# SCAD: Towards a Universal and Automated Network Side-Channel Vulnerability Detection

Keyu Man, Zhongjie Wang, Yu Hao, Shenghan Zheng, Xin'an Zhou, Yue Cao, Zhiyun Qian
{*kman001, zwang048, yhao016, szhen075, xzhou114, ycao009, zhiyunq*}*@ucr.edu*
*University of California, Riverside*

***Abstract*—Network side-channel attacks have recently been highlighted due to their severity and elusive nature. For example, SADDNS attacks allow an off-path attacker to launch cache poisoning attacks leveraging network side channels. Due to the subtle nature of network side channels, it is challenging to identify such side channels. To this date, few automated bug discovery techniques are tailored for such vulnerabilities. Unfortunately, none of them is general and automated enough, making their impact and longer-term use limited. In this paper, we describe the first solution that aims to fill this gap. Specifically, we develop SCAD, aiming at identifying violations of the non-interference property, which are commonly understood as the root cause of network side channels. As non-interference property is a hyperproperty, it necessitates reasoning across multiple execution traces. This motivated us to develop our solution based on under-constrained and dynamic symbolic execution.**

**The state-of-the-art solution, SCENT, applies model checking, which requires extra effort in modeling or simplifying certain parts of a network protocol, in order to scale. Unfortunately, such modeling and simplification is time-consuming, error prone, and can overlook important details, leading to missed vulnerabilities. For example, it was reported that 2.5 person-week was required to construct a self-contained using SCENT. In comparison, SCAD requires only a single person-day to perform labeling of secrets and attacker-observables, and decide the analysis scope. By applying SCAD to multiple TCP and UDP implementations, including Linux, FreeBSD, and lwIP, we find 14 network side-channels, 7 of which were previously unknown, with a false positive rate of only 17.6%. The results reveal serious vulnerabilities, including those that can be used to compromise the previously patched Linux and FreeBSD kernels, making them susceptible to SADDNS attacks or off-path TCP exploits. Our analysis concludes that the majority of the side channels cannot be found by existing solutions due to the aforementioned limitations.**

## 1. Introduction

Network side channels were recently demonstrated to lead to serious attacks. For example, such vulnerabilities can empower anyone with IP spoofing capability to poison the DNS cache of a DNS resolver [1], [2], manipulating TCP connections established between any two hosts [3], [4], [5], or even decipher Wi-Fi passwords [6].

While the threat to network security is undeniable, the discovery of most network side channels remains largely a manual endeavor. Despite the proliferation of tools for automated microarchitectural side channel detection [7], [8], [9], [10], [11], [12], only a handful cater to network side channels[1] [13], [14].

The existence of side channels can be fundamentally formulated as violations of the non-interference property [15], which is a hyperproperty that predicates on relations between execution traces of the attacker and the victim [16]. In essence, it requires a detailed and precise analysis of the interactions between some shared resources and two parties (i.e., attacker and victim), where the interactions can lead some sensitive data to flow through such resources to the attacker. Unfortunately, prior automation attempts, such as `PacketGuardian` [14], instead of looking for violations of the non-interference property, looked for simpler patterns as approximations of non-interference property which has resulted in numerous false positives. `SCENT` [13] adopted a more principled solution, leveraging model checking, to detect non-interference property violations and, by extension, side channels. Yet, the significant reliance on manual interventions *(e.g.,* extraction of relevant functions, abstraction of external functions, marking of state variables) and heuristics bound to protocol implementations *(e.g.,* downscaling) limits its generality, usability, and even completeness. As will be demonstrated in our evaluation, it misses important side channels that are unveiled by our solution.

Given the current landscape, we see an important gap in addressing the issue of side-channel attacks. Specifically, we see a lack of **reusable and general tools** that can be easily applied to a variety of protocol implementations (including future ones). To fill this gap, we introduce `SCAD` (Side-ChAnnel Detector) — an analysis tool for automated side-channel detection. At a high level, `SCAD` employs symbolic execution to explore the state space of a target protocol implementation. For each path, it summarizes the associated data flows regarding (1) how a secret propagates to various shared variables in the protocol along each execution path, and (2) how the value of shared variables may influence

1. For the rest of the paper, the term "side channel" will specifically refer to "network side channel", unless otherwise specified.

attacker-observables (e.g., presence or absence of a response, or differences in a response). Then, it pairs different paths and their associated data flow behaviors to look for any non-interference property violations.

However, network side channels can require hundreds of packets to trigger [1], [2], [3], making the desirable part of the state space difficult to reach using the traditional dynamic symbolic execution [17] where hundreds of symbolized packets need to be analyzed. To avoid such an expensive search, SCAD employs a particular selective symbolic execution to (1) symbolize not only packets but also protocol states, and (2) allow switching between expensive symbolic execution and concrete execution. Such a configuration balances soundness and completeness. In particular, the former can lead to false alarms in reported bugs as it is effectively making the symbolic execution under-constrained, but it avoids the expensive analysis of many symbolized packets. The latter sacrifices completeness, leading to potentially missed bugs. This approach is necessary due to the complexity of network protocol stacks in real-world operating systems. It is essential to restrict the scope of symbolic execution to the specific protocol under analysis.

To evaluate SCAD, we applied it to diverse targets, including the TCP implementations of Linux, FreeBSD, and lwIP, and the UDP implementation of Linux, while also incorporating other protocols that interact with these protocols, e.g., ICMP. With a minimal effort of specifying the secrets and attacker-observables for each target, SCAD seamlessly runs on these targets. With a 48-hour symbolic execution run for each target, SCAD reports 17 side channels, of which 14 were true positives (TPs), and unveiling 7 previously unknown side channels.

**Contributions.** Overall, our contributions can be summarized as follows:

- We develop a solution that requires minimal manual effort compared to state-of-the-art to discover network side channels in real-world network protocols.
- We leverage a specific configuration of symbolic execution that is the most suitable for the task of identifying network side channels.
- We report 7 previously-unknown side channels in previously patched protocols and analyze why such vulnerabilities cannot be found by prior work.
- We open sourced our solution[2] to facilitate the reproduction of results and future research in this domain.

# 2. Background & Motivation

## 2.1. Off-Path Attacks

Off-path attacks refer to a type of 'blind' network attack where the attacker, without the ability to eavesdrop or intercept the victim hosts' communications, can still deduce confidential information about the victims. This is achieved by sending probing packets, including those with spoofed

2. https://github.com/seclab-ucr/SCAD

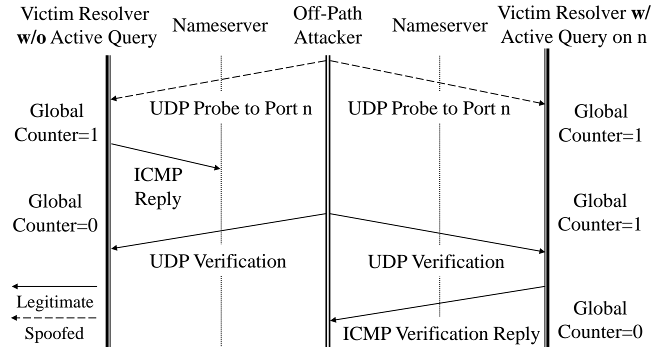



Figure 1: Simplified SADDNS attack mechanism

IP addresses. Off-path attacks have been known for a few decades, although popularized in the recent decade or so as exemplified by [1], [2], [3], [18], [19], [20], [21], [22], [23], [24]. A notable attack documented in 2016 [3] involved an off-path attacker to infer if any two arbitrary hosts on the Internet are communicating using a TCP connection. Further, if the connection is present, such an off-path attacker can also infer the TCP sequence numbers in use, from both sides. This in turn allows the attacker to forcefully terminate the connection and perform data injection attacks. In 2020 and 2021, there are two other prominent examples where an off-path attacker successfully revived the DNS cache poisoning attack using network side channels [1], [2], which have received significant attention [25], [26]. Such attacks are very powerful because they can be launched completely remotely and off-path.

These attacks hinge on two fundamental requirements: (1) the attacker can send packets with spoofed source IP addresses which is allowed in the majority of ASes according to a recent study [27]. (2) there exist some shared resources that inadvertently disclose information about the victims to the attacker through the responses elicited by their probes.

## 2.2. A Motivating Example: SADDNS

SADDNS [1] is a recent and prominent off-path attack that successfully leads to the powerful DNS cache poisoning. Specifically, by leveraging a subtle side channel, an off-path attacker can quickly infer the ephemeral port number associated with a DNS request initiated by a DNS resolver. This enables the attacker to send a spoofed malicious DNS response as a result (by bruteforcing the 65,536 possible transaction IDs). The side channel stems from the global ICMP rate limit counter, a shared token bucket governing ICMP replies. As the name suggests, each ICMP packet emission decreases the counter (or available tokens) by one and no ICMP is allowed to be sent if the counter reached zero.

Figure 1 illustrates the attack and can be divided into two parts: the right part refers to the case where there is an active DNS query using the ephemeral port $n$ on the victim resolver and the left part refers to the case without an active

query. The attack works as follows: (1) the attacker sends a spoofed UDP probing packet (with the source IP of the server that the resolver queries) to port $n$, depending on whether the port is open (right) or closed (left), the resolver will either not change the counter (right) or decrement the counter (left); (2) the attacker then infers the value of the global counter by sending a verification UDP packet using its own IP address as source IP to solicit an ICMP reply. As depicted in Figure 1, by observing whether the ICMP verification reply is generated, the attacker can distinguish between the above two cases. This essentially allows an off-path attacker to scan the ephemeral port status via the shared ICMP counter state. Note that in the actual Linux Kernel, the value of the rate limit counter is 50 by default, which requires 50 probe packets to reach. But for ease of understanding, we simplify the value to 1 in Figure 1.

```
1  int __udp4_lib_rcv(struct sk_buff *skb, ...) {
2      struct sock* sk =
           __udp4_lib_lookup_skb(skb->src, skb->dst,
           skb->sport, skb->dport);
3      if (!sk) {
4          if (icmp_global.credit) {
5              icmp_global.credit--;
6              icmp_push_reply(...);
7          }
8      }
9      return 0;
10 }
```

Listing 1: SADDNS vulnerability in Linux kernel (simplified)

Listing 1 further illustrates the vulnerability in detail. The code is distilled from the incoming UDP packet processing logic in the vulnerable Linux kernel. First, line 2 tries to match the 4-tuple of the incoming UDP packet to the existing sockets. Then depending on the match result, the kernel will try to emit an ICMP error packet (line 6) if there is not a match, and deduct one token (line 5). The difference in behavior of token deduction, depending on whether the four-tuple of an incoming packet matches an ongoing socket, leads to the side channel.

