data structures in heap and the data/BSS segment, and updated with each package exchange in the form of a request-reply as time goes on. That is to say, when a protocol program receives and sends data packets, it often corresponds to the state transition of the program or even the protocol. In particular, the protocol state is represented by global or static variables from the perspective of software, whose lifetime goes across certain individual package exchanges, spanning almost an entire session. Considering this feature, we can use a dynamic instrumentation method to insert some probe codes for capturing the request-reply data, take notes about where they are stored, and record the timestamp of state transitions and the sequence of the current package in the session when communication entities process data packets by system calls on the basis of

parsing the semantics of instructions.

#### 3.3.2 Analysis of heap and data segment

According to the memory management mechanism of mainstream operating systems, different parts of a binary are loaded into different areas in the memory address space. According to the depiction above, protocol software stateful variables may exist in the following forms:

1. Variables in heap. Heap is a piece of virtual memory space that can be dynamically allocated to the running process on demand. Variables in heap are necessarily handled by system calls, for instance, malloc (similarly including calloc and realloc) for allocating, free for releasing, and mem_read/mem_write for reading and writing. Therefore, we can hook these system calls mentioned above to implement dynamic instrumentation when they are called.

2. Variables in the BSS segment. In the virtual memory address space of an executable file, the BSS segment (.bss) is used mainly to store uninitialized global variables and local static variables. Memory in BSS is allocated statically. The value of an uninitialized variable is usually set to 0, and will actually be assigned when initialized. The starting address and spatial scale of BSS can be found by virtual memory address space mapping when the binary is loaded. Then we can monitor all the operations in this specific area for dynamic instrumentation.

3. Variables in the data segment. The data segment (.data) is used to hold the initialized global static variables and local static variables. These two types of variables are handled and used in the same way as constants or read-only variables and are stateless. So, these variables can be ignored in the process of stateful variable identification.

It is worth mentioning that the data segment and BSS segment are combined and referred to the data segment in Linux, so the latter two forms above are processed together accordingly.

Based on the analyses above, we can use a dynamic instrumentation method to insert some probe codes for monitoring allocation and release in the address range that belongs to the heap, and continuously tracking operations by memory_read and memory_write events in the heap, BSS, and data segment. Then we can obtain a view of the definition-usage chain of variables in accordance with their evolution throughout the life cycle.

#### 3.3.3 Analysis of the definition-usage chain

Based on the characteristics of the definition-usage chain, operation security analysis of the definition-usage chain can be carried out to estimate whether the variable has a stateful characteristic.

For one single variable, in the case of no associated variable, if its definition-usage relationship is concentrated in a basic block, it is reasonable to infer that it does not have any stateful characteristic. However, if its definition-usage cross-spans several basic blocks, the sequence of reading and writing operations should be considered. When it contains "write-before-read" spanning basic blocks, the variable is determined to have a stateful characteristic. For multiple variables with correlation, reading and writing operations of these variables are inspected by relative and contrastive analysis. If there is a situation of allocating, using, or releasing a group of co-related variables simultaneously, we can make a deduction that they are all stateful.

Specifically, for this, information related to the protocol state, such as the authentication status, encryption suites, and client data to be processed, is typically stored in the form of structures. A logical address calculated by the base plus an offset is mostly applied to access this type of data structure. In view of such features, the states of a structure can be determined according to recorded mem_read/mem_write events with the base and the size of the address space where the memory block is located. If the logical address is in a certain range of an allocated memory block in heap, the offset from the base of the block will be recorded, and it is predicted to be a structure variable. If several structure variables allocated in a certain position are read or written in a tiny time span during the program execution, it can be inferred that there is a certain correlation between these variables.

#### 3.3.4 Discovering stateful variables

According to the analysis of the characteristics of stateful variables in Section 3.2, we describe and record the properties of the stateful variables with such a data structure in the implementation. The attributes that depict a stateful variable consist of a starting address, size, segment/section, allocation

time, freeing time, and a number of marks that reflect operations on the variable stored in this virtual memory block.

In addition, we summarize some information that needs to be used in the analysis process and extract it into a tiny database. Such information may include, for example, details of memory address distribution of the binary, the format of packages to be sent to the program, and the states and behaviors of the protocol software (Table 1). This information forms our knowledge base.

