DNS Cache Poisoning Like it's 2006 (Extended Version)

(Original paper published in Usenix Security 2026)

Omer Ben-Simhon
Hebrew University of Jerusalem

Amit Klein
Hebrew University of Jerusalem

Abstract

The Domain Name System (DNS) underpins virtually all Internet services, making the integrity of DNS resolution critical to security and availability. We present a comprehensive study of a novel class of DNS cache poisoning attacks against BIND 9, the most widely deployed open-source DNS resolver. Our attack focuses on two key capabilities that set it apart from most prior work: (1) reliably predicting *both* critical challenge parameters – the UDP source port and TXID – whereas most existing attacks target only one; and (2) performing this prediction entirely from the *client side*, without attacker-operated authoritative servers for attacker domains, which to our knowledge is a first. We achieve this by exploiting weaknesses in BIND's pseudo-random number generation, enabling highly reliable prediction even under realistic network conditions. In addition to the client-side-only techniques, we also develop server-side techniques which are needed in order to attack the older 9.18 branch of BIND 9. We evaluate our attacks and demonstrate practical success rates across multiple BIND 9 release branches and configurations. All vulnerabilities were responsibly disclosed to the Internet Systems Consortium (ISC) and the FreeBSD Project, leading to two patches and CVEs and acknowledgments.

1 Introduction

The Domain Name System (DNS) [35, 36] is a foundational component of the Internet, originally designed to map human-readable domain names to IP addresses, and later expanded to store and distribute other types of information in the form of resource records. Its performance, resilience, and security directly affect virtually all Internet services. Unfortunately, its central role also makes it a high-value target for adversaries. In particular, DNS cache poisoning attacks-where an attacker injects forged records into a resolver's cache-can redirect traffic, enable credential theft, facilitate malware distribution, and support large-scale censorship.

DNS Ecosystem. The DNS resolution process involves multiple roles: *stub resolvers* running on end hosts, which translate application requests for DNS resolution into queries to a recursive resolver or a forwarder; *forwarders* that relay queries to upstream resolvers; *recursive resolvers* that perform iterative lookups against authoritative name servers, on behalf of clients and forwarders; and *authoritative name servers* (ANS) that serve records for specific zones. Recursive resolvers are of particular interest to attackers, as poisoning their caches can affect large user populations.

BIND 9 and its Market Share. BIND 9 is an open source DNS resolver and authoritative server developed by the Internet Systems Consortium (ISC). While reliable market share figures for resolvers are scarce – due to their inaccessibility from outside their administrative networks – a measurement study provides a useful estimate. In 2011, Gudmundsson [15] reported that BIND 9 accounts for roughly 42% of resolvers observed in DNS trace data, and in 2016, Klein [29] reported that 47% of the SMTP servers of the Alexa Top-1K domain list use BIND 9 resolvers. As such, BIND 9 is the most widely deployed DNS resolver over the Internet. BIND 9 is maintained in multiple release branches: at the time of writing, these are the "older stable" 9.18 series, the "current stable" 9.20 series, and the "development" 9.21 series.

DNS Cache Poisoning. In a DNS cache poisoning attack, an off-path adversary races the genuine ANS's response with a spoofed response carrying forged records. These records, which may include various DNS record types grouped into RRsets, are cached by resolvers for the duration of their time-to-live (TTL) values, allowing a successful forgery to persist for extended duration. Successful poisoning enables traffic redirection, session hijacking, and disruption of critical services. Empirical studies indicate that such attacks remain practical and impactful: Dai et al. [8, Table 1] catalog recent real-world cache poisoning incidents across major resolver platforms, and Klein [30, App. E] demonstrates that

successful poisoning has large-scale consequences for users and infrastructure.

DNS Security Evolution. Before 2008, most resolvers relied almost exclusively on 16-bit TXIDs for query security, as it was assumed that an attacker would have low probability to poison a record in a single attack "round", forcing the attacker to wait a long time between rounds and thus making a cache poisoning attack much less feasible. In 2008, Kaminsky showed that this assumption is wrong [23], by demonstrating attacks that do not attempt to poison the desired record *directly*, but rather—using a new kind of poisoning payload which included auxiliary records (name server record and/or glue record)—*indirectly* poisoned the desired record. In response, defenses such as randomization of UDP source ports have significantly increased the difficulty of such attacks, but have not eliminated them.

RRset Order Randomization. Many DNS resource-record types contain multiple semantically equivalent records, and resolvers frequently return them in randomized order as a simple and widely deployed form of load balancing. For example, randomizing the order of A or AAAA records distributes client connections across multiple servers; randomizing MX or SRV records shares traffic across mail exchangers and service endpoints.

This Work. We demonstrate potent DNS cache poisoning attacks against BIND 9, using only a small number of forged packets – even under realistic Internet conditions and moderate-to-high query loads. Our techniques exploit weaknesses in BIND 9's pseudo-random number generator (PRNG), in the record set shuffling, and in the resolver's behavior in multiple use cases and contexts.

Our main contributions.

- **Efficient, high-impact attacks:** We demonstrate cache poisoning attacks against BIND 9, a resolver with over 40% market share, that succeed with only a handful of spoofed packets-even under realistic Internet conditions, and in the presence of anomaly detection systems.
- **New attack vectors and models:** We introduce both server-side and novel client-side poisoning techniques. The latter includes an *"RRset"* variant that "lives off the land" using non-attacker-owned ANSes, and an *"RRset-ANY"* variant leveraging attacker-controlled zones hosted on third-party ANSes – without the need for the attacker to own any domains. These innovative methods exploit PRNG state inference from RRset ordering observed in resolver answers, in contrast to observing/inferring UDP source ports and/or TXIDs in resolver queries, which is the basis of prior post-Kaminsky techniques described e.g. by Man et al. [32, 33].
- **Broad applicability and robustness:** Our techniques remain effective in the presence of stateful firewalls, under high query loads, and when targeting DNS forwarders, avoiding the need for direct UDP port inference entirely. Our attack can predict the exact UDP source port and TXID (up to very few candidates) without side channels, does not require numerous packets, does not rely on "unexpected" packets reaching the resolver, and is in full effect at the time of writing. We further design atomic and near-atomic query patterns-exploiting ANY queries, RRset order observation, and QNAME minimization-to robustify PRNG state extraction with minimal number of queries, and account for resolver behaviors such as prefetching and DNS cookies.

Overall, our results show that few-packet, state-recovery-based cache poisoning against widely deployed resolvers is still feasible, posing serious risks to Internet users and infrastructure. The last time a few-packet attack was applicable against popular resolver was in 2006 (2 decades ago), before Klein's disclosure of such vulnerabilities in BIND 9 [25], BIND 8 [24], Microsoft DNS Server [26], PowerDNS Recursor [28] and OpenBSD resolver [27], which were subsequently fixed in 2007-2008.

