Institute of Informatics
Chair for Software and Computational Systems
Prof. Dr. Dirk Beyer

Application of Software Verification to OpenBSD Network Modules

Moritz Buhl
m.buhl@campus.lmu.de
11549061

Period of Writing:	2018-04-09 – 2018-09-30
External Advisor:	Alexander Bluhm
Advisor:	Thomas Lemberger
Supervisor:	Prof. Dr. Dirk Beyer
Project Files:	`https://www.moritzbuhl.de/bachelor-thesis`

Abstract

Ensuring software quality is complicated. A lot of mistakes can be found early with tests, but test coverage or the number of tests are not precise measures to ensure correctness. Formal verification is a promising solution, but especially on mature software that was not built with verification in mind, little effort is known. To evaluate the use of formal verification on widely used software, known security fixes in OpenBSD are undone to rediscover a bug with CPAchecker. The process of adjusting the source code was documented and converting the source code in CPAchecker was corrected for a few constructs.

Contents

1 Introduction **1**

2 Related Work **3**

3 Background **4**
3.1 CPAchecker Architecture, Theory and Technologies 4
3.1.1 Architecture, Operation and Interface 4
3.1.2 Definition of Configurable Program Analysis 5
3.1.3 The CPA Algorithm 6
3.1.4 Symbolic Memory Graphs 7
3.1.5 Application-Oriented Features 8
3.2 OpenBSD Kernel and Network Concepts 8
3.2.1 Memory Buffers Specialized for Network Data 8
3.2.2 Cohesion and Flow in the Network Stack 10

4 Considerations and Adjustments Before Verification **13**
4.1 Features and Restrictions in the Kernel Environment 13
4.2 Required Adjustments to the OpenBSD Source Code 13
4.2.1 Preprocessing Network Modules 14
4.2.2 Simplifying Memory Management 14
4.2.3 Removing Inline Assembler in the Source Code 15
4.2.4 Problems with Abstractions and Precision 15
4.3 Patches to CPAchecker for Usage on Kernel Code 16
4.3.1 Simplification of Constant Binary Expressions 16
4.3.2 Proper Handling of Duplicate Declarations 17
4.3.3 Other Necessary Changes 17

5 Using CPAchecker on OpenBSD Network Code **18**
5.1 EtherIP 18

6 Evaluation **19**
6.1 Running Environment 19
6.2 Considered Software and Systems 20
6.3 Results 20
6.4 Discussion 20
6.5 Future Work 21
6.6 Conclusion 21

List of Tables

1 All patches in the network stack for incoming packets related to the mbuf API since OpenBSD 5.6 2
2 Verification results 20

List of Figures

1 Workflow of CPAchecker 5
2 A OpenBSD-style DLL (top) and its SMG (bottom), like [9] . 7
3 Mbuf data structure and important related identifiers 9
4 Usage of the mbuf API and other important APIs in the network stack 11
5 Network input processing in OpenBSD 12
6 Example program misrepresented as CFA 17

List of Algorithms

1 Simple version of the CPA algorihm 7

Listings

1 Missing NULL check in MPLS 1
2 Missing length check in MPLS 2
3 The adjusted `m_get()` function to use with CPAchecker 15
4 Parts of the `ip_deliver()` function and a pointer dereferece . 16
5 A constant conditional expression in a variable initialization . 17
6 Multiple array declarations 17
7 Wrong return value in EtherIP 18
8 Required initialization to reach EtherIP 19

```
156   switch (rt_mpls->mpls_operation) {
157   case MPLS_OP_POP:
158     m = mpls_shim_pop(m);
159 +   if (m == NULL)
160 +     goto done;
```

Listing 1. Missing NULL check in MPLS

1 Introduction

A reliable and secure network stack is essential since errors in network code have a high chance of being remotely exploitable. The OpenBSD network stack includes abstractions to interface with network devices, implementations of all supported network protocols, routing functionality and a packet filter. Furthermore, all these functionalities use the same implementation to manage the network memory — the mbuf API.

CPAchecker is a tool for configurable software verification that is used successfully in finding errors in the Linux kernel. How difficult is it to apply CPAchecker on a similar program? To check on how applicable new verification technology is, the OpenBSD kernel should be interesting due to its size, quirks, and maturity.

As the OpenBSD Project is known to be the "#1 most secure [general-purpose] operating system"[1], it is used for routing network traffic in facilities with high security requirements. This fundamental requirement for security in the OpenBSD project is also noticeable in the early adoption of IPsec and many other security improvements[2]. Until May 2018 a considerable amount of errors was uncovered by Maxime Villard as he audited the BSD operating systems network stack[3]. The errors found, show a pattern of mistakes related to the mbuf API which is used everywhere in the network stack. Earlier patches published by the OpenBSD project show similar errors.

Table 1 lists patches related to the mbuf API since OpenBSD 5.6 and shows that recurring patterns of errors can be divided into three categories: erroneous calculation of length or offset, memory handling errors and missing NULL pointer checks. It is also noticeable that many patches are related to the next generation protocols IPv6 and IPsec. Seven out of ten bugs listed in Table 1 occurred in network functions that handle incoming network packets.

[1] See `https://www.openbsd.org/goals.html`

[2] A list is available at `https://www.openbsd.org/crypto.html`

[3] Available at `https://blog.netbsd.org/tnf/entry/network_security_audit`

[4] For a full list see `https://www.openbsd.org/errata.html`

Patch	length calculation	memory handling	NULL pointers	Location
003_mpls	✗	✗	✗	`mpls_input()`
005_ahopts	✗	✗		`ah_massage_header()`[*sic*]
006_ipseclen	✗			IPsec `callback()`s
006_prevhdr			✗	`rip6_input()`, `frag6_input()`
007_etherip		✗		`ip_etherip_input()`
010_ahauth	✗			IPsec `_input()`s
010_pipex	✗			`pipex_pptp_input()`

Table 1: All patches in the network stack for incoming packets related to the mbuf API since OpenBSD 5.6[4]

```
255 + if (m->m_len < sizeof(u_char) &&
256 +     (m = m_pullup(m, sizeof(u_char))) == NULL)
257 +   return (NULL);
```

Listing 2. Missing length check in MPLS

Listing 1 shows how a NULL-pointer dereference is prevented by explicitly checking a return value for NULL. Memory handling errors and pointer dereferences are declared as undefined behavior in the standard [20, §7.20.3, §6.3.2.3].