In the final phase of the attack, the verification packet sent by the attacker targets a confirmed closed port. This invariably prompts the victim resolver to execute line 4. Essentially, the attacker can deplete the token limit by dispatching multiple verification packets and monitoring the number of triggered responses. This method effectively reveals the value of the variable `icmp_global.credit`, which is different depending on whether the previously spoofed probing packets hit the correct four-tuple or not. In other words, the secret of "whether a guessed port is correct or not" leaked through the global/shared variable of `icmp_global.credit`, to the attacker-observable *(i.e.,* the presence or absence of an ICMP response).

# 3. Overview

## 3.1. Threat Model

Previous off-path side-channel attacks, as exemplified by [1], [2], [3], [6], [23], [24], can be generalized into a model where two victim hosts (e.g., DNS resolver and an authoritative nameserver) communicate, using some shared secret (e.g., ephemeral port number) that is not visible to an off-path attacker. By definition, an off-path attacker is unable to eavesdrop or tamper with the communication between the victim hosts. However, we assume that such an off-path attacker can leverage IP spoofing (which is allowed in the majority of ASes according to a recent study [27]), and therefore can craft and send any packets using spoofed source IPs (e.g., nameserver's IP). Note that an off-path attacker may not observe any response of the spoofed packets (as they are destined to the spoofed IP address) unless they send non-spoofed probing packets (i.e., with their own IP address as source IP). The primary objective of such attacks is to deduce secrets by sending a combination of spoofed and non-spoofed packets. SCAD aims to detect such vulnerabilities using automated analysis techniques.

## 3.2. Modeling Network Side Channels as Non-Interference Property Violations Network Side-Channel Definition

Similar to other kinds of side channels [8], we model network side channels as non-interference property violations: given a protocol implementation $P$ with a memory state $M$ that captures input, output, and state of the protocol. We can divide $M$ into a low (sensitivity) part $M_L$ *(e.g.,* storing one or more input packets from untrusted remote attackers) and a high (sensitivity) $M_H$ *(e.g.,* storing the ephemeral port number of sockets). $P$ adheres to non-interference property if and only if, for any two initial memory states $M_1$ and $M_2$ with the same low-sensitivity memory *(i.e.,* $M_{L1} = M_{L2}$), after executing $P$, the resulting low-sensitivity memory state remain identical *(i.e.,* $(P(M_1))_L = (P(M_2))_L$) [15], [16], [28]. In essence, the processing of low-sensitivity memory (i.e., attacker inputs) should produce identical attacker-observable outputs, which remain unaffected by variations in high-sensitivity memory states. Note non-interference property violation is orthogonal to the semantics of the memory *(e.g.,* randomness). In fact, randomness only reduces the entropy of the useful information deduced by the attacker but it does not eliminate the violation of non-interference property (as shown in Section 7.1 and Section 7.2).

Accordingly, we consider any network side-channels to be in-scope if there is a notion of multi-user, multi-session in a protocol where (1) there exist some victim secrets; (2) there are shared resources across users/sessions; (3) an attacker can observe responses through his own session by probing the session associated with the victim. In this paper, we focused on TCP, UDP, and ICMP protocols because they meet the above conditions.

## 3.3. Challenges of Finding Network Side Channels

Unlike other classic side channels such as the CPU cache side channels, network side channels are uniquely challenging in the following aspects.:

First, (1) iIt requires multiple rounds of interactions, i.e., probing, to reveal a network side channel. This is because side channels are often triggered in a specific state (under corner cases) and require a careful sequence of probing packets, including both spoofed and non-spoofed packets.

Second, (2) tThe propagation distance from secrets to attacker-observables is much longer in network side channels, due to the multiple rounds of interactions. Cache side channels typically manifest in unique cache footprint as soon as secrets are accessed in memory (e.g., secret access affecting control-flow path). In contrast, network side channels often propagate through intermediate shared resources, and require additional probing packets to leak to attacker-observables.

Third, (3) dDue to the fact that network protocols are complex and can involve dependencies, *e.g.,* lower-layer protocols such as IP and even device drivers, it is necessary to limit the scope of the analysis appropriately as to not overburden the symbolic execution. Unlike user-space programs where common library functions are already modeled, *e.g.,* `memcpy()`, dependencies in the OS kernel have not been well-summarized.

(4) Unlike many passive side channels that allow continuous monitoring (such as watching physical signals during the victim's execution), a network attacker only sees a response packet after the victim has fully processed the probing packet.

(5) Unlike many side channels that focus on novel leakage channels (e.g., electromagnetic waves or accelerometer readings), network side channels focus on novel shared resources (any shared memory can in theory be contributing to a side channel). However, the number of such shared resources is unknown and dependent on implementations. Furthermore, the uses of such shared resources may be diverse and dependent on protocol states.

**Requirements.** To account for the above uniqueness of network side channels, we will need to

(1) reason about protocol behaviors after processing different sequences of packets, *i.e.,* , stitching together execution traces of multiple packets;

(2) explicitly consider potential intermediate shared resources that could ultimately lead to interference with attacker-observable outputs;

(3) limit the analysis scope of the protocol of interest to avoid overburdening the symbolic execution.

### 3.4. Existing Solutions

In practice, network side channels are often discovered by hinging the analysis on interesting shared resources (2nd requirement as mentioned above). In the `SADDNS` example, the root cause of the vulnerability lies in the underlying shared resource, i.e., a single global variable `icmp_global.credit`, and how it interacts with an attacker and a victim. To find other potential vulnerabilities like this, one option is to inspect each and every variable at the implementation level that is possibly shared between an attacker and a victim. This is a daunting task and thus requires automated analysis approaches.

To identify non-interference violations, we can in principle apply a variety of automated formal methods and testing techniques, e.g., static analysis, fuzzing, model checking, and symbolic execution. Each of these methods offers unique strengths and weaknesses when tailored to the specific problem. In the ensuing discussion, we compare them to motivate the design choice of our solution.

**Static analysis.** `PacketGuardian` [14] employed static taint analysis to detect "implicit information leakage", which is an approximation of the violations of non-interference property. However, since it is much less precise, it incurs a high FP rate, primarily attributed to the absence of path sensitivity and the lack of reasoning of relationships among execution traces (required by the non-interference property) [29].

**Fuzzing.** As a dynamic analysis technique that feeds random inputs to test program behaviors by executing them concretely. The advantage of fuzzing is that whatever bugs or violations of non-interference must be true positives, as they can be proven with concrete inputs and execution traces. Although [30] utilized fuzzing to test non-interference property violations, it is by design probabilistic and its sporadic exploration raises concerns regarding coverage, i.e., false negatives.

**Model checking.** In contrast to fuzzing, model checking aims to systematically (and potentially exhaustively) cover the state space of the test target. `SCENT` [13] leveraged a model checker to systematic search for the existence of non-interference property violations in TCP implementations. A key practical challenge lies in the creation of a self-contained model by converting and abstracting actual TCP implementations, including extraction of relevant functions, abstraction of external functions, and marking of state variables. This unfortunately demands substantial manual effort, domain expertise, and consequently is not only prone to human error, but limits the generality, usability, and completeness of the tool. Specifically, to mitigate the state explosion problem, `SCENT` chooses to (1) limit the number of state variables *(e.g.,* by not considering time-related variables as part of the state) and (2) abstract away certain dependencies *(e.g.,* eliminating sources of randomness to simplify the state space), both of which lead to missed vulnerabilities as we will explain in §6 and §7.

Not surprisingly, we find `SCENT` missed some side channels that are discovered by `SCAD` (as will be shown in our evaluation).

### 3.5. The Approach of `SCAD`

Unlike the above solutions, `SCAD` leverages symbolic execution to detect non-interference violations. We propose a symbolic execution based solution for two reasons: (1) Non-interference property is a hyperproperty that requires reasons across multiple traces or execution paths, and the symbolic execution by design traces the execution on a path

basis; (2) Symbolic execution can analyze protocol implementations directly and does not require heavy modeling of protocol behaviors or functions (unlike model checking).

Let us consider a concrete example to illustrate why symbolic execution / SMT solver is more appropriate for non-interference violation detection than existing static analysis [14]. Consider the following example abstracted from `tcp_validate_incoming()` in `net/ipv4/tcp_input.c` of Linux kernel 6.1.32.

```
1 bool in_window = tcp_sequence(pkt_seq);
2 if (in_window) {
3   if (!th->rst) {
4         tcp_send_dupack();
5   }
6 }
7 if (th->rst) {
8     tcp_send_challenge_ack();
9 }
```

Listing 2: Simplified `tcp_validate_incoming()`

In this case, a static analysis will think the path traversing lines 1-4 and the path traversing lines 1,2,7,8 forms a side channel. This is because it thinks the behaviors in Line 4 or Line 8 are different *i.e.,* sending a challenge ACK vs. sending a dup ACK, depending on whether the seq number is in window (Line 1). However, given an input packet, it can pass either the check at Line 3 or Line 7 but not both (note the RST flag of an incoming packet has to be either true or false), so the attacker cannot observe any difference in terms of whether the sequence number guessed is in window or not. SCAD will not produce such a false positive thanks to its constraint-solving capability.

