Athena: Analyzing and Quantifying Side Channels of Transport Layer Protocols

Feiyang Yu, *Duke University;* Quan Zhou and Syed Rafiul Hussain, *Pennsylvania State University;* Danfeng Zhang, *Duke University*

https://www.usenix.org/conference/usenixsecurity24/presentation/yu-feiyang

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Athena: Analyzing and Quantifying Side Channels of Transport Layer Protocols

Feiyang Yu[1], Quan Zhou[2], Syed Rafiul Hussain[2], and Danfeng Zhang[1]

[1]Duke University
[2]Pennsylvania State University
[1]*{fy66, danfeng.zhang}@duke.edu*
[2]*{qfz5074, hussain1}@psu.edu*

Abstract

Recent research has shown a growing number of side-channel vulnerabilities in transport layer protocols, such as TCP and UDP. Those side channels can be exploited by adversaries to launch nefarious attacks. In this paper, we present Athena, an automated tool for detecting, quantifying and explaining side-channel vulnerabilities in vanilla implementations of transport layer protocols. Unlike prior tools, Athena adopts a novel graph-based analysis, making it scalable enough to be the first side-channel analysis tool that can comprehensively analyze the TCP and UDP implementations in several operating systems with significantly higher coverage than the state-of-the-art. Moreover, Athena uses an entropy-based algorithm to identify the most important vulnerabilities. Evaluation on several benchmarks including Linux, FreeBSD, OpenBSD and two open-source IPv4 implementations suggests that Athena can narrow down critical side channels to a single digit (among over 1000 candidates) with a low false positive rate. Besides covering known side channels, Athena also discovers 30 new potential attack surfaces.

1 Introduction

Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are the two most widely used transport layer protocols for reliable and low-latency data transmission between network hosts. However, recent research has shown growing concerns about the susceptibility of side-channel vulnerabilities in these protocols, which can be exploited by adversaries to launch nefarious attacks. Some typical forms of these attacks can expose sensitive information, such as port numbers [2, 10, 22, 23], the existence of an active client in the Internet, and even hijack connections between a server and a targeted client [17]. These side channels primarily lurk in the shared resources that are created and managed by the server's operating systems; the resources are shared across multiple connections established with different clients [6, 7, 10, 11, 13, 16, 19, 22, 23, 25].

Despite an increasing number of identified side channels in transport layer protocols, the majority of those side channels are manually identified, which require extensive domain knowledge and manual effort. Moreover, the manual analysis cannot guarantee full code coverage as there might still be unknown side channels lurking in TCP and UDP protocols, and a new kernel update might introduce new side channels. Recent work [7, 12] follows a more promising approach. They develop a concrete/abstract model of the stand-alone version of a TCP implementation and use a static model checking technique to *automatically* identify side channels in those models. For example, Cao et al. [7] were able to detect 10 side channels in their analysis. However, due to the limited scalability of model checking technique, they were unable to analyze the entire TCP implementation, and bounded model checking cannot guarantee full code coverage either. This resulted in an inadequate analysis of complex protocol behavior and interactions with other components/layers, rendering poor coverage and an imprecise model of the analyzed TCP implementation. Finally, existing approaches require further manual effort to diagnose the root cause of side channels, such as why the side channels exist, when they can be triggered and how they can be observed by an attacker. *This paper, therefore, aims to fill this gap by developing an automated approach for developers to systematically identify side-channel vulnerabilities in TCP and UDP implementations, as well as an automated diagnostic mechanism to locate their root causes.*

Prior work [7, 12] identifies side channels in transport protocols as a violation of a non-interference property: given two instances of the same server state where only security sensitive properties are different, whether sending a set of packets (inputs) to the two servers will result in different responses. However, identifying such non-interference violations directly via model checking is costly. For instance, SCENT [7] reports an exponentially growing running time with regard to the number of packets and its execution time reaches over 10,000 seconds with 4 incoming packets. Ensafi et al. [12] report extreme running time (over 3 days) and memory usage (>16GB) on an abstract model of implementation code. In contrast, our observation is that one can detect the same violations with a much lower cost and scalable manner in two stages: (1) identify all *tainted* branches whose outcomes are affected by sensitive properties, and (2) identify the sensitive branches that influence different responses, i.e., sinks (e.g., the existence or absence of protocol actions such as sending out a network packet). Hence, one can detect side channels by asking the following question: *is there any tainted branch (with respect to the sensitive properties) that leads to different sinks in a control flow graph (CFG)?* Nonetheless, reporting all tainted branches that might reach a sink is inadequate for at least two reasons. First, even with a precise static taint analysis, such branches are enormous in the entire implementation of transport layer protocols (empirically, we found between 572 to 1651 tainted branches in Linux UDP and TCP implementations). So a manual inspection of all tainted branches is infeasible. Second, tainted branches do not shed too much light on *why* there is a side channel and *how* to fix it.

Based on the above observations, we design and implement a static analysis-based systematic side-channel analysis tool called Athena. Athena first models side channels in the transport protocols as a *graph search* problem based on the abstraction of tainted control flow graph (τCFG). To tackle the challenges of reporting and diagnosing all tainted/sensitive branches, Athena introduces an entropy-based approach motivated by *entropy* in information theory [28] to localize the *most important* side channels, and also develop a set of rules to automatically *explain* why the side channels exist (e.g., by observing whether an ACK message is sent or not in Established state, an attacker can reveal the secret values used by TCP) and how the side channels can be observed by an attacker. Finally, to comprehensively identify side channels under our threat model in protocol implementations, Athena adopts an *iterative* "rank-and-replace" approach that mimics how side channels are mitigated in transport layer protocols and then identifies the remaining ones. Our approach greatly improves scalability and usability of prior protocol side-channel analyses [7, 12] as both static taint analysis and CFG construction can be directly applied to protocol implementations without modifying or abstracting the source code, and they both scale to a large code base. Moreover, entropy-based graph-search algorithm also enjoys a complexity that is linear to graph size. Hence, Athena is able to analyze *entire implementations* of transport protocols, while prior work only analyzes their abstract models or partial code due to scalability limit.

We implement Athena and evaluate it on several benchmarks of TCP and UDP implementations, including Linux 3.19, Linux 4.8, FreeBSD 13.2 and OpenBSD 7.4 IPv4 kernel code, as well as two open-source IPv4 implementations on GitHub [24, 31][1]. Athena reports 34 side channels in the TCP model and 8 side channels in the UDP model in total. Athena further extracts information from these flows and provides diagnostic information such as the most critical branches and states needed to trigger the side channel.

Contributions. In this paper, we make the following contributions.

- We model the detection of side-channel vulnerabilities of transport layer protocols as a graph-search problem. Based on the graph, we develop an entropy-based approach to rank side channels according to their importance. Moreover, we develop an iterative rank-and-replace loop to identify a comprehensive list of side channels, i.e., no more side channels exist after the reported ones are fixed;
- We design and implement Athena, an automated tool that detects, quantifies and explains side-channel vulnerabilities thoroughly in the entire implementations of transport layer protocols. Athena is open-sourced at `https://github.com/athena-paper/athena`;
- We evaluate Athena with the TCP and UDP implementations of Linux 3.19, Linux 4.8, FreeBSD 13.2, OpenBSD 7.4 and two open-source programs (microps [24], picotcp [31]). Athena reports 42 side-channel vulnerabilities, including 30 new side channels, 7 known ones and only 5 false positives.

2 Transport Layer Protocols

Transport layer protocols operate at the transport layer of the TCP/IP Internet protocol stack. They are responsible for establishing communication between applications running on different hosts. The two most commonly used protocols are TCP and UDP.

TCP. TCP (Transmission Control Protocol) is a connection-oriented protocol that requires a logical connection to be established between the two hosts through a three-way handshake before data is exchanged. It ensures reliable and ordered delivery of data packets between applications (e.g., SSH and FTP) on different hosts through various mechanisms, such as flow control, congestion control, and error recovery. TCP is, however, vulnerable to a variety of attacks since security was not the primary concern in the TCP design. Some typical attacks include SYN flooding attacks [21], TCP session hijacking attacks [17], and blind RST attacks [27]. Many such attacks involve guessing certain connection secrets, namely sequence (SEQ) numbers and acknowledge (ACK) numbers. Therefore, the inference of these numbers has become a vital part of TCP attacks [1, 6, 11, 12, 25].

UDP. Unlike TCP, UDP (User Datagram Protocol) is a connection-less protocol that does not require any handshake or logical connection between hosts. As such, it does not provide any guarantees for reliable delivery of packets or ordered delivery of data. This feature makes UDP faster and more efficient than TCP. Hence, it is commonly used in applications

[1]The most starred TCP implementations written in C on Github.

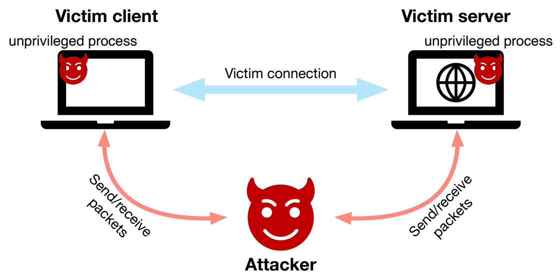


Figure 1: Threat model. An aided off-path attack also utilizes an unprivileged processes residing in the victim machines.

where real-time communication and low latency are critical. One of the most common applications that extensively use UDP is DNS (Domain Name System). DNS queries and responses are sent in UDP packets between clients and servers.

An important type of attack in systems using UDP protocols is the UDP port inference attack [22], as applications relying on UDP usually hide the client port to provide some level of integrity. Attackers use UDP port inference as a stepping stone to performing DNS cache poisoning attacks [22, 23, 29].

3 Overview

3.1 Threat Model

In line with prior work [15], we consider both a client and a server can be the victims of side-channel attacks. In this setting, we assume that the victim client and the victim server have already established TCP/UDP communication, and there is an off-path attacker (as shown in Figure 1). We also consider two different capabilities for such attackers: (a) *Unaided off-path attacker* (Adv_u), and (b) *Aided off-path attacker* (Adv_a). In our analysis, we distinguish the two types of threat models by different sets of sinks observable to an attacker.