A limitation of our research is that we only experimented with our own (default configuration) BIND 9 servers, and with a handful of BIND 9 open resolvers. While we demonstrated excellent results with these platforms, theoretically, production-grade resolvers may run non-default BIND 9 configuration that may somehow reduce the attack effectiveness. We elaborate on this gap in § 9.

2 Threat Model

We consider an off-path attacker who cannot intercept resolver-authoritative traffic but can send spoofed DNS responses to a target resolver or forwarder. The attacker can also issue queries directly to the resolver. This setting is realistic: measurements show that 14.9% of IPv4 prefixes and 30.5% of ASes still allow spoofed-source traffic [31].

Unlike earlier works that predict a TXID via observing prior TXID values sent to an attacker ANS, ours includes *client-side* attack variants that do not require attacker-operated ANSes for attacker domains. Our attacks use either attacker zones on ordinary ANSes or suitably large third-party RRsets ("Living off the Land" – LotL). The attacker's goal is to recover the resolver's PRNG state from observed queries or responses and predict both the UDP source port and TXID. These predictions are computationally trivial (sub-millisecond on commodity hardware).

We assume the target resolver runs BIND 9 with standard defaults, including qname minimization and prefetching, and

we also consider the impact of DNS cookies. None of these protections prevent our attack. As in most of today's domains (consistent across variable popularity measures), we assume the victim domain may not be DNSSEC-protected [19].

Our model also includes resolvers under load up to ∼10,000 queries per second (QPS) and DNS forwarders. Forwarders never query attacker servers directly, yet our LotL and zone-based methods still enable poisoning. This broadens the threat surface considerably.

Table 1 summarizes attacker requirements across BIND versions and roles.

	9.18 resolver	**9.20/9.21 resolver**	**9.20/9.21 forwarder**
Client side w/zone	×	✓	✓
Client side (LotL)	×	✓	✓
Server side (ANS)	✓	✓	×

Table 1: Attack compatibility with BIND versions.

In summary, our model reflects realistic conditions in which off-path attackers can poison BIND 9 resolvers and forwarders quickly, reliably, and with only a handful of spoofed packets.

3 Breaking the BIND PRNG

In this section we explain how to break the BIND PRNG in settings relevant to our attack. This is a fundamental step in our attack: once the internal PRNG state is known, the attacker can predict future PRNG values, from which the TXID and source ports of BIND's outbound queries are derived. The complete DNS cache poisoning attack is described in § 4.

We present two PRNG breaking variants – using the RRset order, and using TXIDs.

3.1 Xoshiro128** PRNG in BIND

BIND employs the Xoshiro128** v1.0 PRNG [5] (sometimes written as "Xoshiro128 star star") to randomize critical parts in DNS messages: the TXID and UDP source port fields in outbound queries, and the order of Resource Record set (RRsets) serialization in responses. Xoshiro128** maintains its internal state as a 128-bit vector, typically implemented as four 32-bit integers `seed[0]`, `seed[1]`, `seed[2]`, and `seed[3]`. Each invocation of the PRNG updates this state vector using linear (bitwise) operations, which can be expressed as multiplication by a known, fixed 128×128 binary matrix over GF(2). The 32-bit PRNG output is derived from the updated state (specifically, `seed[0]`) using a non-linear *star-star transformation* involving two 32-bit multiplications (hence the name) and a left-rotation:

$$\text{starstar}(x) = (9 \cdot \text{rotl}(5 \cdot x, 7))$$

Specifically, TXIDs and UDP source ports are generated using outputs from the PRNG; TXIDs are the least significant 16 bits of the PRNG output value, while for UDP source ports, the PRNG output is cast into the configured port range.

3.2 PRNG Breaking Procedure

The security offered by randomization-based defenses hinges on the unpredictability of the PRNG output. Our attack exploits vulnerabilities that allow an attacker to predict future PRNG outputs accurately. Breaking the PRNG consists of three steps. In the 1st step, the observed DNS queries or responses (TXIDs or RRset permutations) are used by the attacker to obtain partial or full PRNG outputs. In the 2nd step, the attacker obtains some internal state bits from the PRNG outputs, thanks to the mathematical properties of the star-star transformation operations (rotations and multiplications). In the 3rd step, multiple observations of these internal bits are used by the attacker to formulate linear equations over the vector space $\text{GF}(2)^{128}$ which represents the internal state at the first sample (call this the *initial internal state*) – each sample reveals several bits from the internal state corresponding to it, and since this state is a linear combination of the initial state, this provides linear equations on the initial state. The attacker then solves these equations using Gaussian elimination to recover the full 128-bit PRNG initial internal state. The overall flow of this state-recovery process is illustrated in Fig. 1.

We have three attack variants – RRset, RRset-ANY and RR-QMA. In the RRset-ANY and RRset attacks, the attacker obtains 32-bit readouts of several consecutive internal PRNG states by inspecting the order of cached RRsets returned from an ANY query or from a series of regular queries. In the RR-QMA attacks, the attacker obtains 9-bit readouts of several internal PRNG states by observing TXIDs of outbound queries that result from a series of attacker queries. These PRNG states are in "skips" of two (the attacker obtains the bits from consecutive outbound queries' TXIDs, but every outbound query starts with consuming a PRNG value for the UDP source port).

3.3 Obtaining 32 PRNG Internal State Bits from RRset Order

BIND randomizes RRset order using a 32-bit PRNG output and the Fisher-Yates shuffle algorithm [40]. This behavior is enabled by default (`rrset-order random`). Our RRset-based and ANY-based attack variants rely on extracting the full 32-bit PRNG output used for RRset shuffling. This is possible because the entire RRset permutation depends deterministically on a single PRNG value.

BIND generates RRset permutations by applying the Fisher-Yates shuffle, using a sequence of swaps determined

by the PRNG output. Specifically, each swap index is calculated as the PRNG value modulo a decreasing counter, from N down to 1, where N is the RRset size. By comparing the received RRset order to its DNSSEC-sorted order (which is the internal order BIND uses for storing RRsets), the attacker recovers the permutation σ applied.

The permutation σ defines a system of modular congruences for the unknown PRNG output modulo integers 1 through N. However, not all moduli in $\{1,\ldots,N\}$ are pairwise coprime. The attacker selects a subset of congruences for which the moduli are maximal powers of primes. Solving this subset using the Chinese Remainder Theorem (CRT) yields the PRNG output modulo $\mathrm{lcm}(1,\ldots,N)$. Since $\mathrm{lcm}(1,\ldots,23) > 2^{32}$, an RRset of size $N \geq 23$ is sufficient to uniquely recover the original 32-bit PRNG output.

The procedure is formalized in Alg. 1.

