

TCP Cookies in the Linux Kernel

Sebastian Almeida Henriques da Silva, Lion Steger*

**Chair of Network Architectures and Services*

School of Computation, Information and Technology, Technical University of Munich, Germany

Email: sebastian.silva@tum.de, stegerl@net.in.tum.de

Abstract—This paper examines TCP SYN cookies as a defence against backlog exhaustion during connection setup. The central question is whether any server-side state is created while cookie mode is active before the final ACK is verified. We also outline two secondary issues, namely the feasibility of computing a valid cookie in advance for a given connection key and the CPU cost of cookie processing at high connection attempt rates. We analyze the Linux 6.16 TCP stack by tracing the passive open path and the cookie routines, and we validate the readings with kernel counters under controlled floods. We find that Linux keeps no persistent per-connection state between the initial SYN and the validated final ACK while cookie mode is active. In an off-path model, blind one-round-trip completions are implausible, since validation reduces to an effective 24-bit search per minute, and kernel exploits are likewise implausible as they require full system control. The per-attempt hashing work is small compared to packet processing in the network stack, which remains the practical bottleneck.

Index Terms—TCP syn cookies, syn flooding, listen backlog exhaustion, Linux kernel

1. Introduction

Transmission Control Protocol is the dominant reliable transport on today’s Internet. It provides ordered delivery, congestion control and connection management by means of a three-way handshake. During this exchange, the server retains a provisional state after receiving a SYN and before the final ACK arrives [1]. The design is robust to ensure reliability, but it exposes a simple vector for denial of service. The technical document RFC 4987 describes the effect succinctly: “The SYN flooding attack is a denial of service method affecting hosts that run TCP server processes” [2]. By forcing the listener to keep many half-open entries, an attacker can exhaust the finite backlog and block legitimate clients [2].

SYN cookies were introduced in the 1990s to keep services available under flood conditions. Instead of reserving a backlog entry on a SYN, the listener encodes verification data into its initial sequence number and postpones allocation until the final acknowledgement. The original note states the intended outcome plainly: a server using cookies does not “have to drop connections when its SYN queue fills up”, at the cost of rejecting some TCP options [3]. Modern kernels enable cookies automatically under stress, for example in Linux [4].

This paper revisits SYN cookies in the Linux kernel and addresses three research questions: (i) Does a listener

create any server-side state between the initial SYN and the verified final ACK while cookie mode is active (see methodology §3.1 and evaluation §4.1). (ii) Is it feasible to compute a valid cookie in advance for a chosen connection key and a time counter so that the handshake can be completed with only the final acknowledgement (see methodology §3.2 and evaluation §4.2). (iii) What CPU cost do cookie generation and validation introduce at high connection attempt rates? (see methodology §3.3 and evaluation §4.3).

The next section summarizes the attack surface and the defence, after which we discuss these questions in turn.

2. Background and Related Work

This section recalls the TCP parts relevant for connection setup and the SYN-flood vulnerability, then summarizes SYN cookies as implemented in modern systems.

2.1. How TCP works

TCP segments carry ports, sequence numbers, acknowledgements, flags, and options in a fixed header with optional extensions [1]. Figure 1 shows only the fields relevant for the handshake and option negotiation.

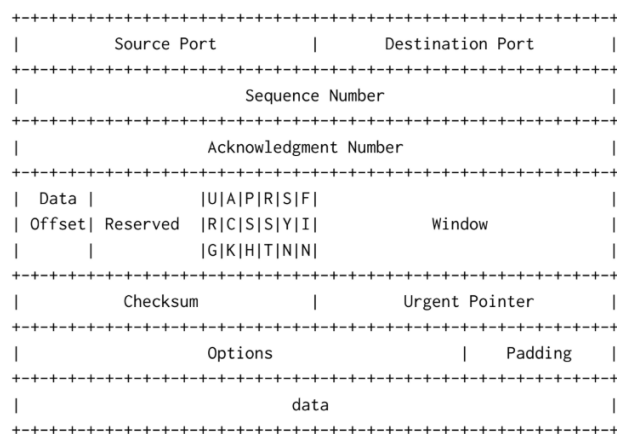


Figure 1: TCP header fields relevant to connection setup and options [5].