Unaided off-path attacker (Adv_u). In this threat model, we assume an off-path attacker who can send TCP/UDP messages (either with their own IP address or spoofed ones with the target client's or server's IP address) to the victim client/server. However, being off-path, the attacker does not have the capability to eavesdrop on the active connection or inject/modify the packet transmitting between the client and the server (shown in Figure 1). Furthermore, the threat model assumes that attackers do not have any control over or aid from any unprivileged/privileged processes running on the victim client/server. Instead, attackers attempt to infer the system connection states only through the side channels discovered in protocol implementations directly or indirectly, such as global counters [6], IPID hash collisions [13] and SYN-backlogs [7]. To reflect this threat model in our analysis, we mark packet transmission functions as sinks, as only transmitted packets are observable by unaided off-path attackers.

Aided off-path attacker (Adv_a). In this threat model, we assume that Adv_a possesses the same capabilities as Adv_u and has additional control over or assistance from an unprivileged process running on the victim client/server. This attack model is consistent with the ones in prior work [2, 25, 26], wherein the unprivileged process is a sandboxed malicious script on the victim's machine injected by the attacker. Running on the application layer, the script cannot directly tamper with the data sent to/received from another host but can observe various networking parameters and statistics, or interact with the network stack on its behalf [2] and stealthily notify the Adv_u to help Adv_u launch attacks. For example, we assume an unprivileged process in one of the victim's machines can read `netstat` files (located in `/proc/net` which only requires normal access privilege) and stealthily report any change to Adv_a [2, 26]. To reflect this threat model in our analysis, we mark functions that modify local counters as sinks.

Attacker's goals. Attackers attempt to exploit side-channel vulnerabilities to infer the state of the victim's connection. Specifically, they would like to infer (a) the port number of the client victim used for the connection (for TCP and UDP), (b) the sequence (SEQ) number from the client (for TCP only), and/or (c) the acknowledge (ACK) number expected by the server (for TCP only). Such information is security sensitive for the following reasons. If an attacker successfully learns the port number, she can determine whether there exists an active connection between the client and server. If SEQ number is further inferred, she can trick the victims into prematurely closing the TCP connection by crafting an RST fragment (e.g., TCP Reset Attack [27]). If all three numbers are inferred, the attacker can arbitrarily inject their own data into the connection, and essentially hijack the connection (e.g., TCP Session Hijacking Attack [17]).

3.2 Illustrative Example

Figure 2 shows simplified code snippets in the Linux implementation of TCP; we use it to demonstrate a known side channel in the Linux implementation of TCP [26].

The first code snippet (from `tcp_rcv_established`) handles incoming packets when the socket is in the ESTABLISHED state. Here, the system first validates the incoming packet's SEQ and ACK number by matching them against the expected numbers (lines 2-5). If the header length is too short (line 6), the packet is discarded. Otherwise, it further validates the incoming packet in function `tcp_validate_incoming` (the second code snippet). Only when the sequence number is legit and RST, SYN flags satisfy required states (line 14-17), sink function `tcp_send_dupack` is being called; otherwise, the packet is discarded or the system calls a sink function `tcp_send_challenge_ack` to send out a SYN challenge. Similarly, the decision of sending out a SYN challenge or discard the packet between lines 24 and 29 also depends on the sequence number and the RST flag.

In this example, the sequence number of the host (i.e., the value of `tp->rcv_nxt)` is security sensitive. The sensitive

```
1   // In tcp_rcv_established(), tcp_input.c:
2   if ((tcp_flag_word(th) & TCP_HP_BITS) ==
3       tp->pred_flags &&
4       TCP_SKB_CB(skb)->seq == tp->rcv_nxt &&
5       !after(TCP_SKB_CB(skb)->ack_seq, tp->
            snd_nxt))
6     if (len <= tcp_header_len) {
7       goto discard;
8     }
9   ...
10  if (!tcp_validate_incoming(sk, skb, th, 1))
11    return;
12
13  // In tcp_validate_incoming(), tcp_input.c:
14  if (!tcp_sequence(tp, TCP_SKB_CB(skb)->seq,
15      TCP_SKB_CB(skb)->end_seq)) {
16    if (!th->rst) {
17      if (th->syn)
18        goto syn_challenge;
19      tcp_send_dupack(sk, skb);
20    }
21    goto discard;
22  }
23  ...
24  if (th->rst) {
25    if (TCP_SKB_CB(skb)->seq != tp->rcv_nxt) {
26      tcp_send_challenge_ack(sk);
27    }
28    goto discard;
29  }
```

Figure 2: A simplified code snippet of a side channel in TCP

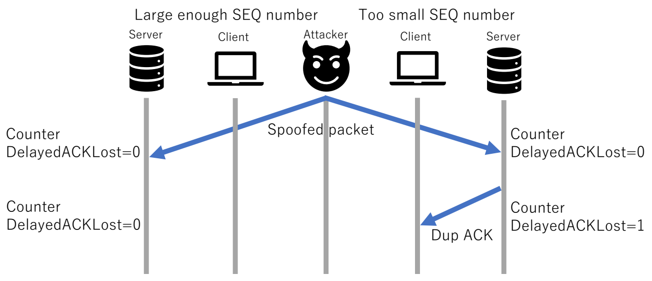
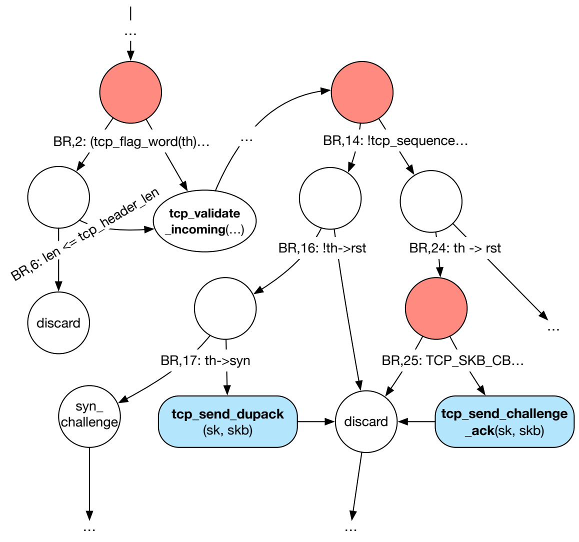


Figure 3: An illustration of the DUP ACK side-channel attack



BR,2: (tcp_flag_word(th)...
BR,14: !tcp_sequence...
tcp_validate _incoming(...)
BR,6: len <= tcp_header_len
discard
BR,16: !th->rst
BR,24: th -> rst
BR,17: th->syn
BR,25: TCP_SKB_CB...
syn_ challenge
tcp_send_dupack (sk, skb)
discard
tcp_send_challenge _ack(sk, skb)

Figure 4: A simplified tainted CFG of the TCP code snippets

data affects the outcomes of multiple branches in the code, including the ones at lines 4, 14 and 25. Hence, an attacker is able to reveal the sequence number of the host by observing which sink function is executed, or neither of them are called. For example, prior work [26] demonstrates how to reveal the sequence number via a side channel attack as illustrated in Figure 3. While an off-path attacker is unable to observe the existence/absence of a DUP ACK packet between client and server directly, an unprivileged process on the victim server can monitor `netstat` counters (located in `/proc/net/`). Depending on if the counter is incremented or not, the attacker can reveal if the sequence number of an incoming packet (crafted by the attacker) is smaller or greater than the sequence number of the host. In the source code, this vulnerability is reflected by the branch at lines 14-15, where sink function `tcp_send_dupack` is called only when the incoming sequence number is within a specific window.

Prior research [7, 12] recognizes side channels in transport layer protocols as a breach of the non-interference principle: when two hosts (e.g., the servers on the left and right respectively in Figure 3) possess identical states, except for sensitive security attributes, transmitting a particular set of packets to both hosts should not produce dissimilar responses. Moreover, they either use bounded model checking [7] on partial code or model checking on abstract model of implementation [11] to detect violations of the non-interference principle. While existing tools were able to unveil side channels in transport layer protocols, using model checking methods is both expensive and incomplete in terms of code coverage.

3.3 Why is Detecting Protocol Side Channels a Graph Search Problem?

In this paper, we recognize side channels in transport layer protocols (e.g., TCP and UDP) as a graph search problem. As shown in Section 3.2, different values of the host's sequence number `tp->rcv_nxt` lead the protocol executions to take different control flow paths. Since each distinct control flow path may eventually refer to disparate behavior of the system observable from the outside of that system (e.g., sending an DUP ACK packet, or sending a Challenge ACK packet, or discard the incoming packet), an off-path attacker can infer the sensitive status of the socket, thus leading to side-channel information leakage. In other words, when secret information, i.e., *sources* (e.g., `tp->rcv_nxt`), has any influence on the existence or absence of protocol actions such as sending out or dropping off a message, i.e., invocation of *sinks*, it signifies the existence of side channels in the transport protocols.

We leverage this intuition and develop a new graph-based approach that detects and ranks potential side channels in implementations of transport layer protocols. We call con-

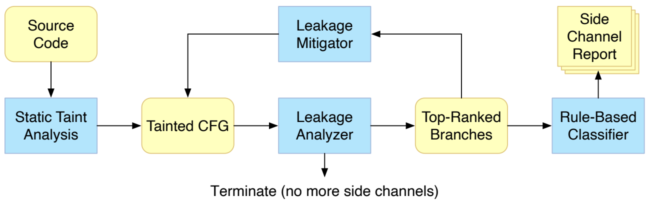


Figure 5: System overview

trol flow graph (CFG) with tainted branches and distinct sink nodes a *tainted CFG*. In essence, we aim to detect side channels by identifying tainted branches (with respect to the sensitive sources) that lead to *multiple* sinks (i.e., packet exchanges between the target system and the outside world) in the corresponding tainted CFG of the transport layer protocol implementation. For example, Figure 4 represents a simplified tainted CFG of the example code in Figure 2. In this graph, red nodes denote tainted branches (whose execution is dependent on the secret sources), and the sink functions are marked in blue. Note that the red node with label (`BR,14:!tcp_sequence...`) is the root cause of the side channel attack in Figure 3; depending on its outcome, the host can either reach sink function `tcp_send_dupack` or simply discard the packet (i.e., going through no sink function).