**Table 1 Components of the knowledge base**

| Content | Details |
|---|---|
| Memory address distribution of the program | Information about the program header, section headers, and allocation range of the virtual memory address space, which can be continuously obtained before and during the running of the program by the ELF parsing tool |
| Format of input data packages | Packages for testing are generated by capturing network traffic or crafting data manually, the format of which can be described by JSON to assist in locating input data during the analysis. |
| Coarse-grained behaviors of protocol software | Information recorded dynamically during the execution of the program, including the action and time of handling input/output packages, control flow of the execution path, and state flag inferred from protocol specification |

Based on the above elements, we use the algorithm shown in Fig. 1 to look for stateful variables that meet our requirements. Because the behaviors, interactions, and handled packages of different protocol software programs are different, we summarize some basic information of each object to be tested for processing by the algorithm, which is the knowledge base in Fig. 1. Therefore, we can apply this approach to binaries of protocol software for discovering stateful variables automatically.

## 4 Implementation

Our prototype system is implemented on top of the PANDA dynamic analysis platform, and developed through a Python interface named PyPANDA, which shields the discrepancies between analysis

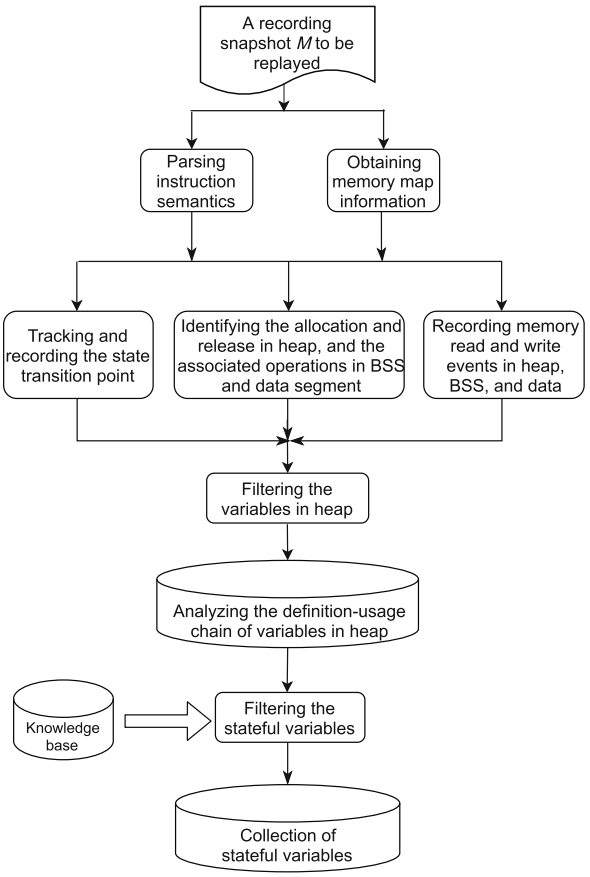


**Fig. 1 Flowchart of the automatic discovery algorithm for stateful variables of network protocol software**

tasks and guest virtual machine behavior and reduces the access threshold of whole-system dynamic analysis. So, we can use the plugin mechanism in PyPANDA for our analysis tasks, replaced with the analysis logic mentioned above. Fig. 2 shows the overall architecture implemented by our prototype system composed of four parts: recording and replaying module, protocol state analysis module, address space analysis module, and definition-usage chain analysis module. For details on the implementation of each module, please refer to the supplementary materials.

## 5 Evaluation

The experiments in this section are designed to evaluate the prototype system in two parts, testing in a benchmark, ProFuzzBench (see Section 1.2 in the supplementary materials), and testing in real-world programs. Then we focus on the experimental results to answer the following three research questions:

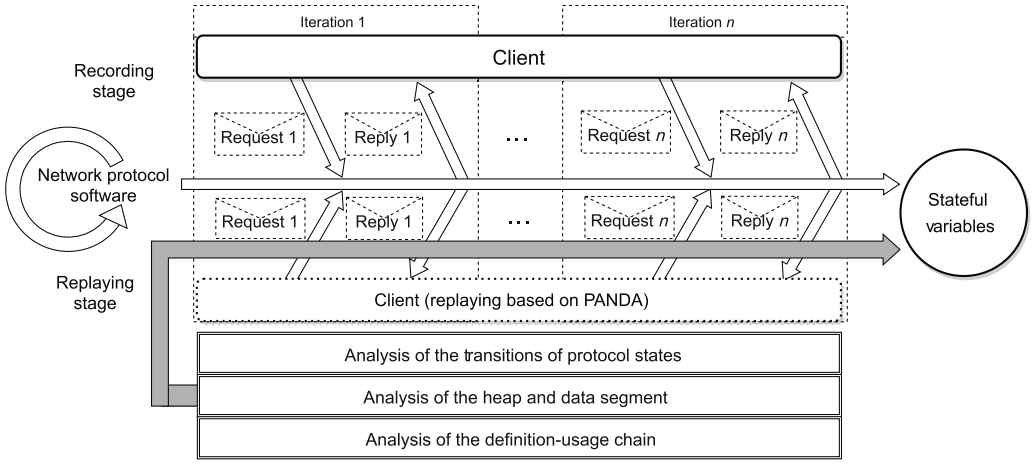


**Fig. 2 Overview of the prototype system for automatic discovery of stateful variables in network protocol software**

**Q1** (Effectiveness) Can the prototype system actually discover stateful variables automatically?

To answer this question, we will conduct experiments on nine sorts of protocol software programs in the benchmark and two kinds of real-world software programs.

**Q2** (Efficiency) Can the prototype system discover stateful variables fast enough?

To answer this question, we will perform experiments on these protocol software programs by one regular communicating interaction. Then we check the execution time taken by the protocol software when it performs normal execution and with our analysis plugins loaded.

**Q3** (Determinacy) Can the prototype system make a deterministic replay of the execution process of the protocol program?

To answer this question, we will compare the translation blocks visited on the execution paths of the protocol program between the recording stage and the replaying stage, to check if there is any divergence in the analysis.

During the evaluation, our analysis experiments are performed on a server with Dual CPU Intel Xeon E5-4650 v4 @ 2.20 GHz and 128 GB memory, running the Ubuntu Server 16.04 OS. The PANDA framework is capsulated in Docker, and the analysis tools we developed are loaded into Docker to target different programs. As mentioned above, our experimental subjects of network protocol software consist of two parts. For a benchmark, we select nine protocols and their corresponding implementations from ProFuzzBench as experimental objects, namely: DAAP/forked-daapd, 28.3; DICOM/Dcmtk, 3.6.6; DNS/Dnsmasq, 2.86; DTLS/TinyDTLS, a refactored version that could be easier to use as a standalone (e.g., without bindings to a specific IP-stack); FTP/LingtFTP, 2.2; RTSP/Live555, 2020.04.24; SIP/Kamailio, 5.5.0; SMTP/Exim, 4.95; SSH/OpenSSH, 8.8p1.

Beyond the test cases in the benchmark, we apply our analysis technique on real-world programs: TLS/OpenSSL, 1.1.1j; SSH/LibSSH, 0.8.9. These two software programs are based on TLS and SSH, respectively, and have been picked for evaluation because both of them define a series of negotiation and authentication stages between the server and the client in the communication to establish a more secure network connection. The working processes of these protocols have typical stateful characteristics and are suitable for our evaluation experiments.

### 5.1 Effectiveness

To evaluate if our approach can be applied to various protocol software, we introduce some widely used metrics: true positive rate (TPR) or recall, true negative rate (TNR), false positive rate (FPR), false negative rate (FNR), precision ($P$), and F1-measure (F1) (see Section 3 in the supplementary materials).

We perform our implementations on the 11 protocol software programs mentioned above, and record the number of stateful variables automatically discovered (Table 2). Subsequently, we obtain the source codes, the numbers of global variables

and variables with stateful characteristics in the code through human checking, and then the results of six evaluation metrics (Table 3), which indicate the effectiveness of the prototype system. In conclusion, the arithmetic mean values of TPR and $P$ are 82% and 96%, respectively.