A new connection uses the three-way handshake: the client sends a SYN with initial sequence number x ; the server replies with SYN-ACK and its initial sequence number y ; the client completes by acknowledging $y + 1$. We refer back to this standard handshake in what follows; a sketch is in Figure 2.

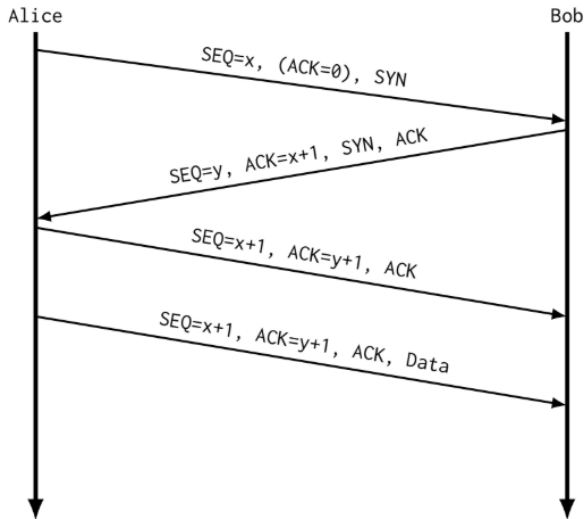


Figure 2: TCP three-way handshake. The server allocates temporary state after the SYN, which leaves the backlog vulnerable to flooding [5].

Two properties matter for both attack and defence: a listener must remember enough to validate the final ACK, and the listen backlog is finite. Too many half-open entries delay or drop new connections.

2.2. SYN floods: the attack

A SYN flood sends many SYNs, often with spoofed sources, and never completes the handshake. Each SYN forces the server to allocate a small temporary connection object (called a *request socket*), which represents state kept on the server side until the handshake either completes or times out. Enough fake entries fill the backlog and block real clients [2]. Figure 3 shows the effect.

Threat model and scope: We consider an off-path attacker, meaning an attacker that can send large volumes of spoofed SYN segments and so-called *blind* final ACKs. By blind, we mean that the attacker cannot see the server's replies and therefore must guess values such as sequence numbers without feedback. We study the IPv4 passive open path, the sequence of kernel functions that handle incoming SYN segments on a listening socket. In Linux, this path includes logic to create or validate a temporary request socket depending on backlog state, followed by decision routines that determine whether to enable cookie mode. Our analysis is based on Linux 6.16 with timestamps enabled and cookie mode triggered either by backlog overflow or forced via `net.ipv4.tcp_syncookies = 2`. Our criterion for “no state” is the absence of any persistent per-connection object (such as a `request_sock`) between the SYN and the validated final ACK in either the request queue or the accept queue. On-path adversaries, middlebox interference, and transport features that require full option negotiation are out of scope for this paper.

2.3. SYN cookies: the defence

SYN cookies remove server state during the vulnerable phase. When cookie mode is active, the server encodes

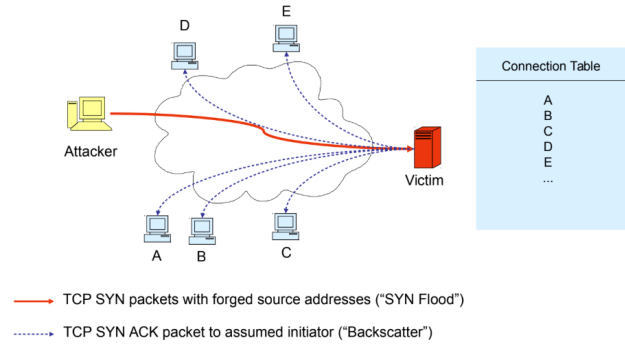


Figure 3: Backlog exhaustion via half-open connections [5].

the minimum needed information into its initial sequence number, replies with SYN-ACK, and defers allocation until the final ACK [3].

Operationally, a cookie is a keyed function of the four-tuple, a coarse time counter, the peer's initial sequence number (ISN), and a compact encoding of options (e.g., MSS). On the final ACK, the server recomputes and validates the cookie and recovers the encoded option. This protects the backlog at the cost of extra hashing and limits on option negotiation [3], [4]. The Linux kernel reflects these choices in `net/ipv4/syncookies.c` [4], and RFC 4987 discusses pros and cons in the wider mitigation set [2].