Why control-flows? Arguably, data-flow leakage where sources are directly leaked to sinks (e.g., directly sending out the sequence number to an off-path attacker) reveals more information than control-flow leakage. However, TCP and UDP implementations are unlikely to have any data-flow leakage due to their designs. We also confirmed this intuition via a static taint analysis, which detects no explicit data-flow leakage in Linux and FreeBSD implementations.

3.4 Workflow of Athena

In our approach, Athena takes tainted CFG (e.g., the one in Figure 4) as an input and reports a list of potential side channels that are both important and complete (i.e., if the reported ones are fixed, there are no remaining side channels). While the approach sounds intuitive, the main technical challenges are two-fold: (1) how to measure the importance of side channels, and (2) how to identify a *small* set of important sensitive branches that are also complete. For example, even in the simplified CFG in Figure 4, there are 3 distinct sensitive branches where each might cause a side channel. For the complete TCP implementation in Linux 3.19 IPv4 kernel, there are 1651 candidates of potential side channels!

To tackle the challenges, Athena follows a novel *rank-and-replace* loop to iteratively identify the most important side channels in each iteration, and then pursue the analysis as if they had been "replaced" with side-channel-free counterparts, until no more side channels are detected. Figure 5 shows an overview of Athena with three major components:

- **Leakage analyzer** (Section 4.3.1): we introduce an entropy-based metric to measure the importance of sensitive branches in a tainted CFG. For example, the red node with label (`BR,14:!tcp_sequence...`) has the highest importance as intuitively, it has direct impact on all possible outcomes: call `tcp_send_dupack`, call `tcp_send_challenge_ack`, or call no sink at all;

- **Leakage mitigator** (Section 4.3.2): inspired by how side channels in transport layer protocol implementations are mitigated in practice, we introduce a special node called *wildcard node* to mimic the effect of fixing a side channel by injecting noise. The revised taint CFG where all top-ranked branches are replaced by wildcard node is being analyzed in the next iteration;

- **Rule-based classifier** (Section 4.4): for each reported top-ranked branch in each iteration, Athena also utilizes the tainted CFG to automatically distill useful information such as the protocol states (e.g., ESTABLISHED) needed to trigger the side channel and the distinct outputs that an attacker can observe to reveal sensitive data. For the node with label (`BR,14:!tcp_sequence...`) in Figure 4, Athena reports that it can be triggered in ESTABLISHED state, and an attacker can observe whether a local counter is updated or not to reveal sensitive data.

With the innovations above, Athena reports 11 side channels for the full version of the tainted CFG of the TCP implementation in Linux 3.19, out of 1651 sensitive branches in the original tainted CFG. Moreover, the reported 11 side channels cover all previously reported ones that exist in the Linux code that we have analyzed. Among the 11 reports, 8 are new, only 1 is false positive, and 2 are previously reported (including the one with label (`BR,14:!tcp_sequence...`).

4 Design Details

4.1 Sources and Sink Functions

To analyze side channels in TCP and UDP, we first need to specify the following items for each protocol:

- Sources: secret data that a side channel might reveal,
- Sink functions: a set of functions of TCP and UDP implementations that an attacker might observe directly or indirectly if a function is called or not,
- Entry point: the starting point of execution. This is typically the function that receives network packages.

In this section, we use Linux implementation as an example to illustrate our design. Our design, however, is generic and can be applied to other operating systems with minimal extra effort to choose source/sink functions of similar functionality.

4.1.1 TCP

As discussed in Section 3.1, we consider the port number, SEQ number and ACK number to be the secret sources in a TCP implementation (called **3-tuple**). For example, Linux implementation stores the sources in the `tcp_sock` data structure, and therefore we mark the structure as the source.

A typical transport layer protocol implementation may have multiple entry points. For instance, the Linux TCP implementation has two different entry points for handling incoming packets in different states: (1) `tcp_rcv_state_process()` function, which handles all TCP states except *ESTABLISHED* and *TIME-WAIT* states, and (2) `tcp_rcv_established()` function, which handles packets in the *ESTABLISHED* state.

In the unaided threat model, an attacker may observe outgoing packets (sent by function `tcp_send_ack()` from the host). We mark a series of functions calling the send function as sinks. In the aided threat model, the execution of macro `NET_INC_STATS` (which increments a counter under `/proc/net` by 1) can be detected locally. Hence, the macro is also marked as a sink with the aided threat model.

4.1.2 UDP

The UDP protocol is more straightforward compared to TCP. There are two handlers in the Linux UDP implementation: (i) the `udp_rcv()` function for handling incoming UDP packets, and (ii) `udp_err()` for handling incoming error messages with embedded UDP packets. They are registered as the two entry points for UDP analysis. In both routines, the system checks the sensitive port number to find the existence of UDP connections (i.e., if there has been communication between the hosts) by looking up in a structure `udp_table`. Therefore, we label `udp_table` as the source in UDP protocol.

In the unaided threat model, an attacker may observe the invocation of `icmp_send()` and `update_or_create_fnhe()`. Hence, they are marked as sinks of UDP. In the aided threat model, the execution of macro `UDP_INC_STATS_BH` is marked as another sink, similarly as in TCP.

4.2 Tainted CFG

In this work, we use a *tainted* control flow graph, or simply τCFG, as the basis of side channel analysis. We first introduce τCFG and highlight why it is useful for identifying side channels in Transport Layer Protocol implementations.

A tainted control flow graph $\tau\text{CFG} = \langle V, E, T, S\rangle$ consists of a CFG $\langle V, E\rangle$, a set of tainted branching nodes T and a set of sink nodes S. As standard, a CFG $\langle V, E\rangle$ is a graphic representation of the control flow of a program. It is a directed graph where the nodes represent basic blocks of code and the edges represent the flow of control between the blocks. Each basic block represents a sequence of instructions that are executed without any jumps or branches, while the edges connect the basic blocks to show the flow of control between them. Conditional statements and loops are represented as nodes with multiple outgoing edges, each corresponding to a possible branch or jump target in the program.

Given a set of sources, say H, of a transport layer protocol implementation, a branching node is *tainted* if there are two initial host states m_1 and m_2 that only differ on the value of sources H (i.e., for any public variable $x \notin H$, we have $m_1(x) = m_2(x)$) such that control flows right after that branch are different when the protocol implementation is executed under m_1 and m_2, respectively. Note that with any sound static taint analysis, which identifies a set of variables whose value explicitly or implicitly depends on H, computing a set of tainted branching nodes T is straightforward: we collect the set of branching nodes whose branching condition uses at least one tainted variable.

Finally, S is a set of sink nodes where each node represents a call to a sink function (Section 4.1). For technical connivance, we assume that the sinks nodes set also contains a special node $\emptyset$ which represents the absence of any sink function (i.e., $\emptyset \in S$). Moreover, each node in CFG without an outgoing edge must be a sink node, including the absence node $\emptyset$. Note that there might be multiple incoming edges to sink nodes (including the absence node) in τCFG by definition.

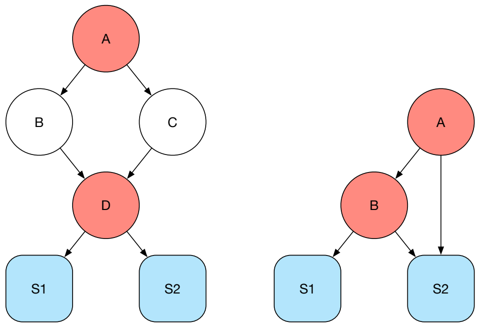
With τCFG, we define a *critical branch* as follows:

Definition 1 (Critical Branch)**.** *Given a tainted CFG $\langle V, E, T, S\rangle$, a branching node $v \in V$ is* critical *if it is both tainted (i.e., $v \in T$) and it can reach at least two distinct sink nodes in the corresponding CFG $\langle V, E\rangle$.*

We note that the critical branch serves as a proper graphic abstraction for analyzing side channels in implementations of Transport Layer Protocols since by definition, the absence of a critical branch in τCFG implies the absence of side channels. The reason is that if there is no critical branch, then for any two instances of the same host state where only security-sensitive properties are different, sending a set of packets (inputs) to the two hosts will always result in the same responses (i.e., sink function calls). Compared to directly analyzing if a transport layer protocol implementation can produce different responses when its host source values change [7, 12], constructing τCFG and computing critical branches in a τCFG is more scalable. τCFG further enables quantification and mitigation of side channels as we will elaborate in Section 4.3. One potential limitation of analyzing τCFG, however, is that a critical branch does not necessarily always cause a side channel. For example, a control-flow path in CFG might be infeasible in program execution (e.g., due to unreachable code). However, as we show in the evaluation (Section 6), our analysis built on τCFG is precise enough in practice.

4.3 Identifying Side Channels

Based on τCFG, one naive approach is to report all critical branches identified in a τCFG. However, the naive approach is



(a) (b)

Figure 6: Example τCFGs

problematic for a couple of reasons. First, a critical branch is not necessarily the *root cause* of a side channel. For example, consider the τCFG in Figure 6a where the tainted branches are in red and the sink nodes are marked as blue squares. Although the top node A is a critical node as it can reach both sinks S1 and S2, it makes no *impact* on which sink is reached. Intuitively, node D constitutes a side channel as depending on its branch outcome, either node S1 or S2 is reached (i.e., its branch outcome controls which sink node is reached).

Second, not all critical branches are equally important since they leak different amount of information, and moreover, they might be dependent on each other. For example, consider the τCFG in Figure 6b where the tainted branches are in red and the sink nodes are marked as blue squares. Intuitively, the red node B leaks more information than node A since by observing if S1 or S2 is reached, we can derive which branch is taken at node B. However, we can only do so for node A when S1 is reached. Moreover, if node B is replaced by always producing S2, then node A becomes non-critical since it can only reach sink S2 after the replacement. In other words, node A depends on node B in terms of criticalness.