**Table 2 The number of stateful variables automatically discovered by the prototype system compared to the numbers of global variables and stateful variables estimated by manual inspection from 11 protocol software programs**

| Protocol software | Number of variables | | |
|---|---|---|---|
| | SVDA | GVFM | SVEM |
| DAAP/forked-daapd | 314 | 446 | 388 |
| DICOM/Dcmtk | 372 | 593 | 453 |
| DNS/Dnsmasq | 0 | 288 | 163 |
| DTLS/TinyDTLS | 168 | 226 | 201 |
| FTP/LingtFTP | 216 | 379 | 263 |
| RTSP/Live555 | 398 | 580 | 477 |
| SIP/Kamailio | 499 | 647 | 593 |
| SMTP/Exim | 569 | 742 | 681 |
| SSH/OpenSSH | 759 | 962 | 920 |
| TLS/OpenSSL | 738 | 934 | 897 |
| SSH/LibSSH | 405 | 591 | 479 |

SVDA: stateful variables discovered automatically; GVFM: global variables found manually; SVEM: stateful variables estimated manually

From the experimental results, we observe that the prototype system can discover most of the stateful variables accurately in the software of stateful network protocols, for example, LightFTP, Live555, Exim, OpenSSL, and LibSSH. For stateless network protocols, the prototype system gives a conclusion that no stateful variable has been found in the software, such as Dnsmasq. The reason for our decision is that the request sent to the Dnsmasq server in the experiment is only to query the IP address corresponding to one testing domain name, such as /daemon.com/. In such a simple request-and-response iteration, the Dnsmasq server enters into neither multi-session concurrency nor multi-level nested logic, so it is reasonable that stateful characteristics are not identified.

Note that we choose open-source protocol software for experimental purposes due to the requirements of the experimental evaluation metrics, so it can be used for manual verification after automatic discovery of stateful variables is finished. However, this technical solution is designed based on the characteristics of the executable file. In principle, it can still work without the source code. Under the circumstances, additional work will be needed to verify the discovered stateful variables.

### 5.2 Efficiency

For the convenience of experimentation and migration, we encapsulate the PANDA framework and analysis module in Docker, and then test 11 experimental subjects using Dockers. During the experiment, we send data packages of a specific network protocol through the client program to simulate the process of message interaction between the client and the server. Furthermore, we keep and compare the duration of the original independent running and instrumentation running through PANDA after the analysis plug-in is loaded. Specifically, the original execution time of the protocol program is the time it takes from the start of the server until it finishes an iteration of package interaction by the protocol with the client. The instrumentation execution time of the protocol program is the time for loading the preserved program execution recording and the analysis plug-in through PANDA, running the instrumented analysis code for stateful variables during the iteration of package interaction between the server and the client by replaying, and collecting the generated information and outputs in the end. Table 4 shows the original execution time and instrumentation execution time of 11 protocol software programs used in our evaluation.

Indeed, the instrumentation execution time is determined by the complexity of the protocol software, the capacity of interactive data packages, and the quantity of instrumentation instructions. It can be seen that the time cost of analyzing software that only interacts with data packages (Live555, Kamailio, etc.) is within a reasonable range. For software with complicated behaviors, such as file transferring (forked-daapd) or encrypting (OpenSSL), there would be a relative slowdown versus its original execution time. Because we do not conduct the parsing and prediction of function semantics, the time overhead is wasted mainly on processing these complex functions. Considering that our evaluation experiments are performed on PANDA in Docker, we believe that if it is executed on a real machine in a production environment, the execution efficiency will be improved to a certain extent. In general, the time overhead above is acceptable.

**Table 3 Results of six evaluation metrics for indicating the effectiveness of our prototype system**

| Protocol software | TPR (%) | TNR (%) | FPR (%) | FNR (%) | $P$ (%) | F1 (%) |
|---|---|---|---|---|---|---|
| DAAP/forked-daapd | 80.58 | 89.23 | 10.77 | 19.42 | 97.77 | 88.35 |
| DICOM/Dcmtk | 81.88 | 95.89 | 4.11 | 18.12 | 98.39 | 89.38 |
| DNS/Dnsmasq | 0.00 | 100.00 | 0.00 | 100.00 | NA | NA |
| DTLS/TinyDTLS | 81.87 | 56.82 | 43.18 | 18.13 | 88.69 | 85.14 |
| FTP/LingtFTP | 81.12 | 89.23 | 10.77 | 18.88 | 93.52 | 86.88 |
| RTSP/Live555 | 82.94 | 88.03 | 11.97 | 17.06 | 96.48 | 89.20 |
| SIP/Kamailio | 83.57 | 72.00 | 28.00 | 16.43 | 95.79 | 89.26 |
| SMTP/Exim | 83.16 | 79.22 | 20.78 | 16.84 | 97.19 | 89.63 |
| SSH/OpenSSH | 82.23 | 75.00 | 25.00 | 17.77 | 98.16 | 89.49 |
| TLS/OpenSSL | 82.09 | 80.43 | 19.57 | 17.91 | 98.78 | 89.67 |
| SSH/LibSSH | 84.05 | 88.19 | 11.81 | 15.95 | 96.30 | 89.76 |