Nevertheless, itsymbolic execution is a heavy program analysis technique, especially in the context of complex network protocols. Worse, these protocols operate inside even bigger OS kernels and interact with rest of the kernel; faithfully analyzing the behaviors of the protocols may or may not require understanding of the other "glue" parts.

To this end, we choose to apply selective symbolic execution [31] to address the above challenges. At a high level, we perform symbolic execution selectively in two aspects: (1) selectively symbolize variables, (2) selectively switch between concrete execution and symbolic execution. We detail the design decision below. As will be shown in Section 6, our solution can indeed successfully reveal new side channels that were previously unknown. We will elaborate on the selective symbolic execution in Section 4.2.

## 4. SCAD Design

### 4.1. SCAD Architecture

The SCAD framework, as depicted in Figure 2, consists two components: the selective symbolic execution component and the Non-Interference Property Violation Checker (NIPVC) component. The former takes two inputs: (1) a live protocol memory snapshot, (2) annotations provided by a security analyst regarding the variables considered as secrets (e.g., ephemeral port number), events/functions considered as attacker-observables, and scope of the protocol to analyze (in the form of memory address ranges). It then conducts selective symbolic execution of the protocol and generates "path summaries" capturing critical information necessary for determining non-interference. The latter ingests the path summaries, orchestrates them to form path-pairs, and subsequently checks for non-interference property violations in these pairs (recall that we are analyzing relationship among traces to verify a hyperproperty). The outcome is a list of detected side channels, with proofs of violations represented as execution traces and implicated state variables.

### 4.2. Symbolic Execution Design

In this section, we discuss the choices of the symbolic execution engine. The choices are important because they affect the soundness, precision, and scalability of our analysis.

**Addressing requirement #1: reasoning over multiple packets via selectively symbolizing variables.** A naive design is to symbolize multiple input packets to drive the execution of a concrete protocol instance. Specifically, only the input packets are symbolized and the rest of the program (e.g., state variables) remains concrete [32], [33], [34], [35]. However, as mentioned before, side channel vulnerabilities in network protocols often require over a hundred packets to reveal and trigger [1], [2], [3]. This means if we were to begin the exploration of a network protocol implementation at its initial state, *e.g.,* after a fresh TCP connection is established, we need to enumerate long sequences of packets by symbolizing all these packets. This is unfortunately not feasible. In prior work, only three symbolic packets have been successfully attempted in the past [35], which are far from sufficient to reveal network side channels.

A better design is to symbolize a single input packet as well as state variables. Our insight is that we found most input packets primarily steer the system towards a specific state where the non-interference property violation becomes possible. Once this state is achieved *(e.g.,* reaching the global rate limit of certain packets), a single packet suffices to trigger the violation of non-interference. Therefore, instead of relying on unbounded input packet enumeration to reach this state, one can assume the system to already be in arbitrary states by "symbolizing the state variables". In other words, *in addition to symbolizing the input packets, we will also symbolize state variables*. This way, we no longer need to enumerate long sequences of input packets; instead, one single symbolic input packet suffices.

We will treat any non-stack *(e.g.,* heap, static) variable, within the symbolic execution range (see §5.2), as a state variable. The exclusion of stack variables is because they are local in scope (processing of a single packet) and therefore not state variables. We opt for a lazy symbolization strategy for state variables: a state variable is symbolized only when it is read — this is because if a state variable is never read, then it will never affect the execution, and thus there is no need to symbolize it.

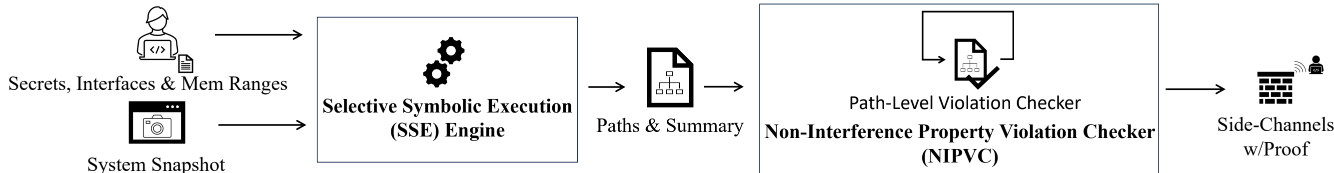



Figure 2: `SCAD` architecture

It is worth noting that once input variables and state variables are symbolized, all of them will be treated the same way (under-constrained) *(i.e.,* we do not differentiate input vs. state variables separately during the symbolic execution). See §4.3 for details.

Taking `SADDNS` as an example, if `icmp_global.credit` in Listing 1 is symbolized, the path summary will directly show a reply will or will not be solicited when `icmp_global.credit` is not or is zero when the input is a UDP packet, without the need to send 49 repeated packets in advance reducing the concrete value of `icmp_global.credit` from 50 to 1.

In summary, this choice makes the tradeoff between scalability and precision. By requiring the analysis of only a single symbolic input packet, the choice satisfies the first requirements as mentioned in §3.3 in a highly scalable manner. The downside is that using symbolized state variables (as opposed to their concrete values) makes the results no longer sound, *i.e.,* false positives can occur due to the under-constrained nature of the analysis.

**Addressing requirement #2: Explicit consideration of intermediate shared resources by tracking all their changes.** As mentioned, oftentimes the secret propagates through intermediate shared resources, *i.e.,* shared variables accessed when processing both spoofed packets and non-spoofed packets. To this end, `SCAD` records all changes to symbolized state variables (which may be potentially shared) during symbolic execution along every single path. This way, the NIPVC component will have the information available to determine which two execution paths would allow secrets to interfere with the shared resource (see §4.3 for details). This design satisfies the last requirement mentioned in §3.3.

**Addressing requirement #3: limit the symbolic execution scope via selective symbolic/concrete execution.** It is natural to consider selectively switching between symbolic execution and concrete execution. First, actual protocol implementations interact with other parts ("glues") of the kernel which can lead to significantly more execution paths and states. For example, when symbolically executing helper functions like `printk()`, a symbolic execution engine may generate multiple execution paths (if the arguments are symbolized) due to the fact that such functions contain loops. By selectively executing such helper functions concretely, it drastically reduces the number of symbols and states and thus increased the scalability.

Second, beyond the helper functions, there can be other more glue code that should not be pruned; otherwise, the analysis may not be faithful. For example, when analyzing the TCP implementation, it may naturally interact with the IP implementation. We do not have to execute the IP layer symbolically, but executing the code in the IP layer concretely will help preserve the faithfulness of the execution.

This is supported by a state-of-the-art symbolic execution engine named S2E [31], which we leverage to develop `SCAD`. The idea is to associate each symbol with a possible concrete value under each state, and whenever the target runs into an uninteresting section *(e.g.,* `printk()`), it executes the code concretely. It is worth noting that sometimes side channels can manifest via interactions of two protocols, As seen in the motivating example, the interaction between UDP and ICMP led to the vulnerability. In practice, based on our expertise in network protocols, we choose to selectively include additional dependent protocols as part of our analysis scope.

| Technique | Precision | Coverage | Generality |
|---|---|---|---|
| Static Analysis [14] | ✗ | ✓ | ✓ |
| Fuzzing [30] | ✓ | ✗ | ✓ |
| Model Checking [13] | ✓ | ✓ | ✗ |
| Symbolic Execution | ✓ | ✓ | ✓ |

Table 1: Comparison among program analysis techniques

**Output.** During the symbolic execution, `SCAD` computes a path summary associated with each execution path by logging the path constraints and symbolic expressions relating to symbolized memory (for both input and state variables). The path constraints are useful for determining if a path is feasible. If it is, NIPVC can later check if a combined path (formed by concatenating two path constraints) is also feasible. The symbolic expressions *(e.g.,* write of symbolic state variable) will be useful for NIPVC to decide whether a secret propagates to an intermediate shared resource *(i.e.,* interferes with the shared resource).

**Comparison with existing approaches.** Compared with model checking, in addition to alleviating the modeling efforts, our approach is also more complete by eliminating input packet enumeration. Compared with static analysis, as an ultimate form of static analysis, symbolic execution by design will present less FPs. Compared with fuzzing, symbolic execution offers superior coverage due to its systematic path exploration strategy. As summarized in Table 1, our approach stands out as the appropriate solution for a general side-channel detector.

### 4.3. NIPVC Component

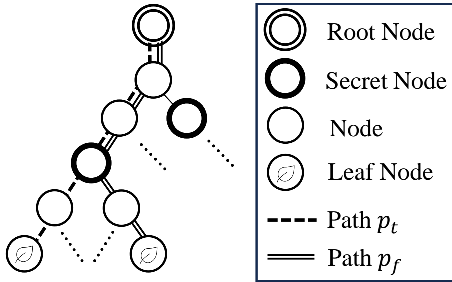



Figure 3: An illustrative example of path tree

**Path-Level Violation Checker.** The NIPVC component processes the paths and summaries generated by the DSE component, as outlined in Section 4.1. Figure 4.3 provides a visual representation of a typical path tree generated by symbolic execution, where nodes represent forking events and edges signify branches. The root and leaf nodes demarcate the beginning and end of a path execution, respectively.

Given the definitions of non-interference property and the associated threat model, a violation implies that, 1) given the same sequence of probing packets created by an off-path attacker (same low input), 2) depending on the different values of the secret (different high memory), 3) the resulting attacker-observable output will be different (different low output). Since the non-interference property is a hyperproperty [36] that requires two traces to verify, in the context of symbolic execution, this translates to the existence of two paths that satisfy the following:

I. the two paths can assign the same value to symbolic inputs (i.e., attacker sending the same crafted packet and two paths refer to the same system state other than the unknown secret).
II. one of the two paths takes the true branch and the other takes the false branch after forking from a node representing the secret (secret node in Figure 4.3) – this implies the path summary can differ depending on the value of the secret (high memory); and
III. the two paths write to at least one state variable[3] differently (e.g., one updates the state variable and the other does not).