Third, even with a precise static taint analysis, critical branches are enormous in the full implementation of transport protocols; for example, our analysis finds 185 critical branches in total in the TCP implementation of Linux 3.19.

4.3.1 Quantifying Critical Branches

To tackle the challenge of simply reporting all critical branches, we develop a novel quantification algorithm based on the concept of *entropy*, which is commonly used in information theory [28, 30]. Intuitively, for each node in τCFG, its entropy quantifies the *uncertainty* of reaching sink functions from that node. Moreover, the *change* of entropy before and after executing a branch (i.e., a branching node) measures the knowledge being learnt (by the attacker) via revealing the branch outcome.

In general, τCFG is not acyclic due to loops. However, to make entropy computation simple and efficient, we assume that τCFG is modified to be acyclic by removing backward edges (i.e., edges from the loop body to the loop condition) before further analysis. Removing the backward edges needs extra caution as it might remove a path from a tainted branch in the loop to an earlier call to a sink in the same loop. Athena avoids the issue by tainting the top node (loop head) whenever there is a tainted branch in the loop on backward edge removal. Hence, the tainted loop head will now reach all sinks in the loop and consequently detect potential side channels in a conservative way. Our evaluation results also suggest that the analysis on acyclic τCFG has high accuracy, as the top results match previously reported side channels [7] (Section 6.3).

Next, we provide a more formal and precise definition of node entropy in an acyclic tainted CFG, τCFG. For a node v in τCFG with sink nodes S, the entropy of v measures the uncertainty of which sink that v will eventually reach.

Definition 2 (Entropy of node). *Let $\tau CFG = (V,E,T,S)$ be an acyclic tainted CFG. For a node $v \in V$, let $\mathcal{H}_S(v)$ be the entropy of reaching the sink set S from v, defined as:*

$$\mathcal{H}_S(v) = \begin{cases} 0, & v \in S \\ -\sum_{s \in S} P(v,s)\log_2 P(v,s) & v \notin S \end{cases}$$

where $P(v,s)$ is the probability that node v reaches node s.

Note that the entropy of any sink node is 0, since it 100% reaches itself but nothing else. For any other node, $\mathcal{H}_S$ is defined as the Shannon entropy [28, 30] on the probability that node v reaches some sink node. For example, consider the τCFG in Figure 6b and assume that each outgoing edge has a 50% chance. Then, we have $\mathcal{H}_{\{S_1,S_2\}}(A) = -0.25\log_2(0.25) - 0.75\log_2(0.75) = 0.81$ and $\mathcal{H}_{\{S_1,S_2\}}(B) = -0.5\log_2(0.5) - 0.5\log_2(0.5) = 1$.

While entropy measures the absolute value of uncertainty, the *difference* of a node and its successor measures how much knowledge is learned by the fact that control flow transfers to the successor. Since a node might have multiple successors, we define the *leakage* of a node as the maximum entropy differences between itself and among its all successors.

Definition 3 (Leakage of node). *Let $\tau CFG = \langle V,E,T,S\rangle$ be an acyclic tainted CFG. For a node $v \in V$, let $succ(v)$ denote the set of the successors of v in τCFG. Let $\mathcal{L}(v)$ be the leakage of v defined as: $\mathcal{L}(v) = \max_{i \in succ(v)} \mathcal{H}_S(v) - \mathcal{H}_S(i)$.*

Returning to the τCFG in Figure 6b, given the assumption that each outgoing edge has a 50% chance, we have $\mathcal{L}(A) = 0.81 - 0 = 0.81$ and $\mathcal{L}(B) = 1 - 0 = 1$. Hence, node A is less critical than node B, which is consistent with our intuition since by observing if S1 or S2 is reached, we can derive which branch of B is taken. However, we can only do so for node A when S1 is reached.

Moreover, we note that the leakage definition also solves the challenge illustrated in Figure 6a. Assume that each outgoing edge has a 50% chance. It is easy to compute that the entropy of all nodes, i.e., A, B, C, and D is 1. Hence, the leakage of node A is 0, which correctly indicates that the node itself does not introduce any side channels.

Finally, we fill in an important missing piece in Definition 2, i.e., $P(v,s)$— the probability that node v reaches a sink node s. One subtlety is that for each branching node, we need to know the probability that it reaches each of its successors. For a tainted branch, we use a simplification assumption that each outgoing edge of the same node has the same probability[2]. For a public branch, we assume the worst-case scenario that an attacker can craft public inputs to direct to the more beneficial successor (i.e., the one with a larger entropy, as taking the branch might see a bigger leakage down the road). Hence, a public branch inherits the probability from the successor with maximum entropy.

Definition 4 (Probability of reaching a node)**.** *Let $\tau CFG = \langle V,E,T,S\rangle$ be an acyclic tainted CFG. For a node $v \in V$, let*

- *$succ(v)$ denote the set of the immediate successors of v in G*
- *m_v be the one with maximum entropy among $succ(v)$*

For two nodes u and v, let $P(u,v)$ be the probability that u reaches v, which is defined as:

$$P(u,v) = \begin{cases} 1, & u = v \\ 1/|succ(u)|, & v \in succ(u) \wedge u \in T \\ 0, & v \in succ(u) \neq m_v \wedge u \notin T \\ 1, & v \in succ(u) = m_v \wedge u \notin T \\ \sum_{i \in succ(u)} P(u,i)P(i,v), & v \notin succ(u) \end{cases}$$

Note that when $v \notin succ(u)$, u might still reach v indirectly. Hence, the last case inductively computes indirect reachability. Moreover, since the graph is acyclic, $P(u,v)$ for any two reachable nodes u,v, and further, the entropy of each node, can be computed efficiently by a topological sort and calculate the probability of reaching and entropy from the deepest nodes iteratively with dynamic programming. The computation complexity is just $O(|V|)$.

4.3.2 Ranking and Replacing Critical Branches

The previous approach only finds the most important critical branches. But fixing only the top-ranked critical branches is insufficient, as there might be other (less important) critical branches that are vulnerable to side channel attacks.

To identify all side channels with the given set of sources and sinks in a tainted CFG, Athena takes an iterative approach that mimics how side channels are mitigated in real implementations: Athena first reports all top-ranked critical branches in iteration 1, say A_1. Then, it marks those branches as "replaced" (by side-channel-free code)[3] and reruns the quantification algorithm on the τCFG with replaced nodes and reports all top-ranked critical branches in iteration 2, say A_2. Athena continues until all critical branches have zero leakage. So in the end, when Athena terminates after N iterations, all reported side channels are $A_1 \cup A_2 \cup \cdots \cup A_N$.

One challenge of the iterative approach is to model "replaced" nodes and incorporate them into the quantification algorithm (Section 4.3.1). To do so, we first observe that side channels of Transport Layer Protocols are typically mitigated by adding noise [6]. Compared to completely removing a side channel by unifying outcomes of a sensitive branch (e.g., ensuring that a packet is always sent whichever branch is taken), injecting noise is more suitable (though less secure) since implementations of Transport Layer Protocols must obey their protocol specifications (e.g., RFC 793).

For example, consider the code snippets in Figure 7 where the code on the top (from Linux 3.19) has a side channel at line 5. The reason is that variable `challenge_count` is tainted by sensitive information: the number of times that `++challenge_count` is executed reveals a sensitive host state (expected sequence number). Hence, by observing the absence of an acknowledgement packet when `challenge_count` reaches the limit, an attacker reveals sensitive host state. The code at bottom (from Linux 4.8) mitigates the side channel by resetting `challenge_count` to a random number instead of 0 at line 6 (we omit the details on the specific random number for simplicity). Therefore, observing the absence of an acknowledgement packet only reveals the fact that the random counter reaches 0 after it is decreased by

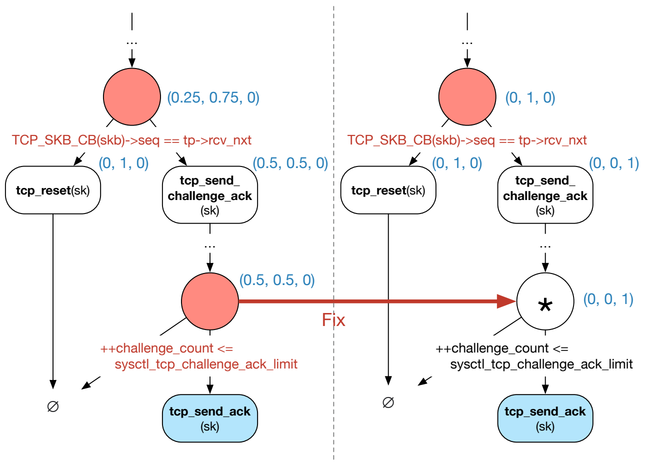
```
1  // vulnerable code in Linux 3.19
2  static unsigned int challenge_count;
3
4  ... reset challenge_count to 0 at fixed
       interval ..
5  if (++challenge_count <=
       sysctl_tcp_challenge_ack_limit) {
6    NET_INC_STATS_BH(sock_net(sk),
         LINUX_MIB_TCPCHALLENGEACK);
7    tcp_send_ack(sk);
8  }
```

```
1  // mitgated code in Linux 4.8
2  static unsigned int challenge_count;
3  struct tcp_sock *tp = tcp_sk(sk);
4  u32 count, now;
5
6  ..reset challenge_count to a random number at
       fixed interval..
7  count = READ_ONCE(challenge_count);
8  if (count > 0) {
9    WRITE_ONCE(challenge_count, count - 1);
10   NET_INC_STATS(sock_net(sk),
         LINUX_MIB_TCPCHALLENGEACK);
11   tcp_send_ack(sk);
12 }
```

Figure 7: Code snippet from `tcp_send_challenge_ack` function before and after a side channel is mitigated.

[2]We assume the uniform distribution for its simplicity and adequacy in practice (note that only the relative ranking, rather than the probabilities, matters in our context). While it is possible to use techniques like profiling to define a more precise model, we also note the concern that an adversary may draft inputs so that the system deviates from how it "usually" works - and thus invalidating the profiling data.