However, calculating the length of network data is not, and an own specification for such errors is required. E.g., Listing 2 is possibly leaking information, but not necessarily accessing invalid memory. Because the length calculation is individual for each protocol, it is not easy to find a unified way to ensure valid length calculations. Especially problematic is that network data and length only need to be consistent after manipulation. Each processing function can manipulate length and content with inconsistencies during packet processing.

Primarily because CPAchecker provides methods to detect or rule out the occurrence of the latter two classes, a model checking approach could rule out future errors in the network stack. But also because of the possibility to define rules for data buffers to identify invalid data accesses, CPAchecker could be used in addition to regression testing to prevent programming errors and ensure correct behavior.

The next chapter will address required knowledge on the network concepts used in OpenBSD as well as CPAchecker and its approaches to check for correct behavoir in the source code. Before application, the source code of OpenBSD and CPAchecker required adjustments. Furthermore, it was necessary to decide on strategies to find the known errors presented in Table 1.

2 Related Work

Linux Driver Verification The Linux Driver Verification (LDV) program [23] finds bugs in the Linux kernel API with the static software verification framework Klever [18]. The Linux kernel is divided into its subsystems. Each subsystem is verified together with extensions that use its interface by checking for various safety and liveness properties. By not requiring complete specifications for each subsystems false-positive errors occur. The modules are processed with multiple verifiers i.a. CPAchecker. Reports by the verifiers are collected and further analyzed.

In contrast to the Linux kernel, OpenBSD kernel does not provide dynamically loaded kernel modules, and there is a less strict interface in the source code. The LDV program is checking the system for new bugs in many Linux modules. Instead, the OpenBSD code will be searched for already known bugs in the network stack.

Network Semantics The NetSem [7] project specifies various protocols used on the internet with HOL. The FreeBSD network stack was examined to create the specifications. It was not a goal to prove the correctness of the protocols or to find bugs in the protocols but to test implementations for conformance. As the project states, the correctness of a protocol implementation depends on the language it is implemented in, and the C language is not viable. Instead, the errors in OpenBSD are errors in the implementation that resulted in kernel freezes.

VigNAT The VigNAT project shows that the formal verification of a Network Address Translator (NAT) does not necessarily come at the cost of performance [24]. The verified components consist only of those that are used in the NAT related execution path of the stack, including the NAT network functions, the network interface drivers, and the required data structures [19]. To ensure a crash-free and memory-safe NAT system, KLEE [8] based symbolic execution and VeriFast [12] based separation logic are used for the proof. The network functions are implemented only for this project and focus on simplicity by avoiding pointer arithmetic and other dangerous

C constructs. The use of symbolic execution required to replace complex data structures in the kernel-bypass framework and not to use multithreading. The main difference is that the network functions in this thesis are used in real-world software and come with the complexity of kernel data structures and low-level memory operations.

Predator Predator [11] is a shape analyzer that finds memory safety errors as illegal pointer dereferences, memory leaks and double-free errors by using Symbolic Memory Graphs (SMG). The predator analyzer can be parallelized to quickly find incorrect memory handling by searching in different depths of the state space [15]. SMGs will be used later on to find similar errors but the support for non-pointer data is limited in Predator, and it would not provide enough flexibility to uncover all the bugs mentioned.

Hyperkernel Hyperkernel is an approach to a verified kernel that is proved automatically. The kernel is based on the UNIX-like xv6 interface and was built to reduce complexity during verification. The kernel interface is adjusted to interface with the Z3 SMT solver to enable a quick method to check the compliance of changes to the kernel. Additionally, the size of Hyperkernel is not comparable with widely used general purpose operating systems.

3 Background

3.1 CPAchecker Architecture, Theory and Technologies

CPAchecker is developed as a flexible software verification framework with stability in mind. It is intended for prototyping verification ideas by providing an interface of Configurable Program Analysis (CPA) [4]. The CPAchecker project has the goal of accelerating the transfer of new verification technology to industrial practice [2]. Due to this, CPAchecker provides a range of different technologies and features to create either proof of correctness for a program or a counterexample (witness) to reach a non-conforming state.

3.1.1 Architecture, Operation and Interface

CPAchecker verifies source code by using the CPA algorithm on a control-flow automata (CFA) [5]. The CPA algorithm decides on the correctness of the CFA by checking for previously defined constraints [3]. This also means that if the process of creating a CFA from source code is wrong, the CFA

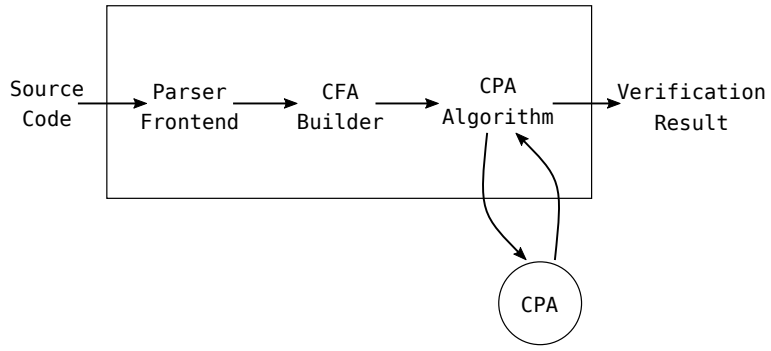


Fig. 1. Workflow of CPAchecker

will not match the source code anymore and the result of the CPA algorithm will be useless.

Figure 1 shows how CPAchecker is used to verify C-source code. The C-parser is based on the Eclipse CDT[5] project and creates a syntax tree from the source code that is then used to generate a CFA. CPAchecker will then iterate over it and check each transition for correctness by interfacing with different analyses. Each analysis provides algorithms to search for reachability on each state and abort if constraints are not met. If all possible states are reached, and the algorithm was not previously aborted, the CFA would satisfy the specified model.

3.1.2 Definition of Configurable Program Analysis

A CFA A [6] uses an imperative programming language which only consists of assignments and assume operations. It is defined as $A = (L, l_0, G)$ and consists of set of program locations L, a set of control-flow edges G and the initial location l_0. Each edge $g \in G \subseteq L \times Ops \times L$ represents an operation that is executed when transitioning to the next node.