**Table 4 The original and instrumentation execution time during recording and replaying analysis**

| Protocol software | Original execution time (s) | Instrumentation execution time (s) |
|---|---|---|
| forked-daapd | 41.2938 | 8919.4608 |
| Dcmtk | 205.4913 | 40 738.0453 |
| Dnsmasq | 211.2848 | 2156.0601 |
| TinyDTLS | 183.6226 | 661.6658 |
| LingtFTP | 389.9277 | 61 566.1171 |
| Live555 | 953.3180 | 63 872.3061 |
| Kamailio | 255.3647 | 14 969.4787 |
| Exim | 276.7425 | 51 114.5468 |
| OpenSSH | 259.9624 | 45 595.6720 |
| OpenSSL | 243.7826 | 51 925.6938 |
| LibSSH | 50.5591 | 3521.5197 |

### 5.3 Determinacy

To check the determinacy of replaying, we focus on the execution paths of the program. The base addresses of the translation blocks passing on the program execution path will be recorded in proper order during the recording stage. The translation blocks on the replay path will also be logged into a set. Then, we compare the set of base addresses of translation blocks obtained in the above two stages and calculate the difference between them. According to our design, we set some acceptable divergences (about 2%) because they are a bit imprecise on the edges where we start and stop. Table 5 shows the results of 11 testing objects. The second and third columns show the numbers of translation blocks visited during the execution when recording and replaying respectively. The fourth column shows the number of translation blocks independently observed in both replaying and the original execution path. The fifth column is the percentage of the number of translation blocks covered by the execution path during the replay process to the total number of translation blocks in the original execution process.

In summary, the consistency between replay execution and the original execution can reach 98%, which is in line with the requirements for deterministic replaying in experimental design.

## 6 Discussion

We have compared our work with several existing mainstream solutions in terms of functionality (Table 6). Although we have achieved some success, discovering stateful variables in network protocol software is a really challenging task, and there are still a few limitations in the prototype system when encountering complex problems in the actual production environment.

The automatic discovery technology for stateful variables discussed above is based on the generation and utilization patterns of variables, with instrumentation analysis. As shown in efficiency evaluation, instrumentation execution will create some performance loss; it is an inherent flaw of dynamic taint analysis, with relatively high time cost and space overhead. For some complicated protocol software programs, loading runtime libraries and doing library function calls will also be involved in the running process. In such cases, whether the recording and replaying mechanism can remain deterministic and whether the instrumentation analysis of library function calls can remain efficacious by using our approach require further verification and evaluation.

As an implementation of one network protocol, the existence of stateful variables is a common

**Table 5 Determinacy of execution paths of network protocol software during recording and replaying analysis**

| Protocol software | $n_{\text{tb\_rec}}$ | $n_{\text{tb\_rep}}$ | $n_{\text{tb\_rep\_rec}}$ | Determinacy (%) |
|---|---|---|---|---|
| DAAP/forked-daapd | 13 581 | 13 704 | 13 580 | 99.993 |
| DICOM/Dcmtk | 30 727 | 30 891 | 30 726 | 99.997 |
| DNS/Dnsmasq | 20 677 | 20 852 | 20 677 | 100 |
| DTLS/TinyDTLS | 17 355 | 17 377 | 17 354 | 99.994 |
| FTP/LingtFTP | 29 034 | 29 100 | 29 034 | 100 |
| RTSP/Live555 | 27 470 | 27 532 | 27 470 | 100 |
| SIP/Kamailio | 36 004 | 36 108 | 36 001 | 99.992 |
| SMTP/Exim | 32 060 | 32 201 | 32 060 | 100 |
| SSH/OpenSSH | 39 828 | 39 319 | 39 243 | 98.531 |
| TLS/OpenSSL | 40 724 | 40 710 | 40 547 | 99.565 |
| SSH/LibSSH | 25 596 | 25 773 | 25 595 | 99.996 |