Algorithm 1: Extracting PRNG Output from RRset Order

Receive DNS response with randomized RRset (where σ is the permutation): $(r_{\sigma(0)}, r_{\sigma(1)}, \ldots, r_{\sigma(N-1)})$;
Sort RRset according to the DNSSEC order to reconstruct BIND's internal order $(r_0, r_1, \ldots, r_{N-1})$ and obtain σ;
Initialize the vector $p = [0, 1, \ldots, N-1]$;
for *j=0,...,N-1* **do**
    Find k such that $p[k] = \sigma(j)$;
    Set $m_{N-j} \leftarrow (k-j)$;
    Swap elements $p[j]$ and $p[k]$;
Solve the modular equations $x \mod i = m_i$ for all $i \in \{q^k \mid q \text{ prime},\ k \in \mathbb{N}^+ \text{ maximal s.t. } q^k \leq N\}$ (using CRT with precomputed constants);
return x;

This approach reliably extracts a full 32-bit PRNG output from a single observed RRset permutation. Obtaining 32 bits of the internal PRNG state is achieved by inverting the star-star transformation (which is a composition of trivially reversible operations) on the PRNG output extracted per above.

SIDE NOTE: the fact that the same 32-bit integer x is used as a source of randomness for all Fisher-Yates algorithm iterations is a security vulnerability in and out of itself. Since the permutation is determined by the sequence $(x \bmod N, x \bmod (N-1), \ldots, x \bmod 2, x \bmod 1)$, and as explained above, there are only $\mathrm{lcm}(1,\ldots,N)$ possible such sequences, thus it follows that for $N \geq 4$, not all $N!$ permutations are possible as an algorithm output, i.e. the algorithm only provides partial shuffle randomness. The problem exacerbates for $N \geq 23$ (where $\mathrm{lcm}(1,\ldots,N) > 2^{32}$) since there are only 2^{32} possible values of x, and so the actual number of possible permutations is only $\min(\mathrm{lcm}(1,\ldots,N), 2^{32})$. Replacing the BIND PRNG has no bearing on this problem.

3.4 Obtaining 9 PRNG Internal State Bits from TXIDs

The 16-bit TXID is simply the least significant 16 bits from the 32-bit PRNG output, thus the attacker trivially obtains a partial PRNG output. We now show how these 16 least significant bits of the PRNG output reveal 9 bits of the internal PRNG state. The least significant 16 bits of the PRNG (i.e., the TXID) are generated as follows (multiplication is done $\bmod 2^{32}$):

$$\begin{aligned}\mathrm{TXID} &= \mathrm{starstar}(\texttt{seed[0]}) \bmod 2^{16} \\ &= \big(9 \cdot \mathrm{rotl}(5 \cdot \texttt{seed[0]}, 7)\big) \bmod 2^{16}\end{aligned}$$

We isolate the intermediate value:

$$\begin{aligned} & x = \mathrm{rotl}(5 \cdot \texttt{seed[0]}, 7) \bmod 2^{16} \\ \Rightarrow \quad & x = 9^{-1} \cdot \mathrm{TXID} \bmod 2^{16}\end{aligned}$$

Next, we extract partial bits of the internal state:

$$\begin{aligned} y &= x \gg 7 = 5 \cdot \texttt{seed[0]} \bmod 2^9 \\ z &= 5^{-1} \cdot y \bmod 2^9 = \texttt{seed[0]} \bmod 2^9\end{aligned}$$

The modular inverses $9^{-1} \mod 2^{16}$ and $5^{-1} \mod 2^9$ can be precomputed. This process reveals the 9 least significant bits of `seed[0]` from each observed TXID. In general, given $k \geq 8$ least significant bits of $\mathrm{starstar}(w)$, this approach can be used to expose the $k-7$ least significant bits of w.

3.5 Recovering the PRNG State using Linear Algebra

Below we explain how to recover the PRNG state from a series of 20 observations of internal 9 bits obtained from TXIDs (180 bits in total). A similar approach is used for RRsets, using 5 samples of RRset orders (160 bits in total).

The PRNG state evolves linearly, so each successive state can be expressed as:

$$x_n = A^n x_0$$

where A is a known 128×128 binary matrix over $\mathrm{GF}(2)$. Since TXID and port generation alternate, each observed TXID corresponds to every second PRNG state.

Observing 20 TXIDs exposes 9 coordinates (bits) from each of the following vectors:

$$A^0 x_0,\ A^2 x_0,\ A^4 x_0,\ \ldots,\ A^{38} x_0$$

This yields 180 equations over 128 unknowns – enough redundancy to ensure a correct and unique solution using Gaussian elimination over $\mathrm{GF}(2)$. While 20 TXIDs provide strong redundancy, successful recovery is possible with fewer equations.