A CPA [3] is defined as $\mathbb{D} = (D, \rightsquigarrow, \texttt{merge}, \texttt{stop})$ where D is an abstract domain, $\rightsquigarrow$ is a transfer relation, `merge` is an operator to combine two abstract states and `stop` is a termination check operator.

The abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ consists of a set C which represents concrete states, a semi-lattice $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ and a concretization function $[\![\cdot]\!] : E \to 2^C$ which assigns a set of concrete states to each abstract state. The semi-lattice of elements E, the top element $\top \in E$, a bottom element $\bot \in E$, a preorder $\sqsubseteq \subseteq E \times E$, and the total function $\sqcup : E \times E \to E$ which is called the join operator.

[5] Available at `https://www.eclipse.org/cdt/`

The transfer relation $\rightsquigarrow \subseteq E \times G \times E$ assigns each abstract state $e \in E$ to new abstract states $e' \in E$ — its successors. Each transition is labeled with a control-flow edge $g \in G$. Writing $e \overset{g}{\rightsquigarrow} e'$ is the same as $(e, g, e') \in \rightsquigarrow$, and $e \rightsquigarrow e'$ means that a g exists with $e \overset{g}{\rightsquigarrow} e'$.

The operator `merge` : $E \times E \to E$ combines two abstract states. The `merge` operator can only make a state more abstract: $e' \sqsubseteq \texttt{merge}(e, e')$.

The termination check `stop` : $E \times 2^E \to \mathcal{B}$ checks that the given abstract state is covered by the set of abstract states.

In CPAchecker, the definition of a CPA is realized as an interface that only needs an implementation of the mentioned operators. The CPA algorithm will then use the implemented CPA in the analysis.

3.1.3 The CPA Algorithm

Algorithm 1 shows the CPA algorithm which uses a CPA $\mathbb{D}$ and two sets `reached` and `waitlist`. The `reached` set is used to keep track of the already reached states, the `waitlist` is used to store the neighbors of previous states, that need to be worked off. The algorithm starts with the first node e_0 which is usually the initialization of global variables and the call to the `main()` function.

Algorithm 1 $CPA(\mathbb{D}, e_0)$ similar to [3]

Input: a CPA $\mathbb{D}$ and an initial state $e_0 \in E$
Output: a set *reached* of reachable states
$waitlist \leftarrow e_0$
$reached \leftarrow e_0$
while $waitlist \neq \emptyset$ **do**
    pop e from $waitlist$
    for each e' with $e \rightsquigarrow e'$ **do**
        for each $e'' \in reached$ **do**
            $e_{new} \leftarrow merge(e', e'')$
            if $e_{new} \neq e''$ **then**
                $waitlist \leftarrow (waitlist \cup \{e_{new}\})\ \{e''\}$
                $reached \leftarrow (reached \cup \{e_{new}\})\ \{e''\}$
        if $\neg stop(e', reached)$ **then**
            $waitlist \leftarrow waitlist \cup \{e'\}$
            $reached \leftarrow reached \cup \{e'\}$
return *reached*

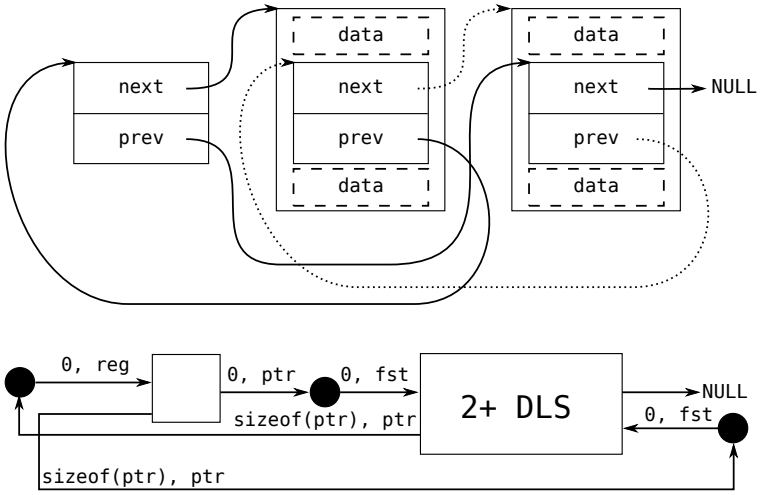


Fig. 2. A OpenBSD-style DLL (top) and its SMG (bottom), like [9]

3.1.4 Symbolic Memory Graphs

SMGs [9] are an efficient way to represent list structures and to efficiently encode program configurations. SMGs consist of objects which are represented as boxes, and values which are represented as circles. An object is split into regions — a continuous area of memory — and list segments —

an abstraction of multiple members of a list. Additionally, every SMG has a null object, which represents the `NULL` constant from the C language. Figure 2 shows how the boxes and circles are connected by *has-value* edges and *points-to* edges. Has-value edges represent the owned values with an offset in the objects block and a type. Points-to edges have an address and a target specifier (before the object, behind the object, inside of the object).

3.1.5 Application-Oriented Features

CPAchecker allows internally defined functions in the source code to empower the program to communicate restrictions on the current state[6]. The function `__VERIFIER_nondet_X()` returns a non deterministic value of the type `X`. This is useful when creating a new entry point to a function which takes user input. When choosing a point to start the verification at, it is necessary to additionally restrict the possible input with the function `__VERIFIER_assume()`. This function only returns if the expression passed to the function is true. Therefore it is possible to filter out assumptions on the input that usually would be handled in a previous path of execution. To mark a program path as unreachable the function `__VERIFIER_error()` is provided. A call to this function should not happen and the verification will result in the code not meeting the specification. CPAchecker also provides functions to mark atomic sequences for verifying multi-threaded run-time environments.

3.2 OpenBSD Kernel and Network Concepts

OpenBSD is a POSIX-like UNIX-based operating system. Its network stack derived from the 4.2BSD project which contained the first widely available release of TCP/IP. A range of changes followed with the introduction of IPv6 and IPsec by the KAME and WIDE projects with the goal to provide an open source implementation of the next generation IP protocol suite [10]. The figures in this section are based on [13] but required adjustments due to changes in the OpenBSD source code. The information in this section is based on [17, 21, 13, 14] and the source code because of the frequent changes.