We point out that I.- III. correspond to 1) - 3). The NIPVC's primary objective is to ascertain the existence of such path pairs.

To find non-interference property violation for a secret, based on the above observation, NIPVC first traverses the path tree to gather all secret nodes, which can be easily differentiated from the forking conditions. For each secret node, it arranges paths, that pass through the secret node, in pairs, and perform the above check for them, and report the found violation.

To formally express the conditions for non-interference property violation, denote the secret symbol as $s$, non-secret symbols as set $N$, and the path constraints of the path-pair $(p_t, p_f)$, as shown in Figure 4.3, as assertions $C_t(s, N)$ and $C_f(s, N)$ (is true) respectively. To satisfy II., two paths need to take different secret values, therefore a shadow secret symbol $s'$ is used for the false (or true) branch to replace $s$ and then its constraint becomes $C_f(s', N)$. Satisfying I. then becomes satisfying $C_t(s, N) \wedge C_f(s', N)$. Note that additionally satisfying $s \neq s'$ is unnecessary as $s = s'$ will dissatisfy I. by introducing the similar contradiction as mentioned above. As stated in Section 5.3, DSE component records the memory write summary of variable $v$ on path $p_q$ as an expression $W_q^v(s, N)$. Therefore, III. can be written as assertion $\exists n \in N, W_t^n(s, N) \neq W_f^n(s', N)$. Putting everything together, a pair of path violates non-interference property is equal to $(C_t(s, N) \wedge C_f(s', N)) \wedge (\exists n \in N, W_t^n(s, N) \neq W_f^n(s', N)))$ can be satisfied. This problem can be solved by SAT solver like `Z3`. In the real implementation, the checker will enumerate all possible $n$ and return every $n$ that satisfies the assertion.

**Iterative Analysis.** An end-to-end non-interference property violation is a composite of multiple path-level violations. Consider the motivating example again, the port match result interferes with the rate limit counter — the first condition for path-level violation defined above. Subsequently, the counter interferes with the ICMP reply generation, which is visible to the attacker — the second condition for path-level violation. In other words, a secret must first interfere with some intermediate state variables, and the intermediate variables must then interfere with the attacker-observables (or other intermediate state variables). To uncover end-to-end side-channel attacks, it is imperative to chain multiple path-level violations.

To achieve this, NIPVC operates iteratively, akin to taint analysis, executing the path-level check in a graph traversal fashion. The overarching goal is to trace a chain of violations from the secret to an output observable by the attacker. Specifically, NIPVC maintains a variable set $P$ that $\forall v_p \in P$, $v_p$ can be interfered by the secret $s$ through the chain of interference and initially only $s \in P$. During the iterative analysis, NIPVC will take a variable $v_p \in P$, that has not been checked, as the secret, to run the path-level checker. If a possible $n \in N$, where $N$ is the set of non-secret symbols, is identified, then a propagation $v_p \Rightarrow n$, along with the proving path-pair, will be recorded and the intermediate variable $n$ will be added to $P$.[4], unless $v_o$ is the packet output buffer prepared for the attacker ($v_a$), which is already visible to the attacker and no further propagation is needed. Arguably, the "propagation" gets its name because it reads the secret implicitly embedded in $v_p$ and encode it to $n$. This iterative process continues until all variables in $P$ have been examined. Upon completion, a directed acyclic graph (DAG) is generated, mapping the propagation relationships among variables, with $s$ as the root node *(i.e.,* 0 in-degrees). NIPVC then traverses this

3. Packet outputs are modeled as state variable as well.

4. If $n$ depends on $v_p$ in data-flow, a propagation will also be produced.

graph to ascertain if a path exists from $s$ to $v_a$, subsequently outputting the identified propagation chain and the details of each propagation. Note this DAG should not be confused with path tree as shown in Figure 4.3. In fact, each edge of DAG represents a propagation associated with a path-pair.

# 5. Implementation

## 5.1. Symbolic execution

The component is constructed atop the `S2E` framework [31], which itself is built upon `QEMU` [37] and `KLEE` [38] to allow switching between concrete and symbolic mode. This choice was motivated by the success of similar systems such as those presented in [35], [39]. We implement this component as a plugin for `S2E`. This plugin, crafted in C++, spans approximately 2,300 lines of code (LoC) and is responsible for state variable symbolization and the collection of path summaries. Additionally, we made non-trivial modifications including hundreds lines of code of `S2E` to fix bugs that can only be surfaced at the scale of `SCAD` (and the need to symbolize a large number of state variables) and all of them were merged into the official `S2E` repository.

## 5.2. Manual Efforts

Since side channels are inherently domain-specific, `SCAD` thus requires 3 annotations, which we believe are the minimum requirements for any side-channel analysis, and are less intensive than the state-of-the-art [13]. As mentioned, `SCAD` requires three types of manual modeling or annotations. The first two are necessitated by the problem of side-channel vulnerabilities, whereas the last one is about the analysis scope to keep the analysis problem manageable. These annotations are minimal and much less intensive than the state-of-the-art. We describe them below.

**Secrets.** One needs to mark secrets we are interested in tracking, i.e., in the form of variables in protocol implementations, *e.g.,* TCP sequence number or ephemeral port number.

**Attacker-observables.** One needs to mark attacker-observables, which we define them as “the presence or absence of responses to attacker’s probing packets” in the context of off-path attacks. More concretely, we will mark specific functions in the protocol that corresponds to generating responses.

**Analysis scope.** As mentioned, a protocol implementation such as TCP and UDP is a part of a much larger code base, e.g., the OS kernel, it is necessary to mark the analysis scope for selective symbolic execution (and fall back to concrete execution for code that is out-of-scope). Specifically, we need to mark the start and end points, and any intermediate functions that should be considered in scope – they are specified in the form of memory address ranges that cover parts of the memory snapshot.

Compared with `SCENT`, `SCAD` users do not need to identify state variables that can potentially lead to side channels; instead, we assume any of the heap and global variables (easy to identify by address ranges) can potentially be a shared resource leading to a side channel. Besides, `SCAD` does not require the user to manually translate the implementation into a form amendable to model checker.

## 5.3. Symbolic Execution Workflow

Upon entering the start point, the selective symbolic execution initiates its operation by substituting the input packet with symbols. Taking Linux TCP as an example, it will hook `tcp_v4_rcv()` [40] as start point and replace the TCP header in `skb` with a symbol when reached.

**Selectively symbolize variables.** In `SCAD`, we mark all non-stack variables as symbolic, including heap and global variables that can have a lifetime longer than processing a single packet. We identify such variables by memory address range (they are separated from stacks). During the symbolic execution mode, it computes the path summary by logging data flows relating to symbols (input or state variables) for subsequent use by the NIPVC.

Upon reaching the termination point, the engine concludes the current path, outputting both path constraints and the corresponding path summary. These termination points typically signify the end *(e.g.,* the return of the packet processing function) of input packet processing or the detection of a fatal error *(e.g.,* kernel panic).

## 5.4. NIPVC

This component, written in C++ for enhanced concurrency, spans approximately 5,000 LoC. It implements the path-level checker and the iterative analysis with `Z3`. Its architecture comprises three fully asynchronous services: *Secret Node Finder*, *Path-Pair Generator* and *Path-Pair Checker*, by allowing each service to fully utilize all CPU cores when necessary. *Secret Node Finder* identifies all secret nodes for a given secret, using the log produced by SSE, subsequently passing which to *Path-Pair Generator*, after eliminating duplicates, pairs paths and forwards them to *Path-Pair Checker*. The checker, leveraging `Z3`, implements the algorithm detailed in Section 4.3. It outputs path-level non-interference property violations for the current secret and also feeds interfered variables back to *Secret Node Finder*, establishing the iterative analysis.

# 6. Evaluation

## 6.1. Experiment Setup and Analysis Targets

Our evaluation platform comprises a server equipped with an AMD EPYC 7542 26-Core processor[5] and 2.0TB RAM, running Ubuntu 20.04 with the Linux kernel 5.4.0.

5. Only 26 of the 32 cores are used to ensure server availability.

We configure SCAD to detect side channels that involve only one intermediate state variable. This is because most side-channel attacks [1], [2], [3], [19], [22], [24] involve a single intermediate variable only and expanding the chain to multiple intermediate state variables can be computationally challenging. We therefore refer to the intermediate variable accessed by both the attacker and victim in the threat model (as discussed in Section 2) as the "shared variable". To classify side channels, we cluster the results according to the shared variables that are involved, in line with previous work [13], [14]. Note that although only one intermediate state variable is considered, SCAD still needs to symbolize all state variables, because SCAD cannot predict which variables are the shared variable that leads to a side channel.

To manage the inherent complexities of symbolic execution, we set the maximum iteration in a loop exploration to 2 to avoid path explosion. We set the maximum symbolic execution time to 48 hours. Note the full tool run time also includes NIPVC execution time, which will be discussed in Section 6.4. For each side channel report generated by SCAD, we manually verified its validity. For both TCP & UDP targets, besides their own protocol implementations, we also include the ICMP implementation, as it interacts with TCP & UDP closely (when corner cases occur).