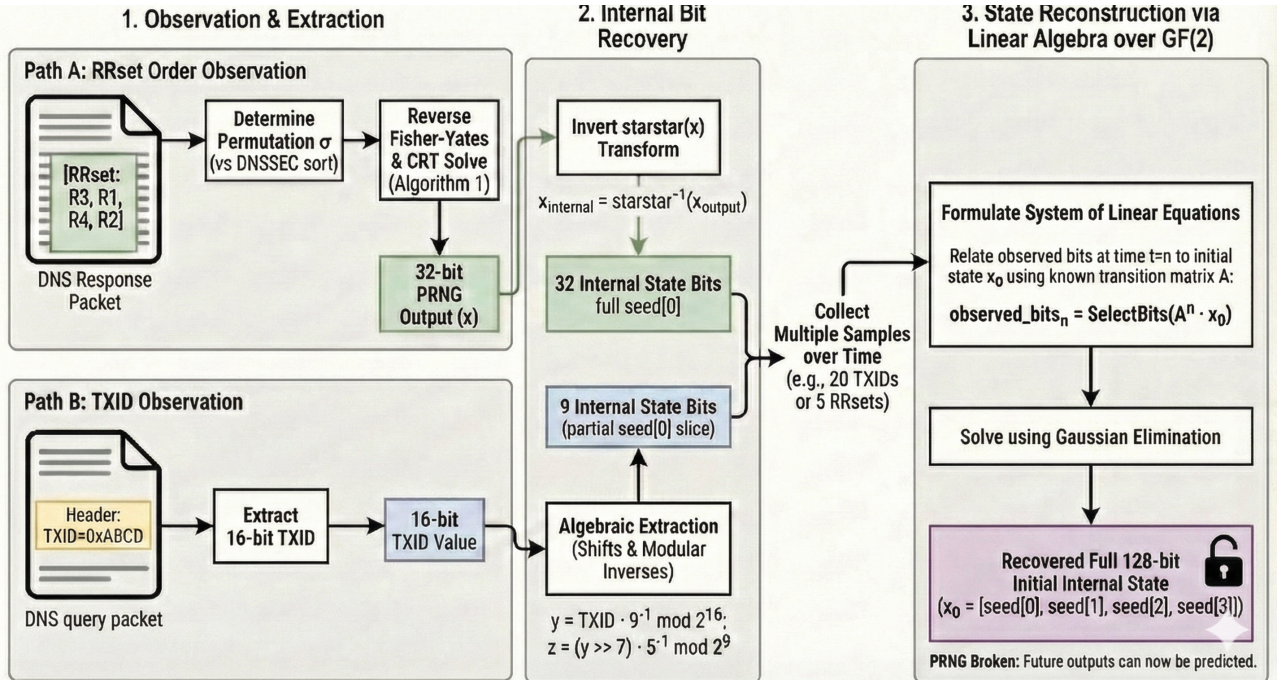

Figure 1: The PRNG Breaking Stages

4 DNS Cache Poisoning

In BIND 9.20 and 9.21, our cache poisoning attack follows a two-phase structure. First, the attacker reconstructs the PRNG state (§ 3.5) by observing randomized outputs such as RRset permutations or TXIDs, as explained in § 3.3 and § 3.4. As part of this phase, the attacker needs to also infer the resolver's ephemeral UDP port range (unless it is the default range), which is necessary to predict destination ports used in outbound queries (described in App. C and App. D). Once both the PRNG state and port range are known, the attacker resynchronizes with the resolver's PRNG via a dedicated query, then sends a "triggering" query to the resolver and a corresponding spoofed DNS response containing a fraudulent NS delegation before the legitimate response arrives, as described in § 4.5. This process allows for precise and reliable cache poisoning with minimal spoofed traffic. The situation is more complex for BIND 9.18, due to its threading model that separates inbound and outbound query handling; we address this case in detail in App. B. Fig. 2 is a schematic view of the entities involved in our attacks.

4.1 BIND 9.20 and 9.21's Threading Model

Upon startup, BIND determines the number of logical CPU cores ("threading cores") and instantiates that many copies of its *worker threads*. Each thread maintains a dedicated instance of the PRNG.

Incoming datagrams (queries) are dispatched by the operating system to the worker threads. Specifically, BIND uses the `SO_REUSEPORT` option (on Linux) or `SO_REUSEPORT_LB` (on FreeBSD) to deterministically assign each packet to a thread. The operating system chooses the thread number that accepts the packet by casting a keyed hash of the transport 4-tuple[1] into the range $[0, n-1]$, where n is the number of BIND worker threads. This design ensures that multiple queries from the same IP-port pair are handled by the same thread, enabling the attacker to target a specific PRNG instance by simply fixing the source IP and port of the attacker queries. While queries are handled by multiple threads, BIND 9 maintains a *single*, central cache from/to which all threads read/write records.

4.2 Attack Variants

We present several attack variants tailored specifically to different BIND versions and configurations. All attacks target BIND's predictable PRNG, enabling accurate cache poisoning through PRNG state recovery.

4.2.1 RRset-based Attack (BIND 9.20/9.21)

This is a client-side attack that leverages the default RRset order randomization (`rrset-order random`) in BIND 9.20 and 9.21 to reconstruct the resolver's PRNG state. The attacker prepares a domain with an RRset of at least 23 records

[1] In FreeBSD, the destination address (i.e. the server's own IP address) is not included in the hash input. This has no impact on our attack.

and uploads it to a standard ANS.

The attacker ensures the resolver caches this RRset, then sends multiple consecutive queries from a single fixed IP-port pair, ensuring all queries are handled by the same resolver thread and its associated PRNG instance. By analyzing the RRset order observed in the resolver's responses, the attacker extracts several consecutive 32-bit PRNG outputs (§ 3.3). From these outputs, the attacker reconstructs the 128-bit PRNG internal state using linear algebra over $\mathrm{GF}(2)^{128}$, as detailed in § 3.5.

This attack can also utilize third-party domains ("Living off the Land") with inherently large RRsets (e.g., domains with numerous A or NS records), thus relieving the attacker from the need to upload *any* data to an ANS. Additionally, the attacker can infer a resolver non-standard UDP source port range using DNS-based UDP port echo services such as DRINK DNS servers [6] (set up by the attacker, or a third-party one), as will be explained later.

Once the PRNG state and UDP port range are known, the attacker proceeds with cache poisoning as described in § 4.5. The attacker sends a query to synchronize with the PRNG state, followed immediately by a query for the actual target domain. Both queries are issued from the same IP-port pair to ensure consistent thread mapping and PRNG instance reuse.

ANY-based Attack Variant (BIND 9.20 and 9.21): This variant significantly improves the attack's resilience under load by exploiting the atomicity of the DNS ANY query. A single ANY query can retrieve multiple (cached) RRsets simultaneously in a single DNS response, allowing the attacker to extract multiple *consecutive* PRNG outputs. This batching behavior is described in detail in § 4.3, and ensures that the output is generated without interference from concurrent PRNG invocations. In practice, the attacker ensures multiple RRsets for various DNS record types of the *same* name are pre-cached by the resolver. The attacker then issues a single ANY query for that name, resulting in a response that includes all cached RRsets. Since BIND constructs the response as a single operation – without triggering additional network activity between records — the attacker obtains multiple PRNG outputs consecutively and without external interference.

To illustrate, an attacker can configure a nameserver hosting multiple RRsets of unassigned type numbers or standard types (such as TXT, SVCB, HTTPS, MX, SRV or A records). The BIND resolver caches these RRsets upon initial queries. A subsequent ANY query triggers the resolver to respond atomically with all cached RRsets. By analyzing the randomized RRset orders in the response, the attacker extracts several consecutive PRNG outputs from this single response. If necessary, due to DNS payload size limitations in UDP, the attacker may adjust the number of records per RRset or use specific standard record types with smaller RDATA to ensure the response fits within a single UDP datagram.

This ANY-based approach notably reduces the number of required queries, significantly improving resilience to resolver load and minimizing detectability.

4.2.2 RR-QMA Attack

The RR-QMA (Query-name Minimization [7] Assisted) attack is a server-side technique that allows the attacker to recover the PRNG state and poison the cache without relying on RRset randomization. This makes the attack more robust to non-default resolver configurations, such as when `rrset-order` is set to `cyclic`, or to `none` (the latter means the records are returned in a fixed, internal order). The attack relies on observing a series of consecutive outbound queries' TXID values and breaking the PRNG using these TXID values.

The query-name minimization (`qmin`) aspect of the attack enhances its reliability under load by triggering near-atomic batching of queries queued "behind" a pending qmin lookup, as discussed in § 4.3. Even without qmin, the attack remains feasible by sending queries in rapid succession and relying on the absence of interfering PRNG activity during that window.

Below we describe the attack for BIND 9.20 and 9.21. For brevity, the adaptation for BIND 9.18 is described in App. B.

To carry out the RR-QMA attack, the attacker first sends multiple queries from a single client socket (a fixed IP-port combination), thereby guaranteeing that these queries are handled by the same worker thread. The attacker controls an ANS for the queried domains and can observe incoming resolver queries to the attacker ANS. We denote by `x` a fully qualified domain name (FQDN) in the attacker's domain, and by `www.target.example` the target FQDN the attacker attempts to poison. The attacker sends a batch of 20 queries for `x`, each with a different unassigned query type (e.g., `40000`, `40001`, ..., `40019`). Throughout this paper, we use the notation `nnnnn?x` to denote a DNS query of unassigned type `nnnnn` (i.e., `TYPEnnnnn`) for the name `x`.