3.2.1 Memory Buffers Specialized for Network Data

In the BSD kernels, network memory is managed in memory buffers (mbufs). Mbufs are used in the kernel from the network interface driver to the socket API and hold network data and information. Network data is either passed

[6]See `https://sv-comp.sosy-lab.org/2018/rules.php`

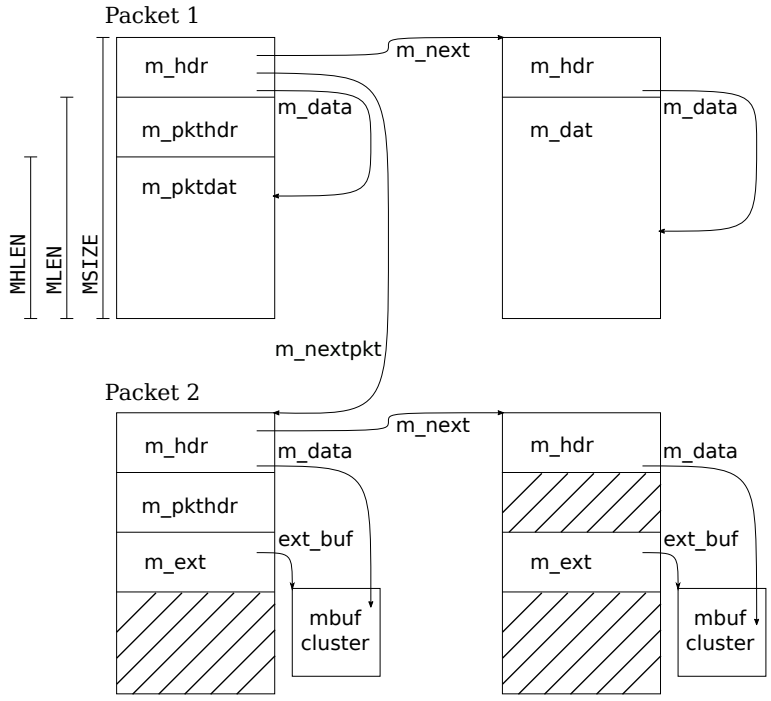


Fig. 3. Mbuf data structure and important related identifiers

from the userland via the socket API or from another network machine via a network adapter. Each network packet is stored in one or more mbufs. If a packet cannot fit into a single mbuf object, the packet data will be split over multiple mbufs. The mbufs are then chained by making each object hold a pointer to its successor.

Figure 3 shows two network packets, each split over two mbufs. This shows all four possible mbuf combinations linked together to store network data. Additionally, the `m_data` pointer is displayed which is used to access the packet data in a mbuf. A mbuf at least consists of a header `m_hdr` and a buffer `m_dat` or `m_pktdat` for network data. The header is a struct that contains the pointers `m_next` and `m_nextpkt` to create linked lists. It stores additional information like the buffer size and the `m_data` pointer to the buffer. The first mbuf in a chain holds further information about the network packet like length and protocol family in the `m_pkthdr` struct. The `M_PKTHDR` flag is used to mark the mbuf as the beginning of a packet. Since each mbuf has the same size the buffer will be smaller if it contains a `m_pkthdr` struct. There are three different length constants: the mbuf size `MSIZE`, the size `MLEN` of the buffer if the mbuf has a predecessor, and the size `MHLEN` of the buffer if the mbuf is the first element in the chain.

For modern network adapters, the driver allocates a buffer of mapped memory and then makes a previously allocated mbuf point to the buffer by setting the `m_data` and `ext_buf` pointers. After this, the buffer is handed to the network device. The mbuf then needs to be marked as extern by setting the `M_EXT` flag.

An external mbuf has uninitialized memory where the buffer used to be. The combination of the flags `M_PKTHDR` and `M_EXT` make four different layouts possible: a normal mbuf only with information about the mbuf and network data, a mbuf additionally containing a packet header, a mbuf pointing to an external buffer instead of holding network data and an external mbuf additionally containing a packet header.

To manipulate the data of a mbuf e.g. to unpack an encapsulated payload, `m_adj()`, `m_pullup()` and `m_copyback()` are used. A length value is required with each of these functions to indicate the start or end of the adjustable data. The manipulation usually depends on the network packet. False assumptions about network data are a reoccurring problem and could lead to kernel crashes in the past.

3.2.2 Cohesion and Flow in the Network Stack

Figure 4 shows a simplified view of the network stack in which the mbuf API is heavily used to handle the packet data. An incoming network packet is passed from the network adapter to the operating system. It is processed in the network stack by decapsulating the payload until it is passed through the socket API to the userland. An outgoing network packet is created due to the `write()` system call or after forwarding. It is then encapsulated and passed down the output functions. If an incoming packet is not conforming to the protocol specification, it should be dropped. If the packet is not destined for the current machine, it will be forwarded to the next machine after a route lookup. It is possible that the packet filter (pf) will alter the procedure.

Figure 5 shows the network stack for incoming packets. All the functions mentioned except `ether_input()`, `ip_input()`, `ip_local()`, `ip_deliver()`, `upd_input()` and `tcp_input()` are patched in at least one erratum from Table 1. The layer each function belongs to is mentioned in the bottom left. The displayed network stack is an implementation of the TCP/IP model and therefore the boxes are merely for orientation. To reduce the number of functions in Fig. 5, regular expressions are used in some function names. `ip6?_input()` represents the functions `ip_input()` and `ip6_input()`. The IPsec related input functions each have an implementation for IPv4 and IPv6. The `callback()` function also is unique for each function.