[3]Note that code replacement does not truly happen during our analysis, that is just an analysis construct.



(0.25, 0.75, 0)
TCP_SKB_CB(skb)->seq == tp->rcv_nxt
(0, 1, 0)
(0.5, 0.5, 0)
tcp_reset(sk)
tcp_send_challenge_ack (sk)
(0.5, 0.5, 0)
++challenge_count <= sysctl_tcp_challenge_ack_limit
∅
tcp_send_ack (sk)
Fix
(0, 1, 0)
TCP_SKB_CB(skb)->seq == tp->rcv_nxt
(0, 1, 0)
(0, 0, 1)
tcp_reset(sk)
tcp_send_challenge_ack (sk)
*
(0, 0, 1)
++challenge_count <= sysctl_tcp_challenge_ack_limit
∅
tcp_send_ack (sk)

Figure 8: Simplified τCFG before and after the top-ranked critical branch is replaced by wildcard node. The blue triple represents the probability of reaching sink node `tcp_send_ack`, absence node $\emptyset$, and wildcard node * respectively.

one in each execution of `tcp_send_challenge_ack`. Hence, it no longer reveals the sensitive host state. It is worth mentioning that such randomization-based mitigation are very common, and there has been similar mitigation in FreeBSD.

To model such replaced/fixed branches, we note that the replaced code neither calls sink function `tcp_send_ack(sk)` 100% of the time nor skips the call (i.e., reach absence node $\emptyset$) 100% of the time. Although there is uncertainty on which sink will be reached, the uncertainty is due to *randomness* rather than public or secret inputs. To reflect this observation, we introduce a special sink node called *wildcard node*, denoted by $*$ and replace each replaced node with a wildcard node. Most rules of Definition 4 remain the same with a wildcard node. The only change is that we redefine successors of a node to *exclude* nodes that surely reach wildcard node, i.e., $succ^*(v) = \{i \mid i \in succ(v) \wedge P(i,*) \neq 1\}$ and use $succ^*(v)$ instead of $succ(v)$ in Definition 4, as well as include an additional rule in case all successors are * nodes.

$$P(u,*) = 1 \text{ if } succ^*(u) \text{ is empty}$$
$$P(u,v) = 0 \text{ if } succ^*(u) \text{ is empty} \wedge v \neq *$$

The special rules of handling wildcard nodes are motivated by the randomness introduced to each replaced node, e.g., a uniform distribution on reaching all normal sink nodes by the injected noise. Therefore, its distribution does not contribute to its parent unless all of its siblings are also wildcard nodes.

Example. Consider a simplified τCFG of the vulnerable code in Linux 3.19 shown on the left of Figure 8. Based on the revised algorithm above that handles wildcard node, we can compute the probability of reaching each sink as the blue triple attached to each node. Since there is no wildcard node, the distributions are identical to the ones according to Definition 4. Since the lower red node has the maximum leakage of 1, Athena reports it and replaces it with a wildcard node as shown on the right of Figure 8.

In the revised graph, the wildcard node, say node A, has $P(A,*) = 1$ as it is a wildcard node (case 1 of Definition 4). Hence, the leakage of the wildcard node is 0 according to Definition 2. The interesting case is the leakage of the nodes above it. The parent of node A, say node B, has no successors according to the revised successor definition $succ^*$. Hence, $P(B,*) = 1$ and $P(B,v) = 0$ for other nodes. For the top red node, say node C, $succ^*(C)$ only contains its left child. Hence, by Definition 2 with $succ^*$ instead of $succ$, node C inherits the distribution from its left child, as shown in the figure, resulting in a leakage of 0. Therefore, there is no more side channels in the replaced τCFG on the right. The intuition behind why nodes that 100% reach wildcard node are excluded in the computation is that such nodes can *cast* to any distribution, such as the triple (0,1,0) on the left child of node C, meaning that the probability that reaching sink node `tcp_send_ack`, absence node $\emptyset$, and wildcard node * are 0%, 100% and 0% respectively. Hence, the combined distribution is the same as simply excluding the casted distributions.

4.4 Categorizing Side Channels

Next, we categorize the detected side channels based on (1) the packet handler function being called, and (2) the distinct reachable sink nodes, to gain a better understanding of the detected side channels. In particular, the TCP receive routine invokes one of the several packet handler functions w.r.t. the state of the current connection. For example, `tcp_rcv_established()` handles incoming packets when the connection is in the ESTABLISHED state. By tracking which function is called along the control-flow path, Athena outputs the TCP state required for triggering the side channel. For UDP, the results are categorized based on the initial packet handler function called (`udp_rcv()` or `udp_err()`).

To compute the distinct reachable sink nodes, recall that in the process of entropy calculation, we also keep track of the probability that a critical branch reaches one of the sinks (or does not reach the sinks at all). For example, if a branching block has 0.5 probability to reach `tcp_send_ack()`, and another 0.5 probability of reaching the absence sink, then an attacker learn sensitive data by observing two different outputs: Immediate ACK vs NULL.

5 Implementation

In this section, we briefly discuss the implementation details of the main components of Athena.

Static taint analysis. We implemented the static taint analysis toolchain in C++ within the LLVM framework. The toolchain is inherited from PIDGIN [18], which is built on top of a summary-based points-to analysis, DSA [20]. Our extension to the original implementation consists of ∼3K LOC. With context-sensitivity, we perform inter-procedural information flow analysis to find out the branches tainted by the sensitive sources. For soundness, both explicit and implicit flows are analyzed. The branches help us annotate the program's CFG, producing the τCFG discussed in Section 4.2. We also add an edge from each non-sink node without any outgoing edge to a special absence node $\emptyset$.

Leakage analyzer and leakage mitigator. The leakage analyzer and the leakage mitigator are written in Python together as a combined component. In a nutshell, this component reads the original τCFG or τCFG from the previous iteration and performs ranking and replacement according to the methods in Section 4.3.1 and Section 4.3.2. The leakage analyzer calculates each node's entropy and leakage in a bottom-up fashion through graph search. Once each node's entropy is computed, it reports the top-ranked branches with the highest leakage. These identified branches are then processed with the leakage mitigator to generate the τCFG with wildcard nodes. Athena iterates over these steps until there are no more side channels.

Rule-based classifier. After retrieving the finalized list of top-ranked branches from the previous steps, we use a rule-based classifier written in Python to categorize the nature of the side channels. The algorithm extracts control-flow paths from τCFG and examines their target source code, to check 1) which sinks will the critical branches lead to, and 2) what system state would be needed to trigger the critical branches.

6 Evaluation

We evaluate Athena with several implementations of TCP/UDP IPv4, including Linux 3.19 and 4.8[4], FreeBSD 13.2, OpenBSD 7.4[5], as well as two open-source implementations named microps [24] and picotcp [31].

- **Reduction**: Can entropy-based rank-and-replace approach significantly reduce the number of reported side channels?
- **Efficacy**: Can Athena detect all known side channels and uncover new side channels in the TCP and UDP implementations being analyzed?
- **Precision**: How many of side channels reported by Athena are false positives?
- **Performance**: Does Athena scale to the full implementations of TCP and UDP?

	# tainted branches	# critical branches	# reported branches
Linux/TCP (Adv$_u$)	1651	185	6
Linux/TCP (Adv$_a$)	1651	528	5
Linux/UDP (Adv$_u$)	572	59	3
Linux/UDP (Adv$_a$)	572	354	3
FreeBSD/TCP (Adv$_u$)	843	199	10
FreeBSD/UDP (Adv$_u$)	310	28	1
OpenBSD/TCP (Adv$_u$)	751	173	10
OpenBSD/UDP (Adv$_u$)	302	27	1
microps	204	35	2
picotcp	505	75	1

Table 1: Reported branches by various strategies.

- **Exploitablity**: For the detected side channels, how feasible is it to exploit them to launch real-world attacks?

6.1 Evaluation Setup

Sources and sinks of interest. As discussed in Section 4.1, we marked several sources and sinks in our analysis. In Linux, we mark the struct sock variable sk (*tcp_input.c: L1593*) as the source in the TCP protocol, tcp_send_ack() and tcp_send_delayed_ack() as sinks (for unaided attacker model Adv$_u$), tcp_send_challenge_ack() and tcp_send_dupack() as sinks (for aided attacker model Adv$_a$). For UDP, we mark the `struct udp_table` variable `udp_table` (*udp.c: L117*) as the source, `icmp_send()`, `ipv4_sk_update_pmtu()`, `ipv4_sk_redirect()`, `UDP_STATS_INC_BH` as sinks.

Sources and sinks in FreeBSD and OpenBSD are similar to those in Linux. We mark the struct tcpcb variable `tp` (*tcp_input.c: L611*) as the source, `tcp_output()` and `tcp_response()` as the sink. For UDP, we mark the struct inpcbinfo variable `pcbinfo` as a sink, and `icmp_error()` as a sink function. In BSD systems, the local counters are not exposed to unprivileged processes, and therefore the aided threat model Adv$_a$ is not applicable for FreeBSD and OpenBSD.

In the open source implementations, we mark the equivalent structures `tcp_pcb pcb` (microps) and `pico_tcp_hdr hdr` (picotcp) as sources, and the function `tcp_send()` as the sink.

6.2 Reduction

To evaluate the effectiveness of the iterative rank-and-replace approach of Athena, we compare it with two plausible solutions that report (1) all tainted branches, and (2) all critical branches (see Definition 1) to further demonstrate how much reduction (in terms of # reported sensitive branches) does Athena achieve. The result is summarized in Table 1.

In the Linux TCP module, there are 1651 tainted branches in both threat models. The total tainted branches are the same as the two settings only differ in their sink functions. By counting only the critical branches (i.e., the ones that at least reach two distinct sinks, including the absence sink), the number

[4]Due to the limitation of the points-to analysis DSA used by PIDGIN, the latest Linux version we can analyze by a compatible LLVM is 4.8. We present the results of 3.19 and 4.8 in this section and discuss the number of vulnerabilities remaining in the latest Linux kernel in Section 6.6.