$n_{\text{tb\_rec}}$, $n_{\text{tb\_rep}}$, and $n_{\text{tb\_rep\_rec}}$ represent the numbers of translation blocks accessed during the execution of recording, accessed during the execution of replaying, and observed in both replaying and recording execution trace, respectively

**Table 6 Comparison of typical techniques related to identifying stateful variables**

| Technique or author | Targeted source code or binary | Static or dynamic | Adapted for network protocol | Analyzing program states | Identifying stateful variables |
|---|---|---|---|---|---|
| Behrad Garmany | Source | Static | No | No | No |
| InvsCov | Both | Both | Not fully | Yes | Partly |
| AFLNet | Both | Dynamic | Yes | Yes | No |
| UAFL | Source | Both | Not fully | Partly | Partly |
| StateAFL | Source | Dynamic | Yes | Yes | Partly |
| SNPSFuzz | Both | Dynamic | Yes | Yes | No |
| Our approach | Both | Both | Yes | Yes | Yes |

phenomenon, but not all of them are potentially dangerous. Our plan is to obtain key information about stateful variables and to identify key behaviors in the memory space. In the future, we may use symbolic taint analysis and conclic testing methods based on this knowledge, to assist the analysis tools in finding vulnerabilities more efficiently and accurately in network protocol software.

## 7 Related works

### 7.1 Static analysis of detection of variables

Static program analysis methods have been widely used in mining potential vulnerabilities, especially for detecting variables that may be mistaken or misused in programs. Giuffrida et al. (2013) presented an infrastructure for monitoring multiple types of variables. They defined a series of security constraints as invariants, and then checked the violation of these invariants by monitoring the real-time execution. A static value-flow analysis method was proposed by Ye et al. (2014), who built a value flow graph based on source code to measure the undefined variables. Safelnit (Milburn et al., 2017) is a binary-hardening-based approach for reducing the usage of uninitialized variables. Reading variables that have been allocated but not assigned in the heap and stacks can be discovered and detected by leveraging a multi-variant execution approach. Garmany et al. (2019) presented a static analysis framework that transforms the binary executables into a knowledge representation that builds the base for specifically crafted algorithms to find uninitialized variables. In addition, there are some systems that implement methods combining static and dynamic analysis (Bruening and Zhao, 2011; Stepanov and Serebryany, 2015). For one thing, certain knowledge can be generated by static analysis; for another thing, the usage of variables on the executed paths can be analyzed by fuzzing tools or testing suites guided by this knowledge with some appropriate corpus. In contrast, besides focusing on the life and use of variables, we combine the program's tracking state transformation with static analysis, and use instrumentation technology to gather and analyze the logic of variables, and as such, our approach is more targeted for analyzing network protocol software.

### 7.2 Stateful network protocol testing

There have been several techniques for discovering vulnerabilities in network protocol software. Fuzzing is a prevailing feasible approach for automatic testing of program states with exploration and detection. AFLNET (Pham et al., 2020) is the first greybox fuzzer for protocol implementations, identifying the server states and progressive regions in the state space by using the server's response codes. It then senses the message structures and state transitions of the network protocol. InvsCov (Fioraldi et al., 2021) tracks all variables at the basic-block level to learn likely invariants and partition the state space, so the feedback can be distinguished when an input violates these invariants and rewards it. StateAFL (Natella, 2022) instruments the protocol software at compile-time, inserting probes on memory allocations and network I/O operations, and infers the current protocol state of the target by analyzing snapshots of long-lived memory areas at runtime for stateful coverage-based greybox fuzzing. SNPS-Fuzzer (Li et al., 2022) dumps the context information about a specific state when the protocol software is running and restores the snapshot when the state needs to be fuzzed. It also proposes a message chain analysis algorithm to explore more and deeper states. Another type of vulnerability detection technique for network protocol software is model checking. Musuvathi and Engler (2004) proposed a model based on C and C++ code to check for errors in TCP/IP and AODV implementations. Brumley et al. (2007) analyzed different implementations of network protocol and built models from them, and then checked them against a protocol specification model to discover errors in implementations. The technology proposed in this paper can be applied to stateful network protocol testing, and the results can be an auxiliary engine for process scheduling in fuzzing, guiding fuzzing tools to more complex paths and deeper states, thereby improving the testing accuracy and efficiency.