Network packets are put into the interface queues by the network de-

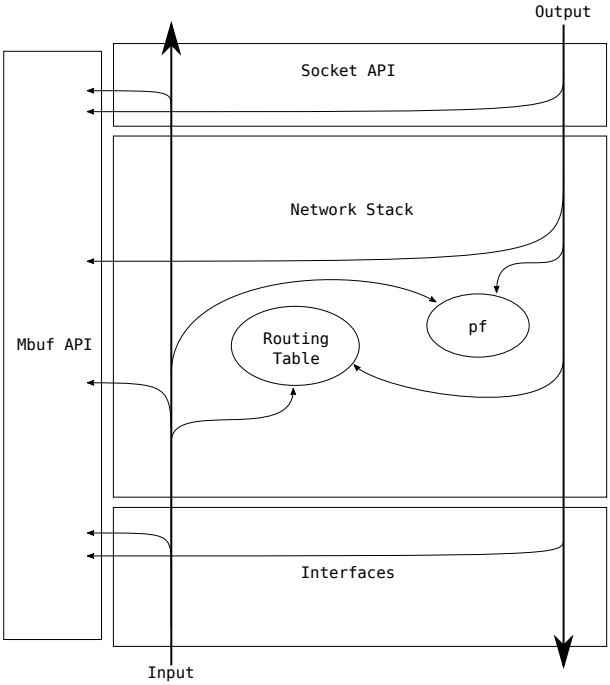


Fig. 4. Usage of the mbuf API and other important APIs in the network stack

vice as layer-2 packets. The most common input is an Ethernet frame that is then passed to `ether_input()`. After processing, the payload is either passed to the layer-3 input functions or if the packet uses multiprotocol label switching (MPLS), it will be forwarded immediately. IPv4 and IPv6 are the only supported layer-3 protocols. The IP input functions `ip_input()` and `ip6_input()` also do a route lookup to check the recipient of the packet. If the packet is intended for the current machine, `ip_local()` or `ip6_local()` will be called otherwise, the packet will be forwarded. After the IP packet is sanitized, it is passed to the `ip_deliver()` function.

Every IP package for the local machine will pass the `ip_deliver()` function[7]. This function passes the IP packet payload to the next protocol processing function depending on the protocol header. These functions either do defragmentation, decapsulation or layer-4 processing. `ip_deliver()` calls many other input functions and is a good entry point for verifying the input functions above it.

[7]It is based on the IPv6 input function by the WIDE Project which is used used for IPv6 header processing [16].

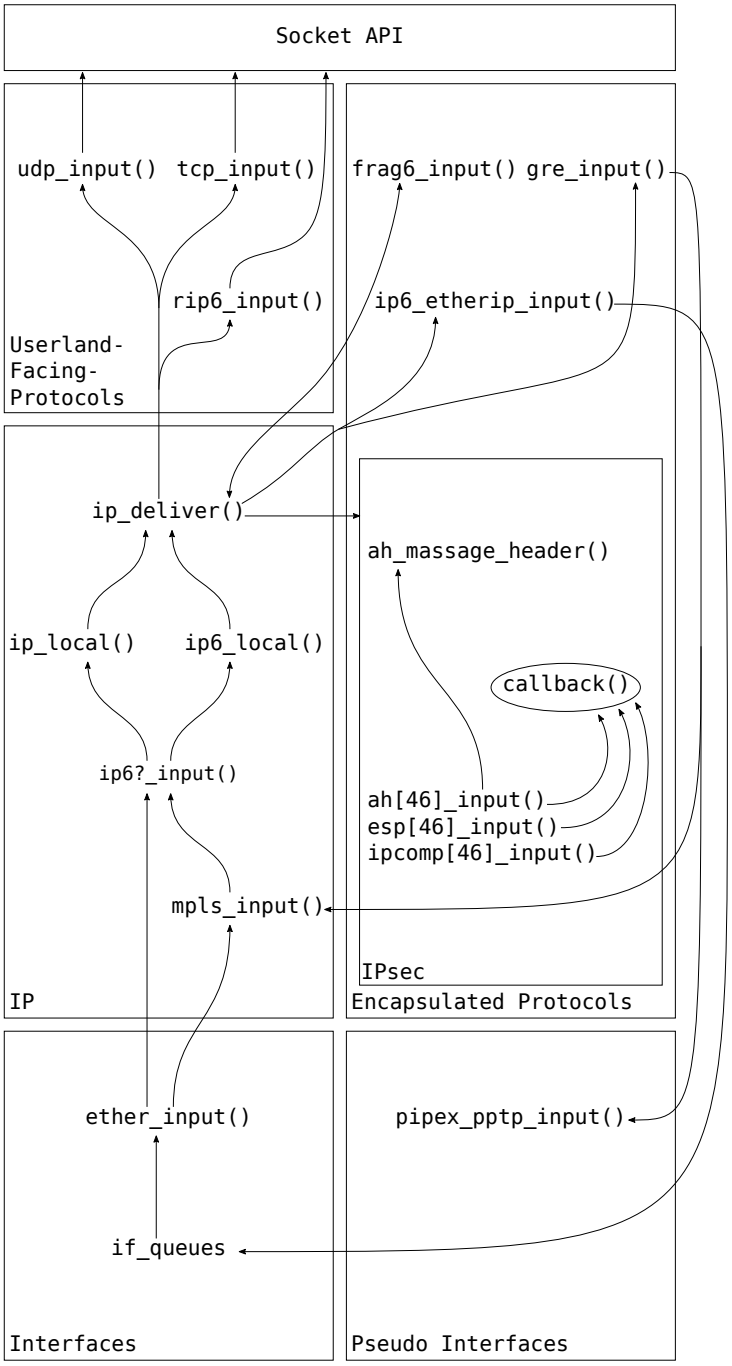


Fig. 5. Network input processing in OpenBSD

4 Considerations and Adjustments Before Verification

4.1 Features and Restrictions in the Kernel Environment

Applying formal methods on existing code bases comes with restrictions due to program complexity and missing specifications. Furthermore, network protocol RFC specifications are not precise enough and vary in implementations. The NetSem [7] project provides specifications for different protocols, but the protocols are not enough to specify a network stack. This complicates the process of specifying the behavoir of the OpenBSD network stack.

Another problem is the runtime environment that the OpenBSD network stack requires compared to many userland programs. The OpenBSD kernel holds a big global state that is accessed from many areas in the program. A solution to this is slicing, which shows promising results [1]. The code was adjusted manually instead to reduce the complexity.

It is important to be aware of the traits of the kernel environment, to select a strategy for verification. A good property of the kernel context is that floating point numbers are normally not used. A kernel has little address space to work with [22] and because of this, the network stack does not use recursion or the recursion is limited to a defined depth. Furthermore, a kernel is required to do low-level memory operations like pointer arithmetics with complicated data structures. Therefore it is essential to efficiently handle the memory operations in CPAchecker as the SMGCPA does.

Even though many changes to support parallelization were added to the network stack in recent years, the bugs from Table 1 are caused in a single threaded environment.