[5]The latest product release.

of suspicious branches drops to 185 (88.7% drop) and 528 (68.0% drop) in the Adv_u and Adv_a settings, respectively. Finally, Athena only reports 6 (99.6% drop) and 5 (99.7% drop) most important branches responsible for all side channels in the analyzed code. In the Linux UDP module, the reduction rate is also significant. We see reduction rates of 88.8% (under Adv_u setting) and 32.7% (Adv_a) when using critical branches, and reduction rates of 99.3% (Adv_u) and 99.4% (Adv_a) by counting only the most critical ones.

In the FreeBSD modules, Athena shows a similar reduction rate (recall that only the unaided threat model is applicable for FreeBSD). There are 843 tainted branches in TCP, which are reduced to 199 (76.3% drop) and finally, 10 are reported (98.8% drop). In UDP, 310 tainted branches are reduced to 28 (91.0% drop) and finally only 1 is reported (99.7% drop).

Similarly, 751 (resp. 302) tainted branches are reduced to 173 (resp. 27) in OpenBSD TCP (resp. OpenBSD UDP) – for a 77.0% drop (resp. 91.1% drop). On the open source implementations, Athena has reduction rates of 82.8% (204 to 35, microps) and 85.1% (505 to 75, picotcp) in terms of critical branches. By counting only the most critical ones, the reduction rates are 99.0% (204 to 2, microps) and 99.8% (505 to 1, picotcp).

It is worth noting that the non-reported critical branches are either removed while replacing the reported ones, or reported and replaced in later iterations. Since our algorithm only terminates when there are no more reported critical branches left, it soundly detects all side channels per our threat model. We further discuss the precision of Athena in Section 6.4.

6.3 Efficacy

Next, we evaluate Athena's efficacy of identifying side channels by answering whether Athena can (1) detect existing side channels (under the same threat model as ours) that were *manually identified* by prior work [6, 22, 23, 26]; and (2) uncover new ones. Table 2 and 3 show the side channels in all benchmarks, reported by prior work under the same threat models (as described in Section 3.1, marked in yellow). Athena successfully reported all side channels in those works. We note that [7] reports 6 more side channels (entries 6-C, 7-C, 8-C and 9-B, 10-B, 11-B in the Table 3 of [7]), which were not identified by Athena. We manually verified that these reported branches do not exist in the Linux kernel code base that we have analyzed. We also searched for them in the GitHub repository associated with the paper[6], but we still did not find those branches or similar branches in the repository.

Other than those 6 entries that we confirmed to be absent in the Linux code base, Athena fully covers all other previously reported side channels [6, 7, 22, 23, 25, 26] in Linux and FreeBSD; it also uncovers new side channels because of more complete source code coverage. We further analyze those new side channels in Section 6.6.

[6]https://github.com/seclab-ucr/SCENT.

6.4 Precision

Next, we evaluate the precision of Athena's reported side channels. For this, we count the number of branches which are true positives (i.e., their branch outcomes depend on the sources) among the reported most critical sensitive branches.

As shown in Table 2 and Table 3, Athena reports 42 side channels in 5 transport layer protocol implementations. After manual investigation, we found that 5 out of the 42 side channels are false positives (the red entries in both tables) due to the imprecision of our static taint analysis. The root cause of the false positives is originated from the underlying points-to analysis used by Athena. In some cases, the points-to analysis falsely links data structures of inconsistent types to the same alias, resulting in a *field-insensitive* taint of certain objects. For example, entry 2 in Table 2 reports a branch `if (th->ack)`, which checks the `ack` flag of an incoming packet's header. Although the structure `th` is derived from a non-sensitive field in a `sock` structure, it is tainted as the points-to analysis is unable to differentiate sensitive fields from non-sensitive fields in the `sock` structure. We manually confirmed that all 5 false positives have the same root cause in the points-to analysis.

The remaining side channels are all true positives, which include 7 that were reported in prior works, and 30 remaining ones newly identified by Athena.

6.5 Performance

Running time.. On a commodity desktop computer with a 3.8GHz 8-core CPU and 32GB of RAM, Athena takes 12 minutes (7 minutes on static taint analysis and 5 minutes on graph search) and 5 minutes (2 minutes on static taint analysis and 3 minutes on graph search) for the Linux TCP and UDP modules. For the FreeBSD's and OpenBSD's TCP and UDP modules, it takes 9 minutes (6 minutes on static taint analysis and 3 minutes on graph searching) and 3 minutes (2 minutes on static taint analysis and 1 minute on graph searching) respectively. The analysis takes less than 1 minute on the open-source implementations microps [24] and picotcp [31].

Code coverage.. Athena covers 15,020 and 7,436 LoC of Linux v3.19 TCP/UDP implementations, 11,507 and 4,120 LoC of FreeBSD 13.2 TCP/UDP implementations, 10,068 and 2,503 Loc of OpenBSD 7.4 TCP/UDP implementations, 3,498 LoC of microps and 5,678 LoC of picotcp, respectively. We did not analyze the whole TCP module of Linux and BSDs since the rest of the code is unreachable from the input/receive functions, the entry functions of our analysis.

6.6 Exploitability of Reported Side Channels

Tables 2 and 3 show the violations that Athena found in the TCP and UDP implementations. Athena discovered 37 true positives in total. Based on the criteria we described in

Index	System	Critical branch	Iteration	Triggering state	Different outputs
1	Linux (Adv$_u$)	`if (((tp->rcv_nxt - tp->rcv_wup) > inet_csk(sk)->icsk_ack.rcv_mss && __tcp_select_window(sk) >= tp->rcv_wnd) || tcp_in_quickack_mode(sk) || (ofo_possible && skb_peek(&tp->out_of_order_queue)))`	1	Any non-closing	Immediate ACK vs Delayed ACK
2		`if (th->ack)`	2	SYN-SENT	ACK pkt vs NULL
3		`if (TCP_SKB_CB(skb)->end_seq != TCP_SKB_CB(skb)->seq && before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt))`	2	ESTABLISHED	ACK pkt vs NULL
4		`if (++challenge_count <= sysctl_tcp_challenge_ack_limit)`	3	Any non-closing	ACK pkt vs NULL
5		`case TCP_FIN_WAIT1`	3	ESTABLISHED	ACK pkt vs NULL
6		`case TCP_FIN_WAIT2`	3	ESTABLISHED	ACK pkt vs NULL
7	Linux (Adv$_a$)	`if (!tcp_sequence(tp, TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq))`	1	ESTABLISHED	Local counter update vs NULL
8		`if (TCP_SKB_CB(skb)-end_seq != TCP_SKB_CB(skb)->seq before(TCP_SKB_CB(skb)->seq, tp-rcv_nxt))`	1	ESTABLISHED	Local counter update vs NULL
9		`if (++challenge_count <= sysctl_tcp_challenge_ack_limit)`	1	ESTABLISHED	Local counter update vs NULL
10		`if (before(ack, prior_snd_una - tp->max_window))`	1	ESTABLISHED	Local counter update vs NULL
11		`if (!tcp_validate_incoming(sk, skb, th, 1))`	1	ESTABLISHED	Local counter update vs NULL
12	FreeBSD (Adv$_u$)	`if (needoutput || (tp->t_flags & TF_ACKNOW))`	1	Any non-closing	TCP pkt vs NULL
13		`if (th->th_ack != tp->snd_una)`	1	LAST-ACK	TCP pkt vs NULL
14		`if (DELAY_ACK(tp, tlen))`	1	Any non-closing	TCP pkt vs NULL
15		`if (tp->snd_una == tp->snd_max)`	1	Any non-closing	TCP pkt vs NULL
16		`if (avail > 0 || tp->t_flags & TF_ACKNOW)`	1	LAST-ACK	TCP pkt vs NULL
17		`if (SEQ_GT(onxt, tp->snd_nxt))`	1	LAST-ACK	TCP pkt vs NULL
18		`if (tp->t_state == TCPS_SYN_RECEIVED && (thflags & TH_ACK) && (SEQ_GT(tp->snd_una, th->th_ack) || SEQ_GT(th->th_ack, tp->snd_max)) )`	1	SYN-RCVD	ACK pkt vs NULL
19		`if (SEQ_LT(th->th_ack, tp->snd_recover))`	2	Any non-closing	TCP pkt vs NULL
20		`if (pps > 0)`	1	SYN-RCVD	TCP pkt vs NULL
21		`if (V_tcp_insecure_rst || tp->last_ack_sent == th->th_seq)`	1	Except LISTEN/ SYS-SENT	ACK pkt vs NULL
22	OpenBSD (Adv$_u$)	`if (so->so_snd.sb_cc || tp->t_flags & TF_NEEDOUTPUT)`	1	Any non-closing	TCP pkt vs NULL
23		`if (tp->t_flags (TF_ACKNOW|TF_NEEDOUTPUT))`	1	Any non-closing	TCP pkt vs NULL
24		`if (SEQ_LT(th->th_ack, tp->snd_last))`	1	Any non-closing	TCP pkt vs NULL
25		`if (tp->t_flags (TF_ACKNOW|TF_NEEDOUTPUT))`	1	FIN	TCP pkt vs NULL
26		`if (SEQ_LT(th->th_ack, tp->snd_last))`	1	LAST-ACK	TCP pkt vs NULL
27		`case TCPS_TIME_WAIT`	2	TIME-WAIT	TCP pkt vs NULL
28		`case TCPS_LAST_ACK`	2	LAST-ACK	TCP pkt vs NULL
29		`if (tp->t_flags & TF_ACKNOW)`	3	SYN-RCVD	TCP pkt vs NULL
30		`if (tp->rcv_wnd == 0 && th->th_seq == tp->rcv_nxt)`	1	ESTABLISHED	TCP pkt vs NULL
31		`if (opti.ts_present && (tiflags & TH_RST) == 0 && tp->ts_recent && TSTMP_LT(opti.ts_val, tp->ts_recent))`	1	Except LISTEN/ SYN-SENT	TCP pkt vs NULL
32	microps	`if (seg->ack <= pcb->iss || seg->ack > pcb->snd.nxt)`	1	SYN-SENT	RST pkt vs NULL
33		`if (!acceptable)`	1	TIME-WAIT	ACK pkt vs NULL
34	picotcp	`if ((t->cwnd >= t->in_flight) && (t->snd_nxt > t->snd_last_out))`	1	Any non-closing	RST pkt vs NULL