When qmin is enabled (the default), BIND issues a minimizing NS query before resolving the original query. The attacker must account for version-specific behavior:

- **BIND 9.21:** The resolver sends an `NS?x` query for the FQDN itself. Thus, query batching is triggered even when all 20 queries target the same name with different types. The attacker simply uses `x.attacker.example` throughout the batch.
- **BIND 9.20:** The resolver minimizes only up to the parent domain. To induce batching here, the attacker adds a variable subdomain level, such as `x.sub-Z.attacker.example`, where `sub-Z` changes across rounds. This causes the resolver to first issue `NS?sub-Z.attacker.example`, withholding the remaining queries until that response arrives.

The attack remains effective regardless of qmin status, although the queuing behavior improves atomicity when qmin

is enabled. In BIND 9.21, the queued queries are dequeued in reverse order once the qmin lookup completes, but this has no impact on the attack, since the attacker can reverse the observed TXIDs accordingly.

Each of the attacker's queries results in a resolver query to the ANS. By observing the TXIDs of these queries, the attacker reconstructs the resolver thread's 128-bit PRNG state using linear algebra over $GF(2)^{128}$.

Following the PRNG reconstruction, the attacker proceeds with cache poisoning as explained in § 4.5.

4.3 Atomicity and the Role of Query Name Minimization

Atomicity, in our context, refers to the ability of the attacker to observe a sequence of PRNG outputs from the resolver that are produced without unrelated or intervening PRNG-consuming operations. This property is essential for accurate PRNG state reconstruction.

In the ANY-based attack, atomicity is trivially achieved: BIND constructs a single response that includes multiple RRsets, each shuffled using a distinct PRNG output. Since the response is composed in one uninterrupted pass, the attacker obtains a sequence of consecutive PRNG outputs without interference.

In the RR-QMA attack variant, we achieve *near*-atomicity through the resolver's implementation of *Query Name Minimization* (qmin). When BIND receives a query for a previously unseen subdomain (e.g., `A?foo.www.target.example`), it first sends a minimizing query (e.g., `NS?www.target.example`) and temporarily queues the original query. Additional dependent queries (e.g., `40000?foo.www.target.example`, `40001?foo.www.target.example`, etc.) are similarly held in the queue. Once the minimizing query completes, BIND flushes the queue and issues the dependent queries rapidly, allowing the attacker to observe multiple PRNG outputs that are likely to be consecutive.

4.4 UDP Port Range Inference

The outbound UDP source port range used by BIND is identical to the operating system ephemeral port range. By default, Linux uses an ephemeral UDP source port range of 32768-60999, and FreeBSD uses 49152-65535. Thus, the port range inference phase is **optional**: if the resolver's operating system is known to use the default port range or if the attacker is able to infer the port range by other means (e.g., prior observation), then port range inference is unnecessary. We describe a client-side source port range inference (using e.g. a 3rd-party DRINK server) in App. C and a server-side source port range inference in App. D.

4.5 The Actual Poisoning Step

Our attacks employ a payload structure similar to the well-known Kaminsky DNS cache poisoning technique [23]. The essence of the Kaminsky payload is injecting fraudulent authoritative NS (name server) records into the resolver's cache instead of trying to directly poison the target name. By injecting these NS records, the attacker redirects all future queries for the targeted domain and its subdomains to attacker-controlled name servers.

Our attack scenario proceeds as follows:

1. **Pre-attack step:** Shortly before the actual attack, the attacker issues a query (e.g., `A?foo.www.target.example`) causing the resolver – due to qmin – to first query `NS?www.target.example` from the victim's ANS. Since `www.target.example` typically lacks explicit NS records (being merely a hostname), the ANS responds with a negative response (an SOA record), cached temporarily (e.g., 60 seconds). This ensures subsequent attacker queries do not trigger additional qmin queries at critical poisoning moments.

2. **Cache Poisoning Attack:** At the intended poisoning moment, the attacker sends two queries in rapid succession-either in the same TCP segment or as back-to-back UDP datagrams. The first query enables the attacker to synchronize with the resolver's PRNG state. For a RR-QMA attack, the query is for an attacker-controlled domain, which allows the attacker (server) to re-sync the PRNG state using the TXID value of the query. For the RRset-based attacks, the query is for a cached RRset, allowing the attacker (client) to re-sync the PRNG state using the RRset shuffle order.

   The second query targets the victim domain with an unassigned query type (e.g., `44444?www.target.example`), causing the resolver to forward it to the legitimate ANS of the victim domain.

   Immediately thereafter, the attacker sends the resolver a DNS response with a spoofed source IP address of the legitimate ANS, with source port 53. The destination is the resolver's IP address, and the destination port is the UDP source port predicted for the resolver's outbound query based on the known PRNG state. The DNS header contains the TXID predicted for that query. The spoofed response carries a Kaminsky-style payload. The authority section contains fraudulent delegation NS records (e.g., `www.target.example NS ns-attacker123.attacker.example`) pointing to an attacker-controlled nameserver. The additional section supplies a corresponding glue A record (e.g., `ns-attacker123.attacker.example A 6.6.6.6`) for immediate resolution of the malicious nameserver.

   Because the previously cached negative SOA response for `NS?www.target.example` is not a "real" positive

cached record, the resolver accepts and caches the attacker's forged delegation. As a result, subsequent queries-including prefetches-are directed to the attacker-controlled server.

4.6 Overcoming DNS Cookies

DNS Cookies (RFC 7873 [1]) provide a mechanism intended to mitigate off-path DNS cache poisoning by embedding a cryptographic identifier (the "DNS cookie") into DNS messages. A resolver includes a unique client cookie in its queries to an ANS, expecting this cookie to be echoed back in the response, alongside an additional server-generated cookie. Once a resolver identifies a server as cookie-compliant, it subsequently rejects any responses from that server lacking valid cookie values.

However, DNS Cookies are not widely supported in practice. Measurements show that only about 20% of domains from the Tranco Top-10K list support DNS Cookies [42], and a separate study found cookie support in 32% of authoritative servers in the Alexa Top 1 Million list [9].

Furthermore, to the best of our knowledge, BIND is the only major open-source resolver that enables DNS Cookies by default. These observations suggest that DNS Cookies have limited real-world adoption and do not provide a robust or widely enforced defense mechanism.

In BIND's implementation, DNS cookie compliance information is cached in the Address Database (ADB), with a relatively short expiration time: 60 seconds in BIND 9.20/9.21, and 1800 seconds in BIND 9.18. These are hard-coded, non-configurable constants.

Our attacks exploit both the short expiration timeout and the brief window following expiration-before the resolver receives a new cookie-compliant response and re-establishes compliance. When the ADB entry for a nameserver expires, the resolver temporarily "forgets" the server's cookie support and accepts responses that lack valid cookies. This creates a short, recurring opportunity in which spoofed responses may be accepted.

To take advantage of this, the attacker continuously attempts to poison the resolver's cache. Once the ADB entry expires and before a new compliant response is received, a spoofed response without a correct client DNS cookie will nevertheless be accepted.