2.4. Cookie construction in Linux

Linux TCP SYN cookies are a keyed function of the four-tuple, a coarse time counter, and the peer's initial sequence number, with compact encodings for a minimal option set. The implementation uses a small constant number of SipHash-2-4 invocations and integer combination in `net/ipv4/syncookies.c`.

Inputs: Let $(s_{addr}, d_{addr}, s_{port}, d_{port})$ be the four-tuple, let x be the peer ISN from the SYN, and let t be a coarse time counter with about one-minute steps. The code computes a keyed hash over these values and combines it with the peer ISN and the time counter to form the server cookie. The source comment summarizes the construction succinctly: “Compute the secure sequence number: `HASH(...) + sseq + (count * 224)`” [4]. Here `count` is the minute counter, and the multiplication by 2^{24} shifts the counter into the top bits of the server ISN.

Option packing: The MSS is encoded by index into a fixed table declared in the source as “`static __u16 const msstab[] = {536, 1300, 1440, 1460}`” [4]. When TCP timestamps are enabled, Linux packs a compact option summary into the low bits of the SYN-ACK timestamp. The code comment states: “TCP Timestamp: 6 lowest bits of timestamp sent in the cookie SYN-ACK stores TCP options.” [4]. These six bits comprise a four bit window scale field where the value `0xF` signals no scaling, plus one bit each for SACK permission and ECN capability [4]. If timestamps are not in use, only the limited MSS choice is retained.

Creation and validation: While cookie mode is active, the listener does not allocate a request queue entry on

receipt of the SYN. It replies with a SYN-ACK whose ISN is the cookie defined above. On the final ACK, the kernel recomputes and age checks the cookie, decodes the MSS index and the timestamp option bits, and only then creates the child socket on the accept path. The reconstruction occurs in `cookie_v4_check()` and the child is created by `tcp_get_cookie_sock()` [4].

3. Methodology

To answer our research questions, we now outline the steps we took to trace the kernel code paths, design our experiments, and build a simple cost model.

3.1. Server-side state under SYN cookies

To verify whether Linux TCP SYN cookies avoid server-side state creation during the half-open phase, we performed a code-level analysis of the upstream Linux kernel TCP stack. We focused on the IPv4 passive open path and the SYN-cookie implementation. Specifically, we traced the control flow on receipt of a SYN segment through `tcp_conn_request()` in `net/ipv4/tcp_input.c`, the flood-handling decision in `tcp_syn_flood_action()`, and the cookie construction and validation routines in `net/ipv4/syncookies.c`. We recorded the conditions under which SYN cookies are activated, the exact constants used in cookie encoding, and whether a `request_sock` and associated queue entries are retained prior to the final ACK. Our criterion for “no state” is the absence of any persistent per-connection object in the SYN (request) queue or accept queue between the SYN and the final ACK.

3.2. Feasibility of a one-RTT handshake by cookie prediction

Method overview: We distinguish two possible avenues to attempt a blind one-RTT handshake.

(1) **Brute force guessing:** The attacker repeatedly issues blind final ACKs, each with a guessed cookie value. Since validation checks a 2^{24} space within the active one-minute counter bucket, this reduces to a probabilistic search problem.

(2) **Kernel compromise:** The cookie value is derived from a per-boot secret, declared in `syncookie_secret` in `net/ipv4/syncookies.c`. This key is randomized at boot and never exported to user space. Only by modifying kernel code (e.g., replacing the random key with a fixed constant and rebuilding the kernel) could an attacker fully control cookie generation. Such a step presupposes privileged access and is therefore equivalent to total host compromise.

Simulation setup: We forced cookie mode with `net.ipv4.tcp_syncookies=2`. A local listener bound to a fixed port while kernel counters were tracked with `nstat` and socket states monitored with `ss`. For brute force, experiments aligned to the start of a fresh minute and then issued N blind final ACKs per bucket. For kernel modification, we rebuilt a Linux kernel after replacing the secret with a constant, which allows deterministic reproduction of cookies for given four-tuples.