Table 2: Reported Branches on TCP. **No color**: new positives, **Yellow**: previously reported positives, **Red**: False positives

Index	System	Critical branch	Iteration	Different outputs
1	Linux ($\mathsf{Adv_u}$)	`if (fib_lookup(dev_net(dst->dev), fl4, res) == 0)`	1	PMTU update vs NULL
2		`if (fib_lookup(net, fl4, res) == 0)`	2	Gateway update vs NULL
3		`if (sk != NULL)`	3	ICMP pkt vs NULL
4	Linux ($\mathsf{Adv_a}$)	`if (count)`	4	Local counter update vs NULL
5		`if (count)`	5	Local counter update vs NULL
6		`if (sk != NULL)`	6	Local counter update vs NULL
7	FreeBSD ($\mathsf{Adv_u}$)	`if (rc == -ENOMEM)`	7	ICMP pkt vs NULL
8	OpenBSD ($\mathsf{Adv_u}$)	`if (inp == NULL)`	8	ICMP pkt vs NULL

Table 3: Reported Branches on UDP. **No color**: new positives, **Yellow**: previously reported positives, **Red**: False positives

Section 4.4, the reported positives can be categorized into 3 classes, which shed light on their exploitability.

Exploitable under unaided off-path attacker model.: Side channels 4, 20 in Table 2 and side channels 3, 7, 8 in Table 3 are caused by certain global rate limits on outgoing packets. In Linux, there are two frequently used global rate limits: `sysctl_tcp_challenge_ack_limit` and `sysctl_icmp_msgs_burst` (`V_icmp_rates` is the equivalent in BSD systems). Prior work has exploited these limits for remote attackers to infer whether a port is in use [22], or whether a guessed ACK number is correct [6] by sending spoofed packets and observe the response from the victim system. Side channel 8 in Table 3 is newly discovered by Athena. It is very similar to the reported ones in Linux (side channel 3) and FreeBSD (side channel 7), and it can be exploited using a similar attack. For the previously reported TCP side channels in Linux and FreeBSD (side channels 4 and 20 respectively in Table 2), no equivalent side channel exists in OpenBSD.

Side channels 1 and 2 in Table 3 involve changes in the routing table, specifically the Path MTU and gateway address for certain hosts. Both the global rate limits aforementioned and the routing table affect whether an outgoing packet (TCP or ICMP) would be sent. Prior work has shown a remote two-stage attack that infers UDP port number without observing the direct responses from the victim system [23]. No equivalent side channels are found in FreeBSD and OpenBSD.

Exploitable under aided off-path attacker model.: Side channels 7-11 in Table 2 and 5 in Table 3 are induced by the sink `netstat` counter that Linux maintains for diagnostic and statistical propose. When a certain operation is executed, the system increments one of these non-decreasing counters, and moreover, the value of the counter is shared globally across all users and processes on the host machine. Furthermore, the file that contains the `netstat` counters (under `/proc/net`) is set to be readable by all users (i.e., has the `-r--r--r--` permission on Linux). This allows even an unprivileged malicious process (similar to the one in the aided off-path attacker model Adv_a described in Section 3.1) to be able to observe if such an operation is performed. All the side channels in this category can be exploited by constantly checking the corresponding local counter. A proof-of-concept attack covering all such side-channels is described in Section 6.7. Since BSD systems do not maintain a publicly visible set of statistical counters, the equivalent side channels are not found.

Uncertain exploitability.: It is uncertain whether the remaining side channels can lead to exploitable in practice, but most of them do exhibit possible vulnerabilities. Side channels 1, 14, 23, 25, and 29 in Table 2 are associated with a TCP socket-specific option *quick_ack*, which controls whether to send a response ACK packet immediately. If an adversary is able to measure the response time of certain crafted packets and filter out network noise, they may reveal SEQ numbers.

Side channels 5, 6, 13, 16, 17, 27, 28 and 33 in Table 2 involve branches that trigger a system state transition into a closing state. Since the transition from an active state into a closing state in TCP is one-way, an attacker cannot learn information about the active connections. However, if the attacker can also control system-level processes that can prevent the connection from actually closing (i.e., staying in the state for a long time), they may exploit these branches. We leave a more thorough study of these side channels as future work.

Results on different Linux versions.: Athena analyzed both Linux 3.19 and 4.8, and the results of the latter are mostly the same as 3.19, except for a minor branch condition change and randomness-based mitigation for side channel 4 in Table 2.

In addition, we manually checked the source code of the most recent Linux kernel (v6.4.0, as of July 2023) to verify the number of the vulnerabilities reported by Athena are still present in the latest version of the Linux kernel. We found that 12 out of 14 vulnerabilities reported by Athena are still present in the latest version. The exceptions are side channels 1 and 3 in Table 3. The reason is that the code in the latest version has been rewritten and the problematic branches are removed. It is also worth mentioning that the critical branch of side channel 2 in Table 3 has slightly different conditions in the latest version. However, the side channel still persists.

6.7 A Sample Proof-of-Concept Attack

In this section, we show a Proof-of-Concept (PoC) attack based on side channel 9 in Table 2. The attack flow is shown in Figure 9 where we have a victim server S, a victim client

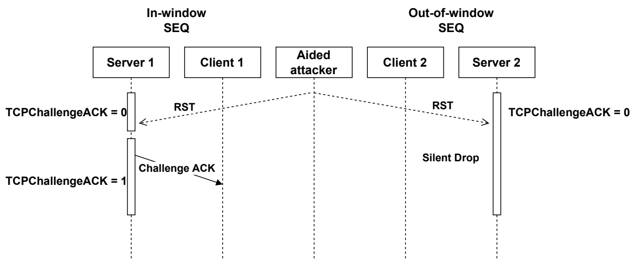


Figure 9: The flow of the PoC attack

C, and an attacker A. In line with previous work [26], we assume that the victim server and victim client have an active TCP connection (in ESTABLISHED state) and Attacker A controls an unprivileged process on S.

Attack Steps.: (1) The attacker A reads the file `/net/proc/netstat` (readable to all processes) on S through the controlled process. The attacker thus monitors the value of the field TCPChallengeACK. (2) The attacker then sends a spoofed RST packet to the server S, with client C's IP address, correct port numbers (the port used for the connection), and a guessed sequence number. (3) The attacker checks whether the monitored value of TCPChallengeACK has increased. If so, she can infer that the previously guessed number is in-window. Otherwise, the guessed number is far from the true sequence number. (4) If the guessed number is not in-window, the attacker repeats step 2.

Experimental results.: We have implemented and tested this PoC attack on a Debian 10-based machine with 100 attempts. In each attempt, the attacker starts with a random guess and then adjusts the guess depending on the outcome of each step. The PoC attack succeeds 69 out of 100 attempts where the failures are due to time-outs when the guessed sequence numbers are all far from the true sequence number. It takes an average of 4.2 seconds when the attack succeeds.

7 Related Work

Side-channel detection.. The most related works are SCENT [7] and the work by Ensafi et al. [12] as they also target side channels in transport layer protocols. They are both built on model checking, a form of static analysis that is likely to be more precise than static taint analysis and CFG-based analysis. For instance, model checking is immune to unreachable code and infeasible execution paths in CFG. However, the very precision also comes with a cost as they are unable to analyze the full vanilla implementations of TCP and UDP and suffer from long running time due to scalability limitation of model checking and inherent coverage limitation of bounded model checking. Moreover, they are unable to diagnose the root cause of the side channels, such as why the side channels exist, when they can be triggered, and how they can be observed by an attacker. In contrast, Athena trades precision for efficiency and full code coverage. Moreover, it automatically ranks side channels according to their importance and provides an informative report on each detected side channel.

Detecting secret-dependent branches is a common practice in identifying *timing* side channels in programs. For example, cryptographic libraries follow *constant-time programming paradigm* [3] to rule out timing side channels. Essentially, the constant-time programming paradigm prohibits both control-flow paths and memory-access patterns from depending on program secrets. Hence, information flow analysis can be used to detect both branch conditions (the same as tainted branches in this paper) and memory access addresses that are tainted by secrets [5, 32, 34]. However, as shown in Table 1, simply reporting tainted branches is infeasible for detecting side channels of transport layer protocols; Athena successfully reduces the reported tainted branches from thousands to a single digit number.

TCP side-channel attacks.. TCP side channels have been widely studied in the literature. Researchers have shown that various globally shared resources could be exploited as side channels to reveal secrets in TCP connections, such as IPID [13, 14], challenge ACK limit [6, 7], SYN-backlog [7] and local netstat counters [26]. The leakage of such connection secrets may grant attackers the capability to learn information about, or even manipulate, the victim's connection. Prior work has also shown that an attacker may utilize these side channels to scan idle ports [12], infer the existence of connection [6], count packets in transmission [19], measure round-trip-time [1, 33], detect intentional packet drop [11], and even hijack the connection [6, 9, 16, 25].

UDP side-channel attacks.. UDP side-channel attacks mainly target application-layer programs, particularly the Domain Name System (DNS). Several studies have exposed various vulnerabilities due to side channels in UDP, such as the works by Man et al. [22, 23]. SADDNS [22] discovered that the global ICMP rate limit, which was implemented to minimize bandwidth usage, forwarding costs, and mitigate the possibility of ICMP flooding attacks [4, 8], may be exploited by malicious attackers to infer the port numbers utilized by UDP-based services. With the revealed port number, an attacker could launch more powerful attacks, such as a DNS cache poisoning attack. In addition, another study by Man et al. [23] demonstrated yet another ICMP-based side channel that permits attackers to infer the UDP port number by forging ICMP error messages with an embedded UDP header, which enables the attacker to alter the host's routing table.