### 7.3 Recording and replaying techniques

The recording and replaying techniques must continuously monitor the behavior of the OS and record the responses to external non-deterministic events for accuracy (Dunlap et al., 2002; Saito, 2005). How to properly handle non-deterministic events becomes a core mechanism for deterministic replaying. In general, it can be divided into a hardware method and a software method. The former method customizes hardware to handle non-deterministic events (Hower and Hill, 2008; Montesinos et al., 2008; Pokam et al., 2013), and the latter method relies on modified OS kernels (Bergan et al., 2010; Aviram et al., 2012). There are some typical works with the capability of deterministic recording and replaying for the whole system besides PANDA. SMP-ReVirt (Dunlap et al., 2008) supports recording and replaying of an entire unmodified system with multiprocessor hardware. RR (O'Callahan et al., 2017) is a lightweight tool that runs only one thread at a time to avoid non-deterministic events caused by interoperations between different CPU cores, working in the user space. In our prototype system, the functional modules are built upon PANDA. The main reasons why we make this choice are as follows: (1) repeatability—recordings can be replayed as many times as needed, which means that the same sequence of instructions is executed in the same order every time; (2) scalability—recordings can be replayed with instrumentations or plugins that are as heavy as needed for analysis; (3) determinacy—recordings can be deterministic because PANDA has taken a snapshot of the state of system first and logged trace point information to distinguish one state from another when it encounters non-deterministic events.

## 8 Conclusions

In this paper, we infer and discover the stateful variables that are used to store and reflect the state of network protocol software from the life cycle and operational characteristics of variables by analyzing prominent operations in the critical areas of memory from activities of the program. We design and implement a prototype system for automatically discovering stateful variables in protocol software, and then perform experiments for the prototype system on nine programs in ProFuzzBench and two real-world programs. The average TPR can reach 82%, and the average precision can be up to about 96%.

### Contributors

Jianxin HUANG and Bo YU designed the research. Jianxin HUANG processed the data and drafted the paper. Runhao LIU helped implement the computer code in the experiments. Jianxin HUANG and Bo YU revised and finalized the paper. Jinshu SU took the oversight and leadership

responsibility for the research.

## Compliance with ethics guidelines

Jianxin HUANG, Bo YU, Runhao LIU, and Jinshu SU declare that they have no conflict of interest.

## Data availability

The data that support the findings of this study are available from the corresponding authors upon reasonable request.

## References

Aviram A, Weng SC, Hu S, et al., 2012. Efficient system-enforced deterministic parallelism. *Commun ACM*, 55(5):111-119. https://doi.org/10.1145/2160718.2160742

Bergan T, Hunt N, Ceze L, et al., 2010. Deterministic process groups in DoS. Proc 9th USENIX Symp on Operating Systems Design and Implementation, p.177-191.

Bruening D, Zhao Q, 2011. Practical memory checking with Dr.Memory. Proc Int Symp on Code Generation and Optimization, p.213-223. https://doi.org/10.1109/CGO.2011.5764689

Brumley D, Caballero J, Liang ZK, et al., 2007. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. Proc 16th USENIX Security Symp, p.213-228.

Dolan-Gavitt B, Hodosh J, Hulin P, et al., 2015. Repeatable reverse engineering with PANDA. Proc 5th Program Protection and Reverse Engineering Workshop, Article 4. https://doi.org/10.1145/2843859.2843867

Dunlap GW, King ST, Cinar S, et al., 2002. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Oper Syst Rev*, 36(SI):211-224. https://doi.org/10.1145/844128.844148

Dunlap GW, Lucchetti DG, Fetterman MA, et al., 2008. Execution replay of multiprocessor virtual machines. Proc 4th ACM SIGPLAN/SIGOPS Int Conf on Virtual Execution Environments, p.121-130. https://doi.org/10.1145/1346256.1346273

Fioraldi A, D'Elia DC, Balzarotti D, 2021. The use of likely invariants as feedback for fuzzers. Proc 30th USENIX Security Symp, p.2829-2846.