3.3. Analytical cycle cost model

When SYN cookies are active, Linux replaces request queue allocation with keyed hash work. On receipt of a SYN, the listener performs two calls to `siphash_4u32`; on the final ACK, it performs two more and reconstructs the minimal context [4]. Let H be the cycle cost of a single `siphash_4u32` on short inputs. Published timings report about 140 cycles for 16 bytes on an AMD FX 8150 class CPU, with similar magnitudes on contemporary cores [6]. We model

$$C_{\text{SYN}} \approx 2H + c_{\text{syn}},$$

$$C_{\text{ACK}} \approx 2H + c_{\text{ack}},$$

$$C_{\text{total}} \approx 4H + c,$$

where c_{syn} and c_{ack} are constant bookkeeping costs such as option handling and MSS lookup, independent of backlog size.

4. Evaluation

This section reports the outcome of our analysis. We examine when SYN cookies are activated, whether blind one-RTT completions are feasible, and the computational cost introduced by cookie processing.

4.1. Server-side state under SYN cookies

Activation conditions: On receipt of a SYN, `tcp_conn_request()` consults `net.ipv4.tcp_syncookies`. If set to 2, the listener unconditionally operates in cookie mode; if set to 1, cookie mode is entered only when the SYN request queue is full. Queue fullness is tested via `inet_csk_reqsk_queue_is_full(sk)`, which “returns `inet_csk_reqsk_queue_len(sk) > READ_ONCE(sk->sk_max_ack_backlog)`” [7]. The function `tcp_syn_flood_action()` is documented to “Return true if a syncookie should be sent” [8] and finalizes the decision to send cookies. This matches the documented semantics of `tcp_syncookies` (0 disabled, 1 on overflow, 2 always).

State retention prior to ACK: When cookie mode is active, the kernel initializes a temporary `request_sock` (256 bytes on a standard x86_64 build) to compute the SYN-ACK, but does not attach it to the listener or insert it into the SYN queue. Immediately after transmitting the SYN-ACK, the temporary request object is freed, and control returns without queuing any per-connection state [4]. Only upon receipt of a valid final ACK does `cookie_v4_check()` reconstruct sufficient context and `tcp_get_cookie_sock()` either create the child socket or free the request and return NULL, at which point state is materialized and the usual accept path proceeds [4].

Implications: Under SYN cookies, the request queue occupancy is held constant during a flood, since no half-open entries are retained. The design avoids memory exhaustion from unacknowledged SYNs at the cost of reduced option fidelity (limited MSS set and option encoding) and the known protocol trade-offs for SYN cookies. Based on our code inspection the amount of server-side persistent state created per incoming SYN before a valid ACK is zero.

4.2. Feasibility of cookie prediction within one RTT

Observed result: In all blind trials we observed no sockets in ESTAB on the listener port. ESTAB is the TCP “connected” state reached only after a full three-way handshake, so its absence means no successful prediction occurred. We did see resets increase, which is expected when the final ACK carries a wrong cookie. Normal (non-blind) connections behaved as usual.

Kernel mod check: We rebuilt the kernel with a fixed `syncookie_secret` and disabled the one time randomization so that cookies are deterministic for a given four tuple and minute. We then used a small user space script that starts a local server, captures one handshake on loopback, and computes a predicted cookie from the observed addresses, ports, and an MSS index. With `tcp_syncookies=2`, the predicted value differed from the kernel cookie; therefore, a one RTT handshake did not follow. We verified the tuple, the minute counter, the address word order, and the setting of `tcp_syncookies`, and we repeated the run many times with the same result. It may work in theory, but after numerous attempts we did not find a way to make this approach succeed.

Probability model: Each blind guess has probability $p = 2^{-24}$. After N guesses in one minute, $P(\text{at least one hit}) \approx 1 - e^{-N/2^{24}}$. Thousands of guesses yield essentially zero hits; about 1.16×10^7 per minute are needed for 50 %, and $\approx 5.0 \times 10^7$ for 95 %. This matches the negative result above.

```
static siphash_aligned_key_t syncookie_secret[2] = {
    {.key = {4242ULL, 2121ULL}},
    {.key = {4242ULL, 2121ULL}}
};
```

Figure 4: Kernel built with a fixed cookie secret for the determinism check.

4.3. Cost model results

Instantiating the model with $H \approx 140$ gives $C_{\text{total}} \approx 560$ cycles per completed attempt, which is about 0.16 to $0.28 \mu\text{s}$ on 3.5 to 2.0 GHz CPUs, plus the constant terms [6].