In practice, DNS cookies do raise the bar for attackers by introducing a validity check on responses. However, due to the resolver's reliance on short-lived compliance caching, they do not constitute a strong defense in BIND. Consequently, our attack remains highly effective when DNS cookies are employed by ANSes. Moreover, our attack is likely not to be hindered by DNS cookies as their deployment in ANSes is limited.

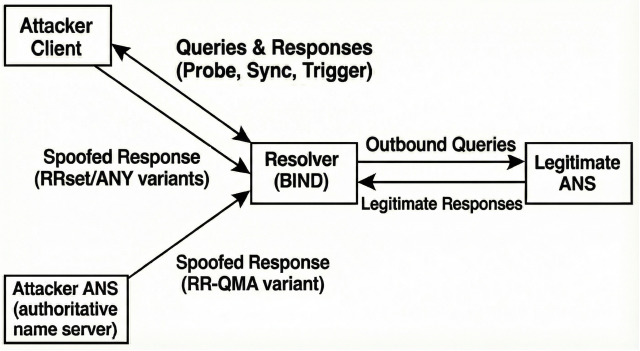

Figure 2: DNS Cache Poisoning Attack Entities

4.7 DNS Prefetching and Its Implications for Cache Poisoning

DNS prefetching is a mechanism used by resolvers (such as BIND) to proactively refresh cached DNS records before their TTL expires, thereby reducing query latency for frequently queried domains. Specifically, when a cached record's remaining TTL is below a configured threshold (e.g., 2 seconds in BIND, via the `prefetch` directive), and the resolver receives a query for that record, the resolver serves the cached response and simultaneously issues a new query to the ANS to refresh the record. While prefetching can complicate other types of DNS cache poisoning attacks – for example, those that rely on precisely predicting when the next query to the target domain will occur – our attack is unaffected by it. This is because it poisons the NS record of the target domain. As a result, any subsequent prefetching attempts to refresh resource records (e.g., `A?www.target.example`) will be directed to the attacker's malicious ANS, not the legitimate one.

4.8 Round Trip Time (RTT)

An important factor in both client- and server-side attacks is the relative round-trip time (RTT) between the attacker's injection point and the target resolver, compared to the RTT between the resolver and the legitimate ANS. The attack's success is conditional on whether the spoofed answer reaches the resolver before the genuine response.

Whether in server-side attacks such as RR-QMA, where the spoofed packet is sent from the attacker's ANS, or in client-side attacks such as the RRset and RRset-ANY attacks, where it is sent from the attacker's client machine, the requirement is the same: the RTT from the attacker's injection point to the resolver must be shorter than the RTT between the resolver and the genuine ANS. If the attacker's machine is significantly closer (network-delay-wise) to the resolver than the genuine ANS, the spoofed response will reliably arrive first, widening the attack window and greatly increasing the probability of success.

5 Experiments and Evaluation

In this section we evaluate the effectiveness and robustness of our DNS cache poisoning attacks against BIND. We begin by detailing the experimental setup and baseline configuration (§ 5.1), followed by results from baseline experiments (§ 5.2). We then explore deviations from the baseline in functional variants, and conclude with high-load scenarios (§ 5.9).

5.1 Experimental Setup

We registered two domains – one under the `.com` TLD and another under the `.net` TLD – used respectively as the attacker-controlled domain and the target domain in our experiments. For anonymity, we refer to these as `attacker.example` and `target.example` throughout this section and the rest of the paper.

For ANSes, as well as all other machines involved in the experiments except for the target resolver, we used Azure `Standard B2s` virtual machines (2 vCPUs, 4 GiB RAM) deployed on Microsoft Azure across two regions: `East US` and `North Europe`. All such machines ran Ubuntu 24.04.2 with Linux kernel version 6.11.0.

The target resolver machine, unless stated otherwise, was an Azure `Standard D4as v5` instance (4 vCPUs, 16 GiB RAM) located in the `Central US` region.

The target resolvers were BIND 9.18.38, 9.20.11, and 9.21.10, compiled from source with default configurations and linked against `jemalloc` (as recommended in the BIND build instructions) unless stated otherwise. Specifically, `prefetch` and `qmin` were enabled, per BIND's default configuration. None of our domains had DNSSEC records, so de-facto, we had no DNSSEC protection for the target domain – consistent with the non-DNSSEC deployment rates in the "Majestic Million" list: 95% for the top 100, 82.4% for ranks 101-1,000, and 89.2% for ranks 1,001-10,000 (see the first line in the first table in ICANN's monthly report [19]). Importantly, the domains in the "Majestic Million" list are not arbitrary: they represent high-value, operationally critical assets that attract both benign and adversarial traffic at Internet scale. As such, the fact that DNSSEC adoption remains low precisely among these influential domains underscores the practical relevance and real-world impact of our attack surface. Finally, the target domain ANS was set not to respond to client cookies, which is the common case as explained in § 4.6.

All experiments used the same Python-based attack framework (available as an artifact) to generate queries, spoofed responses, and monitor resolver behavior. The number of iterations for each experiment was 100 in § 5.2 and § 5.9, and 10 elsewhere.

We injected spoofed responses using a dedicated attacker source IP address. Spoofing was performed "at the destination" by rewriting the legitimate, attacker-owned source IP address of inbound packets into the target domain's ANS IP address using `iptables` SNAT rules. This approach was necessary because Azure prohibits sending packets with forged source IP addresses, and it also ensured that no spoofed traffic was sent over the public Internet. All spoofing thus took place within our experimental network.

In our experiments, we rolled forward the PRNG 500 steps (PRNG offsets) from its extracted state, for each state we generated the predicted TXID and UDP source port, and we sent a spoofed response using these values. We recorded whether the attack succeeded (e.g. Fig. 3) and if so, at which offset (e.g. Fig. 4). As can be seen, when the attack is successful, with very high probability we only need a handful of guesses (i.e. offsets 0-3, not the entire offset range 0-499).

5.2 Baseline Experiments

We begin by evaluating the success rates of our three attack variants: RR-QMA, RRset, and RRset-ANY. Table 2 summarizes the configuration and outcome of each baseline experiment.

Variant	BIND Ver.	Protocol	Attack Success
RRset	9.20.11	UDP	100.00%
RRset	9.20.11	TCP	100.00%
RRset	9.21.10	UDP	100.00%
RRset	9.21.10	TCP	100.00%
RRset-ANY	9.20.11	UDP	100.00%
RRset-ANY	9.20.11	TCP	100.00%
RRset-ANY	9.21.10	UDP	100.00%
RRset-ANY	9.21.10	TCP	100.00%
RR-QMA	9.18.38	UDP	97.00%
RR-QMA	9.18.38	TCP	92.00%
RR-QMA	9.20.11	UDP	100.00%
RR-QMA	9.20.11	TCP	99.00%
RR-QMA	9.21.10	UDP	96.00%
RR-QMA	9.21.10	TCP	97.00%

Table 2: Summary of baseline attacks against latest (at the time of writing) BIND versions: 9.18, 9.20 and 9.21.