4.2 Required Adjustments to the OpenBSD Source Code

OpenBSD is based on a monolithic kernel — this means that all system services are implemented in the same kernel address space. Therefore there is a lot of state in the kernel context which is not needed in the network stack. To reduce this state, functionality is removed when the state is not relevant for the target function. An example of this is the memory management. The kernel provides multiple ways to allocate memory for optimizations like allocation time. Using CPAcheckers `malloc()` implementation instead provides the same functionality with less complexity. Another way to reduce com-

plexity in the OpenBSD kernel is conditional preprocessor directives that can remove modular features like IPv6, IPsec, and MPLS.

4.2.1 Preprocessing Network Modules

Since the support for programs consisting of multiple files is advertised as experimental in CPAchecker[8] and function prototype scope is not supported, it is easier to create a single file that contains all the C source code by concatenating all source files into a new file. This file is called a module and requires preprocessing with the OpenBSD kernel build infrastructure[9].

CPAchecker works around the lack of function prototype scope by prepending the file name to the function name. But this would lead to errors if the functions and their parameters were compared.

Checking the whole OpenBSD kernel in one run for errors is currently not possible due to the complexity of the program. To circumvent this problem, an artificial program entry point is used to start the verification attempt. The previous execution path that calls the erroneous function can be looked up in Fig. 5.

4.2.2 Simplifying Memory Management

API calls were simplified to reduce the number of complex pointer operations that often would use architecture specific assembler. This is the case for the memory pool API that is used in the kernel as another way to allocate memory. For the allocation of mbufs, the pool API is used.

In contrast to a userland program, the kernels can decide not to wait for memory allocation. If not enough resources are available, the kernel will either wait for enough memory or return NULL. Which method to use is indicated with the `M_NOWAIT` flag. This flag is used when mbuf is allocated. To reproduce this behavior in CPAchecker, the mbuf API was adjusted to check for this flag and use the `__VERIFIER_nondet_bool()` function. Listing 3 shows the function used with CPAchecker to model this behavior.

[8] Efffort to resolve this is available online at the previously mentioned location.

[9] Based on `https://github.com/bluhm/preproc`

```
1  struct mbuf *
2  m_get(int nowait, int type)
3  {
4          struct mbuf *m;
5
6          if (nowait == M_NOWAIT && __VERIFIER_nondet_bool()) {
7                  return (NULL);
8          }
9          m = malloc(sizeof(struct mbuf), 0, 0);
10
11         m->m_type = type;
12         m->m_next = NULL;
13         m->m_nextpkt = NULL;
14         m->m_data = m->m_dat;
15         m->m_flags = 0;
16
17         return (m);
18 }
```

Listing 3. The adjusted `m_get()` function to use with CPAchecker

4.2.3 Removing Inline Assembler in the Source Code

Since a kernel is supposed to provide unified access to system resources to multiple programs, certain architecture specific operations either cannot be expressed in the C language or result in a significant speedup when written directly in an assembler. Operations like locking, memory handling and managing CPU statistics are normally implemented in assembler. But CPAchecker doesn't support inline assembler and ignores the statements. This makes the result unsound and alters the execution path. Statements not needed to verify the network stack like the statistics were removed.

4.2.4 Problems with Abstractions and Precision

A big problem in static code analysis is false positive errors. Only one false positive error arose while verifying with the SMGCPA. Usually, those errors should be treatable with `__VERIFIER_assume()` statements but did not treat the property violation in this case. It was possible that CPAchecker wrongfully failed verification due to a NULL-pointer dereference.

Listing 4 shows the location where a false positive error occurs. `ip_protox` is a map from char to char, `inetsw` is an array of protocol definitions and `pr_input` is a constant function pointer. Dereferencing `psw` in line 7 wrong-

```
1
2 switch (af) {
3   case AF_INET:
4     psw = &inetsw[ip_protox[nxt]];
5     break;
6   }
7   nxt = (*psw->pr_input)(mp, offp, nxt, af); // deref. here
8   af = naf;
9 }
```

Listing 4. Parts of the `ip_deliver()` function and a pointer dereferece

fully lead to an error. This was circumvented by replacing the function pointer call with the actual function.

4.3 Patches to CPAchecker for Usage on Kernel Code

This section describes the adjustments that were necessary to evaluate OpenBSD source code with CPAchecker. The biggest problem was converting the OpenBSD source code to a CFA because of the complicated data structures in the OpenBSD kernel.

4.3.1 Simplification of Constant Binary Expressions

Abstractions for processor independence in the OpenBSD kernel are archived by writing architecture specific abstractions in C preprocessor macros. These abstractions can result in constant expressions like in Listing 5 and should be optimized by the compiler [20, §6.6].

In CPAchecker conditional expressions were not simplified and instead a temporary variable was introduced for the condition as in Fig. 6. This lead to side effects in the initializer of a global variable and therefore converting the code to a CFA was aborted. Adding the optimization to CPAchecker resolved this issue.

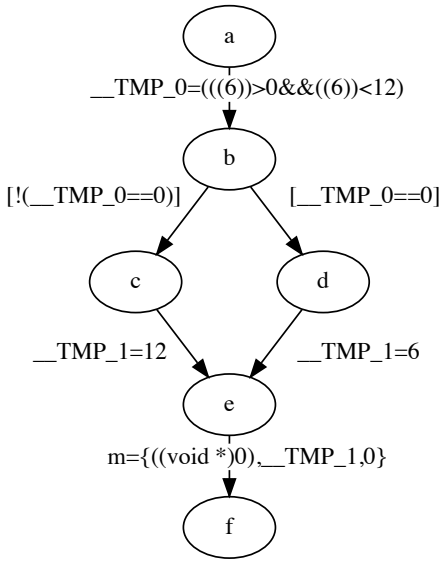


Fig. 6. Example program misrepresented as CFA

```
1   struct mutex m = { ((void *)0), ((((6)) > 0 && ((6)) < 12)? 12 :
        ((6))), 0};
```

Listing 5. A constant conditional expression in a variable initialization

```
1 extern u_char ip6_protox[];
2 u_char ip6_protox[256];
```

Listing 6. Multiple array declarations

4.3.2 Proper Handling of Duplicate Declarations

In the OpenBSD source code, structures are declared in a header file and additionally initialized once in another file. The preprocessed code then looks similar to Listing 6. Multiple declarations are supported by CPAchecker, but when an array was declared as extern, the size was marked as undefined after the first declaration. All following declarations with a length were interpreted as accessing the arrary. This is not according to the standard [20, §6.7.2.3].