Garmany B, Stoffel M, Gawlik R, et al., 2019. Static detection of uninitialized stack variables in binary code. Proc 24th European Symp on Research in Computer Security, p.68-87. https://doi.org/10.1007/978-3-030-29962-0_4

Giuffrida C, Cavallaro L, Tanenbaum AS, 2013. Practical automated vulnerability monitoring using program state invariants. Proc 43rd Annual IEEE/IFIP Int Conf on Dependable Systems and Networks, p.1-12. https://doi.org/10.1109/DSN.2013.6575318

Hower DR, Hill MD, 2008. Rerun: exploiting episodes for lightweight memory race recording. Proc Int Symp on Computer Architecture, p.265-276. https://doi.org/10.1109/ISCA.2008.26

Lee C, Bae J, Lee H, 2018. PRETT: protocol reverse engineering using binary tokens and network traces. Proc 33rd IFIP Int Conf on ICT Systems Security and Privacy Protection, p.141-155. https://doi.org/10.1007/978-3-319-99828-2_11

Li JQ, Li SY, Sun G, et al., 2022. SNPSFuzzer: a fast greybox fuzzer for stateful network protocols using snapshots. *IEEE Trans Inform Forens Secur*, 17:2673-2687. https://doi.org/10.1109/TIFS.2022.3192991

Milburn A, Bos H, Giuffrida C, 2017. Safelnit: comprehensive and practical mitigation of uninitialized read vulnerabilities. Proc 24th Annual Network and Distributed System Security Symp. https://doi.org/10.14722/ndss.2017.23183

Montesinos P, Ceze L, Torrellas J, 2008. DeLorean: recording and deterministically replaying shared-memory multiprocessor execution efficiently. *ACM SIGARCH Comput Archit News*, 36(3):289-300. https://doi.org/10.1145/1394608.1382146

Musuvathi M, Engler DR, 2004. Model checking large network protocol implementations. Proc 1st Conf on Symp on Networked Systems Design and Implementation, p.1-12.

Natella R, 2022. StateAFL: greybox fuzzing for stateful network servers. *Empir Softw Eng*, 27(7):191. https://doi.org/10.1007/s10664-022-10233-3

O'Callahan R, Jones C, Froyd N, et al., 2017. Engineering record and replay for deployability. Proc USENIX Conf on Usenix Annual Technical Conf, p.377-389.

Pham V, Böhme M, Roychoudhury A, 2020. AFLNET: a greybox fuzzer for network protocols. Proc 13th Int Conf on Software Testing, Validation and Verification, p.460-465. https://doi.org/10.1109/ICST46399.2020.00062

Pokam G, Danne K, Pereira C, et al., 2013. QuickRec: prototyping an Intel architecture extension for record and replay of multithreaded programs. *ACM SIGARCH Comput Archit News*, 41(3):643-654. https://doi.org/10.1145/2508148.2485977

Saito Y, 2005. Jockey: a user-space library for record-replay debugging. Proc 6th Int Symp on Automated Analysis-Driven Debugging, p.69-76. https://doi.org/10.1145/1085130.1085139

Song CX, Yu B, Zhou X, et al., 2019. SPFuzz: a hierarchical scheduling framework for stateful network protocol fuzzing. *IEEE Access*, 7:18490-18499. https://doi.org/10.1109/ACCESS.2019.2895025

Stepanov E, Serebryany K, 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. Proc IEEE/ACM Int Symp on Code Generation and Optimization, p.46-55. https://doi.org/10.1109/CGO.2015.7054186

Ye D, Sui YL, Xue JL, 2014. Accelerating dynamic detection of uses of undefined values with static value-flow analysis. Proc Annual IEEE/ACM Int Symp on Code Generation and Optimization, p.154-164. https://doi.org/10.1145/2544137.2544154

Yu B, Wang PF, Yue T, et al., 2019. Poster: fuzzing IoT firmware via multi-stage message generation. Proc ACM SIGSAC Conf on Computer and Communications Security, p.2525-2527. https://doi.org/10.1145/3319535.3363247

# List of supplementary materials

1 Background knowledge
2 Implementation details
3 Six evaluation metrics