A week before submission, ISC released new BIND versions: 9.18.39, 9.20.12, and 9.21.11. To verify that our findings remained valid, we conducted preliminary spot-checks by sampling representative attack variants against these releases. Specifically, we tested the RR-QMA variant against BIND 9.18.39, the RRset-ANY variant against BIND 9.20.12, and the RRset variant against BIND 9.21.11. In all cases, the results were consistent with our baseline experiments, confirming that the vulnerabilities persist unchanged in the latest versions.

In the next subsections, we explore the impact of system configuration deviations from the baseline.

5.3 FreeBSD Operating System

We evaluated our attack variants in a setup identical to the baseline (see § 5.1), except that the resolver was deployed on a FreeBSD 14.3 machine. We tested the RR-QMA attack against BIND 9.18, the RRset-ANY attack against BIND 9.20, and the RRset attack against BIND 9.21. Since FreeBSD lacks `iptables` we could not employ our "spoof-at-the-destination" technique to demonstrate the poisoning step. Instead, we compared our predicted UDP source ports and TXIDs against the actual values used by the resolver for the triggering queries. Across 10 iterations of each experiment, our predictions matched the resolver's values in all cases (10/10 success).

An interesting phenomenon arose in the RR-QMA attack against BIND 9.18, where we were able to perform the poisoning step (without the need for spoofing!) and achieve successful cache poisoning. This was due to a FreeBSD-specific vulnerability we identified during our experiments. We describe it in detail in App. A.

5.4 Non-Prefetched Records

For this experiment, we kept the prefetching feature enabled (which is the default in BIND), but we ensured that our queries did not trigger prefetching by avoiding repeated queries for the same name within the TTL period. Under these conditions, attack success remained 10/10, suggesting that the attack works well when prefetching is not triggered.

5.5 DNS Cookies

To evaluate the impact of DNS cookies on our attack, we ran the RRset-ANY attack variant with a victim ANS configured with DNS cookie support (specifically, we used BIND 9 with `answer-cookie yes`). The setup was identical to the baseline experiments, except that the genuine ANS echoed back DNS cookies in its responses.

As expected, spoofed responses issued *before* the 60-second cookie timeout failed to poison the cache. However, once the timeout elapsed, the spoofed responses were accepted by the resolver, enabling successful cache poisoning. Across 10 independent iterations, the attack succeeded in 10/10 cases, demonstrating that DNS cookies do not provide meaningful protection against our technique once the (short) DNS cookie timeout elapses.

5.6 In-the-Wild Open Resolvers (BIND 9.20, RRset-ANY)

We conducted a safe-testing experiment targeting in-the-wild BIND 9.20 resolvers using the RRset-ANY attack variant over TCP. The goal was to evaluate the feasibility of PRNG prediction under realistic Internet conditions without sending any actual spoofed responses to the resolvers.

Discovery and Filtering. We began with a public list of ∼22,000 open resolvers [41]. Each IP address was queried for the special CHAOS class/TXT name `version.bind`. Only resolvers explicitly reporting a BIND `9.20.x` version were considered for further testing. We found 5 BIND 9.20 open resolvers.[2]

Preparation of Test Data. The simulated attack targeted our own domain to ensure all testing remained safe. We configured a dedicated subdomain of our experimental domain (e.g. `attacker.example`) to host five RRsets of size 23, with record types `MX`, `HTTPS`, `TXT`, `SVCB`, and `SRV`, all with TTL of 60 s. These standard record types were intentionally chosen to avoid a potential risk of undefined behavior that might arise from unassigned types (which we freely use in our experiments against our own servers). We used record data that was syntactically valid but non-functional. Prior to testing, each candidate resolver was validated to ensure it returned RRsets in BIND's random-order mode (as opposed to a cyclic/fixed order).

Safe-Testing Procedure. The safe-testing flow followed our standard RRset-ANY procedure (§ 4.2.1) but instead of mounting the full poisoning step, we simulated it. We did not send spoofed responses, but rather we recorded the predicted TXID and ports and compared them to the values recorded in `target.example`'s ANS logs.

Results. Of the 5 BIND 9.20 servers located, one resolver exhibited frequent timeouts, rendering it unreliable and unsuitable as a valid target. We simulated the RRset-ANY attack against the 4 other open BIND 9.20 resolvers located in the wild. One resolver failed in the DRINK-based ephemeral port range inference phase. We observed that in 9 out of 10 iterations the predicted TXID was correct but in all of them, the predicted UDP port was incorrect. Moreover, all observed ports were below 12,000. These two observations strongly suggest that this resolver is a BIND 9.20 behind a NAT.

For the remaining three resolvers, all simulated attacks succeeded within 500 PRNG steps (i.e. would have required up to 500 spoofed packets for successful poisoning). Notably, 83% of the simulated attacks had PRNG offsets within the range 0-3 (i.e. up to 4 spoofed packets), underscoring the high predictability of the PRNG sequence. In all three cases, the inferred ephemeral port range was consistently 32768-60999, matching Linux's default ephemeral port range — further confirming that these resolvers ran unmodified Linux-based BIND deployments.

These findings demonstrate that, even in uncontrolled Internet conditions, the RRset-ANY variant over TCP retains

[2] Note that the make and version distribution of *open* resolvers is very likely to be quite different than "maintained" resolvers; in addition, many resolvers do not advertise their make and version.

a high success rate for PRNG prediction against BIND 9.20 resolvers.

5.7 Living off the Land with Third-Party Authoritative Servers

To demonstrate that the RRset variant of our attack does not require an attacker-controlled ANS, we conducted an experiment using existing, 3rd-party ANSes that already host large RRsets (size ≥ 23 records). This "living off the land" approach shows that the attack can be launched opportunistically against target resolvers without the attacker having to own a domain and operate an ANS for it. In our experiment, the resolver and domain under attack (`target.example`) were our own, with the 3rd-party ANSes only substituting for the attacker's servers and domains. All queries to these third-party ANSes were standard DNS queries issued at a low rate, ensuring negligible impact on their infrastructure.

Setup. The target resolver was our standard baseline BIND 9.20.11 setup described in § 5.1. The experiment was performed using the RRset attack variant over TCP. Instead of querying an attacker-controlled ANS, we selected 10 third-party ANSes from a curated list of several dozen ANSes obtained from the authors of Moav et al. [34] that serve domain names with at least 23 distinct `A` records in a single RRset. For the (optional) port range inference step (App. C), we used an independent 3rd-party DRINK server. The attack target was a subdomain of our own test domain, ensuring that all cache insertions and resolutions remained within our experimental control. We performed 1 iteration of the RRset attack variant for each third party ANS.

Methodology. The attack followed our standard poison-at-the-destination procedure described in § 4.5, with the only difference being that the large RRset came from a third-party ANS rather than an attacker-controlled one, and the DRINK server used was a 3rd-party server.