4.3.3 Other Necessary Changes

The memory allocation functions in the OpenBSD kernel have more parameters than their POSIX userland equivalents. Removing the check for the parameter count in CPAcheckers `malloc()` implementation resolved this.

```
1  --- sys/net/if_etherip.c 6 Jun 2017 11:51:13 -0000 1.19
2  +++ sys/net/if_etherip.c 1 Feb 2018 19:19:14 -0000
3  @@ -583,7 +583,7 @@ ip6_etherip_input(struct mbuf **mp, int
4    if (!etherip_allow && (m->m_flags & (M_AUTH|M_CONF)) == 0) {
5      m_freem(m);
6      etheripstat.etherips_pdrops++;
7  -   return IPPROTO_NONE;
8  +   return IPPROTO_DONE;
9    }
10
11   ip6 = mtod(m, const struct ip6_hdr *);
```

Listing 7. Wrong return value in EtherIP

CPAchecker does not support extern variables in block scope. When encountering non-global variables with the extern storage class, CPAchecker aborts. To work around this issue, all extern variables were declared as global. This is possible since a compiler should error when encountering out of scope access of extern variables.

Structs containing anonymous structs can be referenced in multiple files by including them in a header file. But in CPAchecker each anonymous structs have a unique identifier and anonymous structs would never be recognized as equal because of this. Comparing the content of anonymous structs will resolve the problem. Since only one file was used instead the issue can be ignored. First steps for a patch are available online but the patch does not cover all possible cases.

5 Using CPAchecker on OpenBSD Network Code

5.1 EtherIP

Listing 7 was submitted on February 2nd, 2018. The unpatched code leads to memory corruption and could crash the kernel. If an Ethernet frame was encapsulated in an IPv6 packet and Ethernet over IP (EoIP) was disabled, memory would be freed but used again because of a wrong return value.

Listing 8 shows the artificial entry point that was created to detect this error. First the IP data structures are initialized with `ip_init()` and `ip6_init()`. Additionally, the mbuf `m` is allocated and then initialized with `m_gethdr()` and `m_tag_init()`. The length `len` of the packet and the

```
1  int main() {
2    struct mbuf *m;
3    int len, off;
4
5    ip_init();
6    ip6_init();
7    m = m_gethdr(0, MT_HEADER);
8    m_tag_init(m);
9
10   len = __VERIFIER_nondet_int();
11   __VERIFIER_assume(len > 0);
12   __VERIFIER_assume(len <= MHLEN);
13   off = __VERIFIER_nondet_int();
14   __VERIFIER_assume(off > 0);
15   __VERIFIER_assume(off <= len);
16
17   m->m_hdr.mh_len = m->M_dat.MH.MH_pkthdr.len = len;
18   ip_deliver(&m, &off, IPPROTO_IP, AF_INET6);
19 }
```

Listing 8. Required initialization to reach EtherIP

mbuf data are set to any value between 0 and the maximum data size of a single mbuf. The offset `off` needs to point somewhere into the data buffer and therefore can't be bigger than `len`. Then `m`, `off` and protocol information are passed to the `ip_deliver()` loop. There the packet should be identified as an EoIP packet and passed to `ip6_etherip_input()`. If EoIP is disabled, the mbuf `m` will be freed and a wrong value returned. The `ip_deliver()` loop will not stop because of the wrong value and therefore dereference `m` again.

With SMG it was possible to reliably find the use after free bug. No more bugs in `ip_etherip_input()` were found as soon as the patch was applied.

6 Evaluation

6.1 Running Environment

The verifications were evaluated on an Intel(R) Core(TM) i5-3570K CPU (3.4 GHz) with 8GB memory and a time limit of 15 minutes. An adjusted version of the memory-safety CPA from the SV-COMP18 was used.

Patch	Time	Max Memory	Result
007_etherip	314.712*s*	2329MB	False
007_etherip (patched)	313.433*s*	2577MB	True

Table 2: Verification results

6.2 Considered Software and Systems

CPAchecker `r29066` with additional patches[10] was used with the Java version `openjdk version "1.8.0_181"`, the runtime `OpenJDK Runtime Environment (build 1.8.0_181-8u181-b13-0ubuntu0.18.04.1-b13)` and the server `OpenJDK 64-Bit Server VM (build 25.181-b13, mixed mode)`. The tasks were run on a machine with the following configuration: `Linux 4.15.0-33-generic #36-Ubuntu SMP Wed Aug 15 16:00:05 UTC 2018 x86_64 x86_64 x86_64 GNU/Linux`. To build the modules, `OpenBSD 6.3 GENERIC#100 amd64` was used.

6.3 Results

The results are presented in the following table:

The bug is found in the unpatched version by CPAchecker. No further bugs are found in the module after applying the patch.

Reliably finding a bug after a lot of initial work is a success but more is necessary to actually use formal verification on software for productive operation. Since a solid foundation is established to find further error classes, it is safe to assume the use of CPAchecker in more areas of the OpenBSD kernel.

6.4 Discussion

Using SMG to ensure correct behavior in modules that manipulate memory is a solid approach. Especially when trying to find other low-level memory handling errors, there are no other promising approaches. But it is desirable to try different approaches like k-induction where applicable.

The results are few due to time constraints. Since every bug from Table 1 is unique, understanding and reproducing a bug takes a lot of time, even though the initial work of adjusting the data structures and ensuring the correct parsing of the source code is reusable.

[10] Available at `https://moritzbuhl.de/bachelor-thesis`

The OpenBSD source code uncovered errors in the CFA creation process which is surprising since the Linux kernel is using similar constructs in its source code.

6.5 Future Work

First steps for using CPAchecker on the OpenBSD code are taken but more testing is required and other known bugs should be reproduced before expanding the use. Only processing functions for incoming network packets were inspected, many more areas could be checked.

As effort is put into parallelization of the network stack, new opportunities for formal verification arise. For the OpenBSD kernel this would require effort on the scheduler and the memory locking mechanisms since it is implemented underneath the POSIX-layer.

6.6 Conclusion

Reproducing an error in the code required a lot of work. Especially the initial setup was a lot of work because the modules depend on each other without a strict separation of the API. Therefore function calls had to be adjusted for verifying single modules as more dependencies emerged. This lead to an iterative process of adjusting the source code, the environment, and the verifier.