Throughput bound from hashing alone: For a core with clock f , hashing alone allows roughly $f/(4H)$ attempts per second. With $f = 3 \text{ GHz}$ and $H = 140$, this is about 5.3×10^6 attempts per second.

What actually dominates: Moving and interpreting packets are the slow part in practice. Empirical analysis of the Linux host network stack shows that packet I/O and data movement dominate CPU time at high rates, rather than small per packet arithmetic [9].

Scaling considerations: Since stack processing and memory traffic dominate, throughput scales mainly with the number of cores that can process packets and with NIC queue capacity, while the small per packet hash cost has only marginal influence on end to end throughput [9].

5. Conclusion and Future Work

This paper examined TCP SYN cookies in Linux 6.16 as a defence against backlog exhaustion during connection setup. Tracing the passive open path and the cookie

routines shows that, while cookie mode is active, the listener retains no persistent per-connection state between the initial SYN and the validated final ACK [4], [8]. A temporary request object used to craft the SYN-ACK is freed without being queued, and state is materialized only after a valid cookie is acknowledged [8]. We described the cookie construction as a keyed function of the connection four-tuple, the peer ISN, and a minute granularity counter, with compact option encodings via an MSS table and timestamp bits [4]. In an off-path model, blind one-RTT completions are implausible: validation reduces to an effective 2^{24} search per minute bucket [4], implying about 1.16×10^7 guesses for a 50 % success chance. Similarly, a kernel exploit requires full system control and is therefore equally implausible, albeit in a different sense. A simple micro-cost model indicates that the added hashing work is on the order of a few hundred cycles per attempt [6] and is unlikely to dominate end-to-end processing in the stack.

Future Work: Useful extensions include measuring costs and limits under real workloads and hardware, extending the analysis to IPv6 and different timestamp or option configurations, and studying interactions with NIC offloads, SYN proxying, and middleboxes. It would also be valuable to quantify the practical impact of reduced option fidelity under floods and to evaluate secret rotation and time-bucket sizes against active attackers.

References

- [1] J. Postel, “Transmission Control Protocol,” Internet Engineering Task Force, Tech. Rep. RFC 793, Sep. 1981. [Online]. Available: <https://www.rfc-editor.org/rfc/rfc793>
- [2] W. M. Eddy, “TCP SYN Flooding Attacks and Common Mitigations,” Internet Engineering Task Force, Tech. Rep. RFC 4987, Aug. 2007. [Online]. Available: <https://www.rfc-editor.org/rfc/rfc4987>
- [3] D. J. Bernstein. (1996) SYN Cookies. Accessed 2025. [Online]. Available: <https://cr.yp.to/syncookies.html>
- [4] (2024) Linux kernel source, `syncookies.c`. Accessed 2025. [Online]. Available: <https://elixir.bootlin.com/linux/v6.16/source/net/ipv4/syncookies.c>
- [5] G. Carle, H. Kinkelin, L. Steger, K. Glas, and M. Obermaier, “02_TCP_Attacks.pdf, Network Security (NetSec) Lecture Slides,” 2024, iN2101 – Winter Semester 2024/25, Chair of Network Architectures and Services, Technical University of Munich. Lecture material.
- [6] J. Aumasson and D. J. Bernstein, “SipHash: A Fast Short-Input PRF,” in *Progress in Cryptology – INDOCRYPT 2012*, ser. Lecture Notes in Computer Science, vol. 7668. Springer, 2012, pp. 489–508. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-34931-7_28
- [7] (2024) Linux kernel source, `inet_connection_sock.h`. Accessed 2025. [Online]. Available: https://elixir.bootlin.com/linux/v6.16/source/include/net/inet_connection_sock.h#L286
- [8] (2024) Linux kernel source, `tcp_input.c`. Accessed 2025. [Online]. Available: https://elixir.bootlin.com/linux/v6.16/source/net/ipv4/tcp_input.c
- [9] Q. Cai *et al.*, “Understanding Host Network Stack Overheads,” in *USENIX ATC*, 2021. [Online]. Available: https://www.cs.cornell.edu/~qizhec/paper/tcp_2021.pdf