As mentioned in Section 5.2, we had to perform some minimal modeling and annotation for each target. For TCP, we marked all the variables that were previously considered secrets, including ephemeral port number of a live connection, expected TCP sequence number, and expected acknowledgement number. For UDP, we mark the expected port number of a UDP socket as the secret. We take a memory snapshot of the system when there is either a live TCP or UDP socket. For a single target, marking these secrets, as well as attacker-observables and analysis scope did not exceed **a single person-day** according to our experience. This efficiency stands in contrast to SCENT, a state-of-the-art side channel discovery solution [13], which requires 2.5 weeks to construct a self-contained model for a single target. As an automated solution requiring only the bare minimum modeling of the side channel in question, SCAD, compared with the manual solution, SCENT, not only requires less labor intensity, but is more error prone.

We choose the following target protocols:

(1) TCP in Linux kernel v4.8, for comparison against a prior work;

(2) TCP in Linux kernel v6.1.32, FreeBSD kernel v13.2, and lwIP 2.2.0 RC1, for finding new vulnerabilities that affect the latest TCP protocol implementations;

(3) UDP in Linux kernel v6.1.32, for finding new vulnerabilities that affect the latest UDP protocol implementations.

Next, we will first summarize the results of all the side channels in Section 6.2, followed by a comparison with SCENT [13] in Section 6.3. The performance analysis will be presented in Section 6.4.

## 6.2. Summary of Results

As shown in Table 2, SCAD reported 13 side-channel vulnerabilities discovered across all target protocols. We confirm 11 of them are true positives (see column "TP/FP"), 6 of which are previously unknown (see column "New?"). These vulnerabilities span across all target protocols, including both UDP and TCP. For each case, we listed the shared variable name, the type of variable, and the kind of secrets that are leaked through the shared variable.

For Linux kernel v4.8, and v6.1.32, SCAD identified seven side channels, including two new side channel, #1 and #3, and two false positives, #6 and #7. We explain the causes of the false positives in Section 9. For FreeBSD, three side channels related to rate limits were reported, with two being novel. Ironically, `V_icmplim_curr_jitter`, which was used to introduce randomness to the rate limit counter to patch SADDNS, creates another side channel that forfeits the randomization effort. In the case of lwIP, SCAD discovered a "SYN-backlog-based side-channel", labeled as #13.

For UDP in Linux kernel 6.1.32, SCAD identified two side channels related to the ICMP global rate limit counter (#8 and #9), the foundation of the SADDNS attack. Even though considered patched, we are surprised to find that they can still work in the latest kernel. As will be shown later, the discovered `icmp_global.stamp` can be used to revive the SADDNS attack.

We manually checked the false positive cases. It turns out that they are due to implementation issues. For example, for #6, this false positive arose from the fact that we use memory addresses as unique variable identifiers. However, this approach can lead to ambiguities when two paths dynamically allocate memory after the snapshot is taken. Specifically, even though the addresses of the allocated request socket objects *(i.e.,* `req`) are the same in two paths, they actually refer to different sockets and therefore causes an FP. We will discuss the limitations of SCAD in Section 9.

## 6.3. Comparison with SCENT

We focus on true positives for Linux v4.8 since it is the one evaluated by SCENT, i.e., #1 to #5 in Table 2. Out of the five, #2, #4, and #5 were found by SCENT [13], and #1 and #3 are missed.

We carefully analyzed the reasons these two vulnerabilities are not discovered by SCENT. It turns out that they cannot be found regardless of how much time is given to run the model checker (this is also confirmed by SCENT authors).

For #1, SCENT failed to mark the variable `inet_csk(sk)->icsk_accept_queue->young` as a shared variable during the manual modeling process. In comparison, SCAD does not require marking individual variables and simply considers all heap and global variables as candidate shared variables.

For #3, SCENT cannot find it because it **did not model time** (time is fixed and never progresses in SCENT) and **limited the TCP state to `ESTABLISHED` or `SYN_RECV`**

| # | Target | Shared Variable | TP/FP | Secret[1] | New? | Variable Type |
|---|---|---|---|---|---|---|
| 1 | Linux 4.8 | inet_csk(sk)->icsk_accept_queue->young | TP | CP | Y | Queue Length |
| 2 | Linux 4.8 | tcp_memory_allocated | TP | SN&RN | N | Memory Limit |
| 3 | Linux 4.8 & 6.1.32 | challenge_timestamp | TP | CP&SN&RN | Y | Timestamp |
| 4 | Linux 4.8 & 6.1.32 | inet_csk(sk)->icsk_accept_queue->qlen | TP | CP | N | Queue Length |
| 5 | Linux 4.8 & 6.1.32 | challenge_count | TP | CP&SN&RN | N | Rate Limit |
| 6 | Linux 4.8 & 6.1.32 | req->rsk_rcv_wnd | FP | CP | | |
| 7 | Linux 6.1.32 | skb (output) | FP | CP&SN&RN | | |
| 8 | Linux 6.1.32 UDP | icmp_global.stamp | TP | CP | Y | Timestamp |
| 9 | Linux 6.1.32 UDP | icmp_global.credit | TP | CP | N | Rate Limit |
| 10 | FreeBSD 13.2 | V_icmplim_curr_jitter | TP | CP | Y | Randomness |
| 11 | FreeBSD 13.2 | cr->cr_ticks | TP | CP | Y | Timestamp |
| 12 | FreeBSD 13.2 | cr->cr_rate | TP | CP | N | Rate Limit |
| 13 | lwIP 2.2.0 RC1 | tcp_pcb_listen->accepts_pending | TP | CP | Y | Queue Length |

[1] CP=Client Port #  RN=rcv_nxt  SN=snd_nxt

Table 2: Side channels reported by SCAD

only. If time were allowed to progress in SCENT, the state space would grow substantially, *(i.e.,* different times could correspond to different states). This is why SCENT chooses not to consider time progression.

In Section 7.2, we will also show that most new side channels cannot be found by SCENT due to similar reasons.

Finally, we acknowledge that SCAD requires more computation than SCENT, due to the nature of symbolic execution and the lack of hand-tuned simplification of models. We will analyze the performance of SCAD in more details in Section 6.4.

### 6.4. Performance Analysis

| Target | Paths (k) | # of Symbols | CPU Hours |
|---|---|---|---|
| Linux 4.8 | 2,763 | 527 | 3,089 |
| Linux 6.1.32 | 2,762 | 596 | 1,348 |
| Linux 6.1.32 UDP | 774 | 107 | 1,306 |
| FreeBSD 13.2 | 3,856 | 538 | 1,867 |
| lwIP 2.2.0 RC1 | 33 | 103 | 27 |

Table 3: Statistics of SCAD on different targets

As shown in Table 3, SCAD required 3,089 CPU hours to evaluate the TCP stack of Linux 4.8 kernel. The SSE component consumed 1,248 of these hours, with the remainder attributed to the NIPVC component. For the SSE component, we found the paths explored in the first 234 CPU hours are sufficient for NIPVC component to find all 6 side channels, which means all side channels could have been found in 238.5 CPU hours (including 4.5-CPU-hour NIPVC time). In total, the SSE component explored 2.76M paths with 527 symbolized state variables. As mentioned before, we found NIPVC only took 4.5 CPU hours to find all 6 side channels (including one FP) listed in Table 2 and the remaining 1,836.5h are spent on exploring side-channel possibilities over other variables. This is due to short-circuit evaluation — if two propagation paths are considered to interfere with a shared variable, we will consider this variable a candidate for side channels and ignore all other paths regarding the same variable.

The SSE component explored a similar number of paths for both Linux 4.8 and 6.1.32 within the 48-hour limit. This consistency is expected given the stability of the Linux TCP stack across versions. It is also noticed that Linux 6.1.32 takes a shorter time to finish, and again this is due to short-circuit evaluation used in NIPVC component as the path-pairs are randomly chosen to check for non-interference property violation. For lwIP, the SSE component finished exploring all possible paths in 25.76 CPU hours. This is expected as lwIP is a relatively light weight user-space network stack designed for embedded devices [41] and the TCP implementation of lwIP only consists 8,000 lines of C code. This also proves SCAD is an universal tool that can be applied to user-space targets as well (rather than only kernel targets).

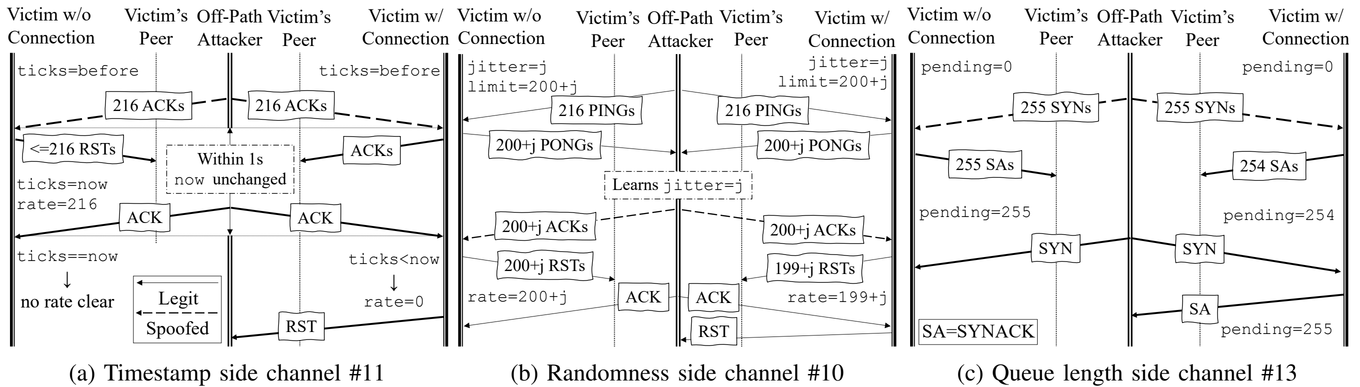
## 7. Case Study

In this section, we provide details on the new vulnerabilities, including their detailed root causes, whether existing methods can find them, and how exploitable they are. Specifically, we categorize the new vulnerabilities into three types and discuss them separately.