The process surfaced another problem because the reporting functionality in CPAchecker is limited with growing input sizes. At a certain size, the CFAs are too big to see anything in them. Bigger dot files cannot be converted to a displayable program either because of memory limits or when a label has 2^{16} or more characters. The HTML report has similar problems with expressiveness at a certain size of the input.

To ensure fewer problems when applying formal verification with CPAchecker on software used in production environments, it is important to use CPAchecker on large software projects. Most problems encountered exist due to unexpected constructs in the C language. Some were added to ensure architecture independence, whereas others are well thought out constructs of engineering that in some form exist in any bigger project. Another required feature is to support multiple files in CPAchecker by removing the remaining problems with file scope and global scope.

References

[1] Pavel Andrianov et al. "CPA-BAM-Slicing: Block-Abstraction Memoization and Slicing with Region-Based Dependency Analysis - (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*. Ed. by Dirk Beyer and Marieke Huisman. Vol. 10806. Lecture Notes in Computer Science. Springer, 2018, pp. 427–431. ISBN: 978-3-319-89962-6.

[2] Dirk Beyer. "Software Verification and Verifiable Witnesses". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Christel Baier and Cesare Tinelli. Springer Berlin Heidelberg, 2015, pp. 401–416. ISBN: 978-3-662-46681-0.

[3] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. "Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis". In: *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. Ed. by Werner Damm and Holger Hermanns. Vol. 4590. Lecture Notes in Computer Science. Springer, 2007, pp. 504–518. ISBN: 978-3-540-73367-6.

[4] Dirk Beyer and M. Erkan Keremoglu. *CPAchecker: A Tool for Configurable Software Verification*. Tech. rep. SFU-CS-2009-02. School of Computing Science (CMPT), Simon Fraser University (SFU), Jan. 2009.

[5] Dirk Beyer and M. Erkan Keremoglu. "CPAchecker: A Tool for Configurable Software Verification". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 184–190. ISBN: 978-3-642-22109-5.

[6] Dirk Beyer et al. "Software model checking via large-block encoding". In: *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*. IEEE, 2009, pp. 25–32. ISBN: 978-1-4244-4966-8.

[7] Steve Bishop et al. “Engineering with Logic: Rigorous Test-Oracle Specification and Validation for TCP/IP and the Sockets API”. Nov. 2017. URL: https://www.cl.cam.ac.uk/~pes20/Netsem/paper3.pdf.

[8] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. Ed. by Richard Draves and Robbert van Renesse. USENIX Association, 2008, pp. 209–224. ISBN: 978-1-931971-65-2.

[9] Kamil Dudka, Petr Peringer, and Tomás Vojnar. *Byte-Precise Verification of Low-Level List Manipulation*. Tech. rep. FIT-TR-2012-04. Faculty of Information Technology, Brno University of Technology, Apr. 2013.

[10] Jun-ichiro itojun Hagino. “Mbuf Issues in 4.4BSD IPv6/IPSec Support: Experiences from KAME IPv6/IPSec Implementation”. In: *Proceedings of the Freenix Track: 2000 USENIX Annual Technical Conference*. June 2000, pp. 291–300.

[11] Lukás Holk et al. “Predator Shape Analysis Tool Suite”. In: *Hardware and Software: Verification and Testing - 12th International Haifa Verification Conference, HVC 2016, Haifa, Israel, November 14-17, 2016, Proceedings*. Ed. by Roderick Bloem and Eli Arbel. Vol. 10028. Lecture Notes in Computer Science. 2016, pp. 202–209. ISBN: 978-3-319-49051-9.

[12] Bart Jacobs et al. “VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java”. In: *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*. Ed. by Mihaela Gheorghiu Bobaru et al. Vol. 6617. Lecture Notes in Computer Science. Springer, 2011, pp. 41–55. ISBN: 978-3-642-20397-8.

[13] Claudio Jeker. “OpenBSD network stack internals”. In: *AsiaBSDCon-2008 Proceedings*. Feb. 2008.

[14] *Kernel Developer’s Manual MGET(9)*. OpenBSD-6.3. The OpenBSD project. 2018. URL: https://man.openbsd.org/OpenBSD-6.3/mbuf.

[15] Michal Kotoun et al. “Optimized PredatorHP and the SV-COMP Heap and Memory Safety Benchmark - (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Marsha Chechik and Jean-François Raskin. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 942–945. ISBN: 978-3-662-49673-2.

[16] Qing Li, Tatuya Jinmei, and Keiichi Shima. *IPv6 Advanced Protocols Implementation*. IPv6 Advanced Protocols Implementation. Elsevier/-Morgan Kaufmann Publishers, Apr. 2007, p. 656. ISBN: 9780123704795.

[17] Marshall Kirk McKusick et al. *The Design and Implementation of the 4.4BSD Operating System*. UNIX and Open Systems Series. Addison-Wesley Publishing Company, Aug. 1996. ISBN: 0201549794.

[18] Evgeny Novikov and Ilja Zakharov. “Verification of Operating System Monolithic Kernels without Extensions”. May 2018.

[19] Solal Pirelli, Arseniy Zaostrovnykh, and George Candea. “A Formally Verified NAT Stack”. In: *Proceedings of the 2018 Afternoon Workshop on Kernel Bypassing Networks, KBNets@SIGCOMM 2018, Budapest, Hungary, August 20, 2018*. ACM, 2018, pp. 8–14.

[20] *Programming Languages — C*. Standard. International Organization for Standardization, Dec. 1999.

[21] W. Richard Stevens. *TCP/IP Illustrated, Volume 3: TCP for Transactions HTTP, NNTP, and the UNIX Domain Protocols*. professional computing series. Addison-Wesley Publishing Company, Jan. 1996. ISBN: 0201634953.

[22] Andrew S. Tanenbaum. “Modern operating system”. In: Prentice Hall PTR, Feb. 2001. Chap. Static versus Dynamic Structures. ISBN: 0130313580.

[23] Ilja S. Zakharov et al. “Configurable toolset for static verification of operating systems kernel modules”. In: 41.1 (2015), pp. 49–64.

[24] Arseniy Zaostrovnykh et al. “A Formally Verified NAT”. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*. ACM, 2017, pp. 141–154. ISBN: 978-1-4503-4653-5.