8 Discussion and Future Work

Limitations of the static analysis. There are several approximations in the static analysis of our approach. We discuss the impact of each next.

- Field-sensitivity issues in the current version of the points-to analysis have negative impacts on precision (exemplified in Section 6.4). While the empirical study shows that the resulting implementation produces a low false-positive rate for TCP/UDP code we have analyzed, this might not be the case for other implementations. For example, the use of complex data structures and union types usually downgrade analysis precision as they are challenging for sound points-to analyses. So Athena might have more false positives when the program has frequent use of those features.

- For the soundness of our analysis, Athena taints the loop head whenever there is a tainted branch in the loop on backward edge removal (Section 4.3.1). However, doing so may over-taint call paths, which leads to an over-approximation of critical branches and hence, imprecise branch possibilities/entropy computation. Athena's approach of removing backward edges does not result in any false negatives in the TCP/UDP implementations being analyzed, but it remains an open question of how to remove backward edges in a both sound and precise manner beyond those implementations.

- The uniform distribution assumption on branch outcomes might deviate from the ground-truth distribution of branch outcomes. Take an example of a branch leading to both a sink node and an absence node. If there is practically 75% chance of reaching the sink, then the branch has 0.811 entropy. Meanwhile, the uniform assumption would approximate the branch to have 1 entropy, thus resulting in different entropy values of the branch. But we note that the change in entropy values may or may not change the overall ranking of critical branches. Moreover, a change in the ranking only affects the *quality* of reports (i.e., the number of top-ranked positives in each iteration and how many iterations Athena takes until termination). The uniform distribution assumption will not lead to false negatives.

Generalizability of Athena. While we present the instantiations of Athena exclusively in TCP/UDP implementations, our insight and algorithm are general and they can be applied to other network protocols, and even other scenarios with some extra effort. We believe that the workflow of the analysis would remain the same for other scenarios, though extra domain knowledge is needed. One example is manual specification of sources and sinks. For TCP/UDP implementations, such manual efforts are relatively low since the sources are usually specified in TCP/UDP standards (e.g., RFCs) and attack papers, and the sinks are specified by the threat model. However, the task of identifying sources and sinks may not be as easy for other scenarios as it may require domain expertises. In addition, one also needs domain knowledge to understand the severity and the exploitability (e.g., what could be the potentially different outputs) of reported side channels. Moreover, as discussed earlier, several approximations in the static analysis of our approach might turn out to be problematic for other scenarios. For example, some coding styles with intensive use of pointer arithmetic, design of complex data structures, and different memory allocation strategies might have negative impacts on analysis precision. In such cases, fine-turning the static analysis used by Athena is required to achieve a low false-positive rate.

Responsible disclosure.. We have reported our findings to the Linux, FreeBSD, and OpenBSD community.

9 Conclusion

This paper presents Athena, an automated tool for developers to detect side-channel vulnerabilities in vanilla implementations of TCP and UDP. Additionally, an automated diagnostic mechanism is developed to rank these vulnerabilities according to their importance, and further, pinpoint their origins. We evaluate Athena on several TCP and UDP implementations including Linux, FreeBSD and OpenBSD kernels; Athena successfully detects both previously reported side-channel vulnerabilities and new vulnerabilities within our threat model, with a low false positive rate. In the future, we plan to address the limitations of our static analysis, extend our approach to other protocols, such as QUIC, as well as more general scenarios, such as network traffic analysis.

Acknowledgement

We express our gratitude to the anonymous reviewers and the shepherd for their insightful feedback and suggestions. This work has been supported by the NSF under grants 2401182, 2401496, 2145631, 2215017 and 2226447, the Defense Advanced Research Projects Agency (DARPA) under contract number D22AP00148, the NSF and Office of the Under Secretary of Defense– Research and Engineering under grant ITE 2326898, and the National Telecommunications and Information Administration (NTIA)'s Public Wireless Supply Chain Innovation Fund.

References

[1] Geoffrey Alexander and Jedidiah R Crandall. Off-path round trip time measurement via tcp/ip side channels. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 1589–1597. IEEE, 2015.

[2] Fatemah Alharbi, Jie Chang, Yuchen Zhou, Feng Qian, Zhiyun Qian, and Nael Abu-Ghazaleh. Collaborative client-side dns cache poisoning attack. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1153–1161. IEEE, 2019.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying

constant-time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 53–70, 2016.

[4] Fred Baker. Requirements for ip version 4 routers. Technical report, 1995. https://www.rfc-editor.org/rfc/rfc1812.

[5] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. Constantine: Automatic side-channel resistance using efficient control and data flow linearization. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security*, pages 715–733, 2021.

[6] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V Krishnamurthy, and Lisa M Marvel. Off-path tcp exploits: Global rate limit considered dangerous. In *USENIX Security Symposium*, pages 209–225, 2016.

[7] Yue Cao, Zhongjie Wang, Zhiyun Qian, Chengyu Song, Srikanth V Krishnamurthy, and Paul Yu. Principled unearthing of tcp side channel vulnerabilities. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 211–224, 2019.

[8] Rocky KC Chang. Defending against flooding-based distributed denial-of-service attacks: A tutorial. *IEEE communications magazine*, 40(10):42–51, 2002.

[9] Weiteng Chen and Zhiyun Qian. Off-path tcp exploit: How wireless routers can jeopardize your secrets. In *27th USENIX Security Symposium USENIX Security 18)*, pages 1581–1598, 2018.

[10] Tianxiang Dai, Philipp Jeitner, Haya Shulman, and Michael Waidner. From ip to transport and beyond: cross-layer attacks against applications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 836–849, 2021.

[11] Roya Ensafi, Jeffrey Knockel, Geoffrey Alexander, and Jedidiah R Crandall. Detecting intentional packet drops on the internet via tcp/ip side channels. In *Passive and Active Measurement: 15th International Conference, PAM 2014, Los Angeles, CA, USA, March 10-11, 2014, Proceedings 15*, pages 109–118. Springer, 2014.

[12] Roya Ensafi, Jong Chun Park, Deepak Kapur, and Jedidiah R Crandall. Idle port scanning and non-interference analysis of network protocol stacks using model checking. In *USENIX Security Symposium*, pages 257–272, 2010.

[13] Xuewei Feng, Chuanpu Fu, Qi Li, Kun Sun, and Ke Xu. Off-path tcp exploits of the mixed ipid assignment. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1323–1335, 2020.

[14] Xuewei Feng, Qi Li, Kun Sun, Chuanpu Fu, and Ke Xu. Off-path tcp hijacking attacks via the side channel of downgraded ipid. *IEEE/ACM Transactions on Networking*, 30(1):409–422, 2021.

[15] Xuewei Feng, Qi Li, Kun Sun, Ke Xu, Baojun Liu, Xiaofeng Zheng, Qiushi Yang, Haixin Duan, and Zhiyun Qian. Pmtud is not panacea: Revisiting ip fragmentation attacks against tcp. In *Proc. Netw. Distrib. Syst. Secur. Symp*, pages 1–18, 2022.

[16] Yossi Gilad and Amir Herzberg. Off-path attacking the web. In *WOOT*, pages 41–52, 2012.

[17] Brendon Harris and Ray Hunt. Tcp/ip security threats and attack methods. *Computer communications*, 22(10):885–897, 1999.

[18] Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. Exploring and enforcing security guarantees via program dependence graphs. *ACM SIGPLAN Notices*, 50(6):291–302, 2015.

[19] Jeffrey Knockel and Jedidiah R Crandall. Counting packets sent between arbitrary internet hosts. In *FOCI*, 2014.

[20] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. *ACM SIGPLAN Notices*, 42(6):278–289, 2007.

[21] Jonathan Lemon. Resisting syn flood dos attacks with a syn cache. In *BSDCon 2002 (BSDCon 2002)*, 2002.

[22] Keyu Man, Zhiyun Qian, Zhongjie Wang, Xiaofeng Zheng, Youjun Huang, and Haixin Duan. Dns cache poisoning attack reloaded: Revolutions with side channels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1337–1350, 2020.

[23] Keyu Man, Xin'an Zhou, and Zhiyun Qian. Dns cache poisoning attack: Resurrections with side channels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3400–3414, 2021.

[24] Yamamoto Masaya. microps: An implementation of a small tcp/ip protocol stack for learning. https://github.com/pandax381/microps, 2023.

[25] Zhiyun Qian and Z Morley Mao. Off-path tcp sequence number inference attack-how firewall middleboxes reduce security. In *2012 IEEE Symposium on Security and Privacy*, pages 347–361. IEEE, 2012.

[26] Zhiyun Qian, Z Morley Mao, and Yinglian Xie. Collaborative tcp sequence number inference attack: how to crack sequence number under a second. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 593–604, 2012.

[27] Anantha Ramaiah, Randall Stewart, and Mitesh Dalal. Improving tcp's robustness to blind in-window attacks. Technical report, 2010. https://www.rfc-editor.org/rfc/rfc5961.

[28] Claude E Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[29] Haya Shulman and Michael Waidner. Fragmentation considered leaking: port inference for dns poisoning. In *Applied Cryptography and Network Security: 12th International Conference, ACNS 2014, Lausanne, Switzerland, June 10-13, 2014. Proceedings 12*, pages 531–548. Springer, 2014.

[30] Geoffrey Smith. Quantifying information flow using min-entropy. In *2011 Eighth International Conference on Quantitative Evaluation of SysTems*, pages 159–167. IEEE, 2011.

[31] Altran Intelligent Systems. picotcp: a small-footprint, modular tcp/ip stack designed for embedded systems and the internet of things. https://github.com/tass-belgium/picotcp, 2019.

[32] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 15–26. ACM, 2018.

[33] Xu Zhang, Jeffrey Knockel, and Jedidiah R Crandall. High fidelity off-path round-trip time measurement via tcp/ip side channels with duplicate syns. In *2016 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2016.

[34] Quan Zhou, Sixuan Dang, and Danfeng Zhang. CtChecker: a precise, sound and efficient static analysis for constant-time programming. In *Proceedings of the European Conference on Object-Oriented Programming*, 2024.