### 7.1. Timestamp-Based Side Channels

#3, #8 and #11 in Table 2 are timestamp-based side channels. We successfully identified these timestamp variables as shared state variables because they are marked as symbolic (due to their location in heap), despite the fact that they take different concrete values at runtime. These shared variables are used to measure time and reset rate limit counters, which were previously known to introduce side channels *(i.e.,* #5, #9 and #12, respectively). However, nobody anticipated that these timestamp variables themselves introduce side channels. In fact, such side channels cannot be found by SCENT because of its decision to freeze time in the model checker (and thus not considering timestamps as state variables), as previously discussed in Section 6.3.

**The case for FreeBSD (#11).** FreeBSD limits the rate of outgoing RST to 200 packets per second (pps). Therefore, aside from the counter (cr_rate) counting how many RSTs have been sent within 1s, it also records the last time




(a) Timestamp side channel #11 (b) Randomness side channel #10 (c) Queue length side channel #13

Figure 4: Exploits for newly found side channels on the latest versions of Linux/FreeBSD/lwIP

when the counter was reset as a timestamp (`cr_ticks`). Every time the counter is accessed *(i.e.,* a RST is solicited), the current time will be compared with the timestamp. If the timestamp is more than 1s old, the counter will be reset to 0 and the timestamp itself will be updated to the current time. Previous side-channel attacks [13] leveraged the (`cr_rate`) as the shared variable to infer the source port of an established TCP connection, as the counter value is interfered by port matching result.

Later on the side channel was patched by introducing a random jitter (`V_icmplim_curr_jitter`) to the hard limit 200pps, and now every time the counter needs to be reset, the real limit is calculated by adding the hard limit with the jitter. By default, with the jitter ranging [-16, 16], the real limit ranges between [184, 216]. This destroyed counter-based side channel as the attacker will not predict counter value precisely and thus cannot effectively correlate their observance with the probing result.

Nevertheless, SCAD provides us a new perspective to the same logic by indicating `cr_ticks` can also be used to infer the port match result. According to the output of SCAD, the generation of a RST packet can set `cr_ticks` to the current time (when 1s has elapsed since the last reset) which can leak the port inference result to `cr_ticks`. This is because `cr_ticks` can decide whether the RST transmission will occur or not.

Figure 4a depicts the end-to-end exploit. Since the attacker does not know the exact value of the limit, to make the `cr_rate` meet or exceed the limit, they first send the max possible amount *(i.e.,* 216) of spoofed ACKs to a guessed port. If there is no such a connection on the victim, meaning a port guess was wrong, then RST packets will be solicited and the following events will happen in order: 1) upon receiving the very first ACK packet, `cr_ticks` will be set to now, assuming 1s has elapsed since the last reset of `cr_rate`. `cr_rate` will first be reset to 0 and then incremented to 1 immediately upon generating a RST packet in response; 2) up to an additional 215 RST packets will be generated in response to the attacker's ACK probes, and 3) `cr_rate` will increment to 216 (its value can go beyond the rate limit).

If there is a connection, challenge ACK packets will be returned instead, leaving anything related to RST rate limit unchanged. In other words, `cr_ticks` is updated when the guess of a port is incorrect, and remains unchanged if the guess is correct. To observe such a difference in value of `cr_ticks`, within 1s of sending the previous 216 spoofed ACK probes, the attacker sends one single non-spoofed ACK packet to a known closed port to solicit a RST. If `cr_ticks` was just updated to now, then `cr_rate` will not be reset to 0, and no RST will be sent to the attacker; this is because `cr_rate` is 216 which already exceeds the limit. If `cr_ticks` has not been recently updated, the counter will be reset, and a RST will be sent. By observing the presence or absence of the RST packet, the attacker will learn the result of the guess, indirectly through `cr_ticks`. Note that even if `cr_rate` is patched to be per-IP as opposed to global, `cr_ticks` can still be leveraged to perform this attack, as `cr_ticks` by design is a shared variable itself.

The attack effectively allows an off-path attacker to scan client ports at the speed of 1 port/s, as `cr_ticks` needs to be reset before the next trial.

**The cases for Linux UDP (#8) and TCP (#3).** They are somewhat similar to #11. For the UDP port scan, the only difference is that we would target the ICMP error rate limit (instead of FreeBSD's RST rate limit) which is 20 tokens per 20ms, where each packet consumes 0, 1, or 2 tokens for randomness (as a result of patching previous side channels) [42]. To ensure the limit is reached, an attacker would send 150 spoofed packets per 20ms. Hre, the number 150 is empirically determined. Because a packet may consume 0 tokens, we need to send more than 20 packets to guarantee the consumption of the 20 tokens. Note that since the tokens are reset every 20ms (as opposed to 1s), the effective port scan speed increases to 50 ports per second, which is good enough to revive SADDNS. For the TCP port scan, we would target the challenge ACK rate limit [3]. However, since there is a per-socket challenge ACK rate limit that is much smaller (i.e., two per second), it effectively prevents the attacker from reaching the global limit with spoofed packets.

To demonstrate the revival of SADDNS, we implemented

an end-to-end attack based on timestamp side channel found in Linux UDP stack. Similar to the evaluation setup in SADDNS [1], we set up the attacker, victim resovler and nameserver host on AWS with Linux kernel 6.2.0. The resolver runs unbound 1.13.1. The nameserver has the response rate limit feature enabled and can be leveraged by an attacker to extend the attack window [1]. We repeated the experiment for 5 times and it took 4,586s on average to successfully poison the resolver's cache, which is about 10x slower compared with the original SADDNS. Given the slower probing speed of 50 pps, this result is expected. Note that despite it took 76 minutes to poison the cache, it is still a potent attack because the poisoned cache records can stay for days via a large TTL.

## 7.2. Randomness-based Side Channel

Perhaps the most unexpected side channel is #10, as the jitter (`V_icmplim_curr_jitter`) itself was introduced to mitigate the rate limit counter (`cr->cr_rate`) side channel, as mentioned in Section 7.1. Even though this variable takes a random value at runtime, it is marked as symbolic during our analysis and is considered in SCAD. However, even if SCENT were applied to the same target, it would be unable to discover this side channel, because it explicitly eliminates sources of randomness (by forcing `rand()` to return a fixed value), in order to avoid creating too many states due to these random values.

In FreeBSD, the jitter is reset *(i.e.,* pick another value from [-16, 16]) every time when the counter (`cr_rate`) exceeds the previous limit, and at least 1 second has passed since the last reset. This means that if the limit has been reached in the previous second, the jitter will be reset only when the host generates the next RST in the following second. This seems to be a flawless scheme where an attacker can never learn the jitter from the previous second, and reuse it in the subsequent second. In other words, as soon as the probing starts in the new second, the previously learnt jitter will be immediately reset. Nevertheless, SCAD found that the jitter is actually shared across protocols whereas the counter and timestamp (`cr->cr_ticks`) are not. In other words, it is possible to use one protocol to infer the jitter value without resetting it, and then use it to de-randomize another protocol's rate limit, provided that both occur within the same second.

Figure 4b illustrates the attack process. Since both ICMP echo reply (PONG) and TCP RST packet are rate limited using the same `V_icmplim_curr_jitter`, the attacker first sends non-spoofed 216 (the max possible real rate limit) ping packets to reveal jitter $j$ by counting received ping replies. With the knowledge of $j$, the victim is now vulnerable to the counter-based attacks which can be used to infer the client port number. Similar to [1], [3], [13], the attacker then sends $200 + j$ spoofed probing ACK packets each destined to a different port. If one of the probed port has a connection, $199+j$ RSTs and 1 ACK will be triggered on the victim, and the `cr_rate` will become to $199 + j$. If there is no connection, the `cr_rate` will become to $200+j$ after sending $200+j$ RSTs. To differentiate between these two possible values of `cr_rate`, similar to previous exploits, the attacker sends a legitimate ACK to a closed port and check if they can get the RST back, which depends on the value of `cr_rate`, and thus reveals the probing result. Note the probing result only reveals whether there is a port open among probed ports without specifying the open port number, and therefore binary search should follow [1], [2]. Since `V_icmplim_curr_jitter` is added to FreeBSD to patch SADDNS after 2020, SCENT would not have a chance to discover this side channel. But similar to the timestamps, which were explicitly excluded in [13], randomness were also excluded; therefore it would be impossible to SCENT to uncover this randomness-based side channel.

The exploit enables an off-path attacker to scan the TCP client port at 200 pps to 216 pps. The exploit can also be used for UDP and SCTP ephemeral port inference, as they both share the same `V_icmplim_curr_jitter`. We implemented the PoC exploit without binary search. By limiting the port scan range to 5000 ports, it can correctly figure out the correct port number range for 10 out of 10 times.

## 7.3. Queue-Length-Based Side Channels

Unlike temporal side channels discussed in Section 7.1 & 7.2, #1 and #13 are spatial side channels that leverage the limited queue size for half-open TCP connections.

Taking #13 as an example, in lwIP, `accepts_pending` represents the length of the backlog queue that stores the new TCP connection requests that have not been finished *(i.e.,* half-open). The queue belongs to a listen socket that can be accessed by any host, and the queue length will increase by one when the socket received a SYN packet, and decrease by one when 3-way handshake is finished. If the queue length exceeds the maximum backlog limit, then new SYN packet will be dropped to prevent DoS attacks. Similarly, by observing whether the queue is full, the attacker can learn whether the client port guessed is correct or not.

Figure 4c shows the exploit of #13. Since in lwIP, the limit of the backlog queue is 255 by default, the attacker sends 255 probing SYN packets with each destined to a different port. If none of the probed port has a connection, the backlog queue will be saturated with half-open connections *(i.e.,* `accepts_pending` becomes 255), as the SYN will be treated as new connection requests, and the attacker never sends ACKs to finish the 3-way handshake. If one of the probed port has a connection, there will be one remaining slot in the queue *(i.e.,* `accepts_pending` becomes 254), as one of the SYN packet will solicit an challenge ACK of the existing connection. To detect the difference in `accepts_pending`, the attacker tries to inject another half-open connection by sending a legitimate (non-spoofed) SYN packet, and if they can receive the SYNACK reply, meaning the queue is not full and thus a connection is found and otherwise it is not found. Similarly, a binary search should follow to pinpoint the exact open port.

This side channel allows an attacker to scan 255 client ports in 20s, which is 12.75 ports per second. This is because of the 20-second purging interval of backlog queue of lwIP. We implemented the PoC exploit without binary search. By limiting the port scan range to 1,000 ports, it can correctly figure out the correct port number range for 10 out of 10 times.

Similarly, #1 represents the number of half-open connections of a listen socket plus the number of orphaned half-open connection that the listen socket has received. The difference is the default limit of backlog queue is 128 in Linux kernel v4.8. Besides, to trigger the side channel, an additional pre-condition must also be true, namely the *accept queue*, which stores TCP connection requests that have finished 3-way handshake but have not been accepted by the user-space application yet, must be full. The default size of the *accept queue* is also 128 in Linux kernel v4.8. Given most applications will accept connections as soon as they can, unless the system is under load, it is unlikely to make it saturated in order to trigger the side channel. We thus consider #1 to be unlikely exploitable.

### 7.4. Responsible Disclosure

We reported the discovered side channels to the Linux, FreeBSD and lwIP maintainers. At the time of writing, Linux kernel maintainers acknowledged the vulnerabilities and a patch has been released after our discussion [43]. In particular, the patch fixes the most critical vulnerability (#8 in Table 2). It is available on Linux 6.12-rc1 and onwards, as well as the majority of LTS versions. The patch works by checking the per-IP rate limit before the global rate limit, at the cost of slightly lower performance under heavy load. FreeBSD maintainers acknowledged our report and is actively working with Netflix team for the patch. Unfortunately, we have yet to hear back from lwIP maintainers.

## 8. Mitigation

There are two common approaches in mitigating side channels: (1) inject randomness into the values of shared variables (as has been done in both Linux and FreeBSD to fix prior side channels); (2) avoid the sharing of variables and instead isolate them into per-IP or per-socket variables.

Either approach has pros and cons. Previously, it was widely accepted that randomization is a reasonable solution. However, our findings reveal that it can be error-prone to implement the added randomness correctly — `icmp_global.stamp` still allowed port scan at 50pps and `V_icmplim_curr_jitter` is ineffective because it is shared and can be easily leaked. Isolation addresses the root cause of side channels (e.g., no longer making such variables shared across connections/IPs), but it may not allow a global control of resource uses, e.g., a per-socket RST rate limit can still allow a large number of RST packets in aggregate over many connections.

Therefore, we propose the following best practices:

(1) Consider integrating the per-socket or per-IP rate limit together with a randomized global rate limit. The former gives a stronger security guarantee while the latter still allows the global control.

(2) The global rate limit should be larger than the per-socket or per-IP rate limit.

(3) The randomness introduced in the global rate limit should not be inferable; in other words, the variable that represents the jitter should not be leaked or shared itself.

As an example, to patch the timestamp-based side channels (#3, #8, and #11) and rate-limit-counter-based side channels (#5, #9, and #12), their sharing scope can be minimized by transitioning to a per-IP equivalent. This means that the timestamp or rate-limit variables will no longer interfere with attacker-observables, i.e., an off-path attacker cannot influence such variables and observe their effects (as they are no longer global). A global counterpart can be retained, provided it adheres to the aforementioned three criteria.

Similarly, to patch the queue-length-based side channels, one can first introduce a per-IP rate limit where each client IP may create up to certain number of half-open connections; on top of it, a randomized global rate limit can be imposed on all client IPs to limit the total resource usage.

## 9. Discussion and Limitations

**Generality.** `SCAD` is a general solution that requires minimal manual effort to set up and use. Nevertheless, it is still a tool that is designed for security analysts or researchers who are familiar with the concept of network side channels. This is partly due to the nature of side channels where at least the secrets and attacker-observables have to be defined a priori. In this paper, we choose open-source protocol implementations as our target protocols because they are relatively easy to understand and annotate. In theory, `SCAD` can also be applied to closed-source protocol implementations such as those in Windows, as it operates on top of S2E which is in turn built on top of a general virtualization platform, i.e., QEMU [37]. However, it requires significant reverse engineering effort to model and annotate closed-source programs. We thus leave them to future work.

**Analysis scope selection.** While `SCAD` offers flexibility in adjusting the execution range, determining the optimal range can require some expertise of network protocols to be analyzed. An overly broad range might cause `SCAD` to waste time on irrelevant logic, leading to scalability challenges. Conversely, a narrow range might overlook certain side channels if the vulnerable code executes concretely, resulting in missed vulnerabilities. In our solution, we include all functions of the target protocol, as well as functions in dependent protocols based on our understanding of their frequent interactions. In theory, we can even prune certain functions in the target protocol that are clearly not relevant (if any), to speed up the analysis. However, that would require additional human effort, which is why `SCENT` takes 2.5 weeks to complete the model construction [13]. Unlike

SCENT, SCAD does not perform such fine-grained decision and instead use coarse-grained memory address ranges for an entire module *(e.g.,* TCP or ICMP).

**Exploration strategy during symbolic execution.** We have experimented with depth-first search (DFS) and breadth-first search (BFS) during the symbolic execution when forking different states for exploring different execution paths. What we find is that BFS is much more desirable than DFS. This is because DFS often gets stuck in exploring deep parts of the control flow graph, where the vulnerable code is not necessarily located. In contrast, according to the previous network side channels, most issues are relating to error handling logic, which is triggered relatively early. In fact, we find that most vulnerable logic is uncovered early in BFS. Taking the Linux kernel v4.8 as an example, we find that it took only 4.5 CPU hours to find all 6 side channels (including the one FP). This gives us hope in analyzing even more complex protocols.

**Potential concerns on selective symbolic execution.** In addition to the issues on selective symbolic execution mentioned previously, there are two other concerns worth mentioning. First, symbolizing state variables assumes a state variable can take any arbitrary value, which may not always be true. This can lead to the exploration of non-existent system states and thus false positives. For instance, in Table 2, the 32-bit variable `icmp_global.credit` of #9 will never surpass 50 according to the rate limit logic. Yet, SCAD assumes it can adopt any valid 32-bit integer value. If a side channel is contingent on this counter exceeding 50, it will result in a false positive (FP).

Second, switching between symbolic execution and concrete execution can lead to erroneous concretization. Specifically, erroneous concretization pertains to situations where an under-constrained variable, due to relaxed constraints, might be concretized to a value that is only locally or statically feasible, but globally infeasible *(e.g.,* concretize `icmp_global.credit` to 51). This not only risks identifying side channels in unreachable states but can also cause system crashes, for instance, by concretizing a pointer that was just symbolized, to 0. The two false positives reported in Table 2 were due to this reason. In theory, this can be circumvented by forking and concretizing the variable for each feasible value under the current constraint. However, such a brute-force approach will negate the benefits of concrete execution. To address this, SCAD employs heuristics, such as avoiding the symbolization of pointers.

## 10. Related Work

**Side-channel attacks.** Network side channels has a rich history, though it has not attracted widespread attention [1], [2], [3], [4], [5], [6], [22], [23], [24], [44], [45], [46], [47], [48]. For instance, the IPID side channel has been known since 2007 [24]. Since then, a number of research studies have considered using various shared variables in network protocols, most notably in TCP and UDP to infer port numbers and TCP sequence numbers. For example, [46] harnessed global counters to discern open ports. The work in [3] employed the global challenge ACK counter to inject segments to a TCP connection. The study in [6] successfully compromised the WPA-TKIP group cipher suite of Wi-Fi by exploiting the side channel stemming from the shared power save state in the Linux Kernel. In the domain of UDP, several studies [1], [2] revealed ICMP global rate limit counter and next hop exception cache as shared variables that enable DNS cache poisoning attacks.

**Automated side-channel detection & prevention.** The realm of automated side-channel detection has seen various methodologies. PacketGuardian [14] employed static analysis to identify ”implicit information leakage”, essentially a form of side channel. However, it yielded a significant number of false positives (FPs) and was limited to detecting leaks to statistical counters, which remain inaccessible to off-path attackers. An enhancement over this was presented in [49], which improved upon [14] by focusing on leaks observable to attackers, *(i.e.,* packet outputs), but it still grappled with a high FP rate. The study in [13] adopted model checking to pinpoint side channels within the TCP stack. However, as delineated in Section 3, its versatility remains limited. On the preventive front, [50] introduced a novel approach to ensure non-interference property at the programming language level by integrating new notations for Java. While promising, this method demands rewriting existing applications under the new framework, posing challenges for its widespread adoption.

## 11. Conclusion

In this work, we introduced SCAD, a first general, systematic, and automated tool capable of finding network side channels in real-world protocol implementations. By carefully applying selective symbolic execution that works on live targets, we manage to identify 17 side channels, with 14 being true positives. Notably, 7 of these vulnerabilities were never discovered before. We believe the tool can be reused to check the future protocols and their implementations.

## Acknowledgement

We sincerely thank the anonymous reviewers for their insightful comments, especially the follow-up engagements during the interactive rebuttal period. This research was sponsored by the National Science Foundation under Grant No. #1652954.

## References

[1] K. Man, Z. Qian, Z. Wang, X. Zheng, Y. Huang, and H. Duan, “Dns cache poisoning attack reloaded: Revolutions with side channels,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1337–1350. [Online]. Available: https://doi.org/10.1145/3372297.3417280

[2] K. Man, X. Zhou, and Z. Qian, "Dns cache poisoning attack: Resurrections with side channels," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 3400–3414. [Online]. Available: https://doi.org/10.1145/3460120.3486219

[3] Y. Cao, Z. Qian, Z. Wang, T. Dao, S. V. Krishnamurthy, and L. M. Marvel, "Off-Path TCP exploits: Global rate limit considered dangerous," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 209–225. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/cao

[4] W. Chen and Z. Qian, "Off-path tcp exploit: How wireless routers can jeopardize your secrets," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1581–1598.

[5] Y. Cao, Z. Qian, Z. Wang, T. Dao, S. V. Krishnamurthy, and L. M. Marvel, "Off-path tcp exploits of the challenge ack global rate limit," *In IEEE/ACM Transactions on Networking (TON)*, 2018.

[6] D. Schepers, A. Ranganathan, and M. Vanhoef, "Practical side-channel attacks against wpa-tkip," ser. Asia CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 415–426. [Online]. Available: https://doi.org/10.1145/3321705.3329832

[7] C. Ma, D. Wu, G. Tan, M. T. Kandemir, and D. Zhang, "Quantifying and mitigating cache side channel leakage with differential set," *Proc. ACM Program. Lang.*, 2023.

[8] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. Kandemir, "Casym: Cache aware symbolic execution for side channel detection and mitigation," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 505–521.

[9] G. Doychev, D. Feld, B. Kopf, L. Mauborgne, and J. Reineke, "CacheAudit: A tool for the static analysis of cache side channels," in *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, Aug. 2013, pp. 431–446. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/doychev

[10] G. Doychev and B. Köpf, "Rigorous analysis of software countermeasures against cache attacks," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 406–421.

[11] D. Wang, A. Neupane, Z. Qian, N. B. Abu-Ghazaleh, S. V. Krishnamurthy, E. J. Colbert, and P. Yu, "Unveiling your keystrokes: A cache-based side-channel attack on graphics libraries," in *NDSS*, 2019.

[12] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "CacheD: Identifying Cache-Based timing channels in production software," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[13] Y. Cao, Z. Wang, Z. Qian, C. Song, S. V. Krishnamurthy, and P. Yu, "Principled unearthing of tcp side channel vulnerabilities," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 211–224. [Online]. Available: https://doi.org/10.1145/3319535.3354250

[14] Q. A. Chen, Z. Qian, Y. J. Jia, Y. Shao, and Z. M. Mao, "Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 388–400. [Online]. Available: https://doi.org/10.1145/2810103.2813643

[15] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy*, 1982, pp. 11–11.

[16] G. Smith, "Principles of secure information flow analysis," in *Malware Detection*, M. Christodorescu, S. Jha, D. Maughan, D. Song, and C. Wang, Eds. Boston, MA: Springer US, 2007, pp. 291–307.

[17] T. BALL, J. DANIEL, and T. Ball, "Deconstructing dynamic symbolic execution," Tech. Rep., 2015.

[18] X. Feng, Q. Li, K. Sun, Z. Qian, G. Zhao, X. Kuang, C. Fu, and K. Xu, "Off-Path network traffic manipulation via revitalized ICMP redirect attacks," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 2619–2636. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/feng

[19] D. Schepers, A. Ranganathan, and M. Vanhoef, "Practical side-channel attacks against wpa-tkip," ser. Asia CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 415–426. [Online]. Available: https://doi.org/10.1145/3321705.3329832

[20] X. Feng, C. Fu, Q. Li, K. Sun, and K. Xu, "Off-path tcp exploits of the mixed ipid assignment," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20, 2020.

[21] "Off-Path attacking the web," in *6th USENIX Workshop on Offensive Technologies (WOOT 12)*. Bellevue, WA: USENIX Association, Aug. 2012. [Online]. Available: https://www.usenix.org/conference/woot12/workshop-program/presentation/Gilad

[22] Z. Qian and Z. M. Mao, "Off-path tcp sequence number inference attack-how firewall middleboxes reduce security," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 347–361.

[23] Z. Qian, Z. M. Mao, and Y. 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*, ser. CCS '12, 2012.

[24] lkm, "Blind tcp/ip hijacking is still alive," http://phrack.org/issues/64/13.html, 2007.

[25] "Sad dns explained," https://blog-cloudflare-com.webpkgcache.com/doc/-/s/blog.cloudflare.com/sad-dns-explained.

[26] "Sad dns - side channel attack," https://www.isc.org/blogs/2020-saddns/.

[27] T. Dai and H. Shulman, "Smap: Internet-wide scanning for spoofing," in *Annual Computer Security Applications Conference*, ser. ACSAC '21, 2021.

[28] "Non-interference (security)," https://en.wikipedia.org/wiki/Non-interference_(security).

[29] D. King, B. Hicks, M. Hicks, and T. Jaeger, "Implicit flows: Can't live with 'em, can't live without 'em," in *Proceedings of the 4th International Conference on Information Systems Security*, ser. ICISS '08. Berlin, Heidelberg: Springer-Verlag, 2008, p. 56–70. [Online]. Available: https://doi.org/10.1007/978-3-540-89862-7_4

[30] L. Lampropoulos, M. Hicks, and B. C. Pierce, "Coverage guided, property based testing," in *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*, Oct. 2019.

[31] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: a platform for in-vivo multi-path analysis of software systems," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, R. Gupta and T. C. Mowry, Eds. ACM, 2011, pp. 265–278. [Online]. Available: https://doi.org/10.1145/1950365.1950396

[32] Cyberhaven, "Symbolic execution of linux binaries — s2e 2.0 documentation," https://s2e.systems/docs/Tutorials/BasicLinuxSymbex/s2e.so.html\#\#what-about-other-symbolic-input, 2018.

[33] V. Kuznetsov, V. Chipounov, and G. Candea, "Testing Closed-Source binary device drivers with DDT," in *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. USENIX Association, Jun. 2010. [Online]. Available: https://www.usenix.org/conference/usenix-atc-10/testing-closed-source-binary-device-drivers-ddt

[34] V. Chipounov and G. Candea, "Reverse engineering of binary device drivers with revnic," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 167–180. [Online]. Available: https://doi.org/10.1145/1755913.1755932

[35] Z. Wang, S. Zhu, Y. Cao, Z. Qian, C. Song, S. V. Krishnamurthy, K. S. Chan, and T. D. Braun, "Symtcp: Eluding stateful deep packet inspection with automated discrepancy discovery," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. [Online]. Available: https://dx.doi.org/10.14722/ndss.2020.24083

[36] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Journal of Computer Security*, vol. 18, no. 6, pp. 1157–1210, 2010.

[37] "qemu," https://github.com/qemu/qemu.

[38] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[39] Z. Wang, S. Zhu, K. Man, P. Zhu, Y. Hao, Z. Qian, S. V. Krishnamurthy, T. La Porta, and M. J. De Lucia, "Themis: Ambiguity-aware network intrusion detection based on symbolic model comparison," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 3384–3399. [Online]. Available: https://doi.org/10.1145/3460120.3484762

[40] "tcp_v4_rcv()," https://github.com/torvalds/linux/blob/v6.1/net/ipv4/tcp_ipv4.c#L1924, 2022.

[41] "lwip," https://en.wikipedia.org/wiki/LwIP.

[42] "SAD DNS patch," https://github.com/torvalds/linux/commit/b38e7819cae946e2edf869e604.

[43] E. Dumazet, "icmp: change the order of rate limits," https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8c2bd38b95f75f3d2a08c93e35303e26d480d24e, 2024.

[44] G. Alexander and J. R. Crandall, "Off-path round trip time measurement via tcp/ip side channels," in *2015 IEEE Conference on Computer Communications (INFOCOM)*, 2015.

[45] J. Knockel and J. R. Crandall, "Counting packets sent between arbitrary internet hosts," in *4th USENIX Workshop on Free and Open Communications on the Internet (FOCI 14)*. San Diego, CA: USENIX Association, Aug. 2014.

[46] R. Ensafi, J. C. Park, D. Kapur, and J. R. Crandall, "Idle port scanning and non-interference analysis of network protocol stacks using model checking," in *Proceedings of the 19th USENIX Conference on Security*, ser. USENIX Security'10. USA: USENIX Association, 2010, p. 17.

[47] Z. Qian, Z. M. Mao, Y. Xie, and F. Yu, "Investigation of triangular spamming: A stealthy and efficient spamming technique," in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 207–222.

[48] A. Quach, Z. Wang, and Z. Qian, "Investigation of the 2016 linux tcp stack vulnerability at scale," *Proc. ACM Meas. Anal. Comput. Syst.*, 2017.

[49] K. Ru, Y. Zheng, X. Feng, and D. Wang, "The side-channel vulnerability in network protocol," in *Proceedings of the 2021 11th International Conference on Communication and Network Security*, ser. ICCNS '21. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–8. [Online]. Available: https://doi.org/10.1145/3507509.3507510

[50] J. Xiang and S. Chong, "Co-inflow: Coarse-grained information flow control for java-like languages," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 18–35.


# Appendix A. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

## A.1. Summary

This paper presents a symbolic-execution-based approach, called SCAD, to detect side-channel vulnerabilities in the implementation of network protocols. SCAD relies on manual efforts to annotate secrets, attacker-observables, and analysis scope. SCAD is a selective symbolic execution, which chooses a subset of variables to symbolize and limits symbolic execution in a small scope via concolic execution.

## A.2. Scientific Contributions

- Identifies an Impactful Vulnerability
- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field

## A.3. Reasons for Acceptance

1) This paper found 6 new vulnerabilities in 5 benchmarks, while also finding vulnerabilities that bypass patches for previously known (and fixed) side-channel vulnerabilities.
2) This paper expands the area of automated detection of side-channel vulnerabilities by directly comparing against prior work.
3) This paper will open-source the system, thus allowing future researchers to build on this work.