Results. In our tests, the RRset variant succeeded in predicting the PRNG state in 10/10 of experiments and achieved a successful poisoning rate of 10/10 when using the third-party ANS and DRINK servers.

5.8 RRset-ANY Attack Against BIND Forwarders

We evaluated the RRset-ANY variant (§ 4.2.1) against BIND 9.20 configured as a forwarder to Google's Public DNS servers (`8.8.8.8, 8.8.4.4`). The general setup was identical to § 5.1, except that the resolver operated solely in forwarder mode, via the BIND configuration directives `forward only; forwarders { 8.8.8.8; 8.8.4.4; };`.

Unlike the baseline resolver experiment, here the attack was triggered using a simple `A?target.example` query, and the spoofed payload was a corresponding `A` response. The attack succeeds only if the record is not already cached by the forwarder, so our experiment explicitly assumes cache misses.

The attack achieved a PRNG prediction accuracy of 10/10 and a cache poisoning success rate of 10/10 as well.

5.9 Load Experiments

To evaluate the effectiveness of our attack variants under resolver load, we first ensured that BIND can withstand high pressure by configuring `recursive-clients 30000`, enabling up to 30,000 concurrent resolution contexts. We then deployed a load generator that sent DNS queries to BIND at a controlled rate. Importantly, our load generator issued only cache-miss queries: each query used a unique name, with short TTLs to simulate real world DNS records, as well as not to put excessive pressure on the resolver RAM. This places maximal pressure on both the PRNG and the UDP/TCP port allocation logic, representing a strictly harder scenario than real-world resolver operation, where the overwhelming majority of queries are cache hits.

Operational studies consistently show that only approximately 5-10% of DNS queries at production resolvers result in a cache miss. Jung et al. report that roughly 90% of queries served by recursive resolvers are cache hits [22, Fig. 12]. APNIC's DNSSEC validation performance study reports end-to-end resolver rates of roughly 135,000 QPS (queries per second) for heavily loaded resolvers and 9,000 QPS for lightly loaded ones [18]; assuming a standard 90% hit / 10% miss ratio, this corresponds to approximately 13,500 and 900 cache-miss QPS, respectively. Huston reports that a large operational deployment reaches peaks of 15.9 million QPS across 265 resolvers (about 60,000 QPS per machine), but again the vast majority of these queries are cache hits [17].

In this context, our 10,000 QPS experiment corresponds to approximately a 100,000 QPS real-world recursive workload under the typical cache-hit ratios (i.e., 10% cache misses). Likewise, our 1,000 QPS cache-miss experiments correspond to roughly 10,000 QPS of real-world resolver traffic. Because our experiments consisted exclusively of cache misses, they represent a stronger stress-test of resolver entropy mechanisms than real deployments typically experience. Since real deployments do not experience sustained 100% cache-miss workloads, we regard our test region of 1-10K cache-miss QPS as representative of operational stress conditions.

Experiments at 10,000 QPS were executed on dedicated Azure `Standard D16ls v5` 16-core, 32 GiB resolver machines. All other experiments (1-1,000 QPS) were conducted on Azure `Standard D4as v5` 4-core, 16 GiB resolver machines. Each load point was tested for 100 iterations.

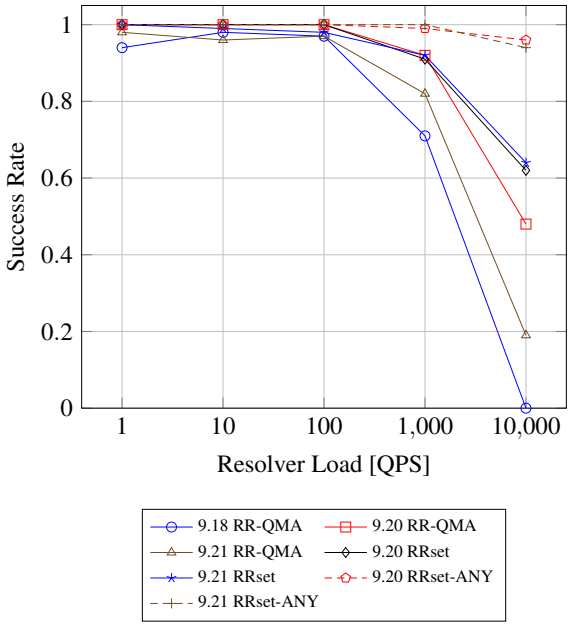

Figure 3: Attack success rate vs. resolver load.

Success Rate vs Load. Fig. 3 shows the cache poisoning success rate as a function of query load, ranging from 1 to 10,000 QPS. While most variants remained effective at moderate loads, several configurations exhibited performance degradation under high load. In particular, the RR-QMA variant against BIND 9.18.37 completely failed at 10,000 QPS. RR-QMA also showed reduced reliability on BIND 9.21.9 and 9.20.10 under load. In contrast, RRset-ANY maintained a success rate above 94% even at the highest tested load.

Offset CDF. In our load experiments, we also measured the PRNG offset (i.e., PRNG steps) between the PRNG state right after it is observed by the attacker (obtained from the re-sync query, immediately before the poisoning query), and the PRNG state used to generate the UDP source port and TXID for the poisoning query. For each offset, the attacker needs to send a spoofed answer with source port and TXID corresponding to the PRNG state at the prescribed offset. Since the attacker's goal is to cover as much probability space (in terms of offsets) as possible in the attack window, it follows that the ideal situation for the attacker is to have very few (ideally one) offsets that cover almost 100% of the probability space. Fig. 4 shows the cumulative distribution of this offset for various attack variants and query loads.

6 Discussion

Our experiments highlight several noteworthy aspects of the attack's behavior and operational implications.

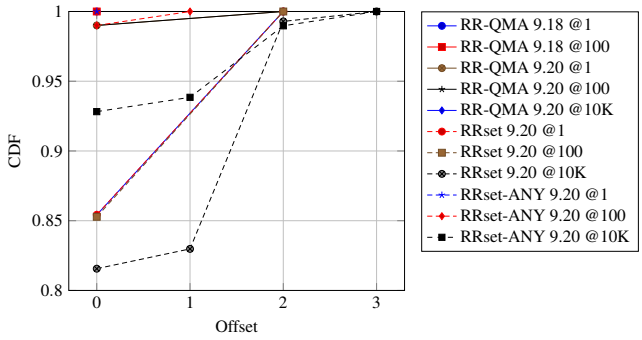

Figure 4: CDF of PRNG offset values per attack variant and load. Most attacks succeed with low offsets, even under heavy load. RR-QMA 9.18 @10K is not plotted since the attack fails 100% of the time. Note that RR-QMA 9.20 @1 graphically coincides with RR-QMA 9.20 @100, and RR-QMA 9.20 @10K graphically coincides RRset 9.20 @1 and RRset 9.20 @100.

PRNG offset distribution under load. Fig. 4 shows that for low to moderate loads (1-100 QPS), most successful cache poisoning attempts occur with a PRNG offset of 0 or 1. As the load increases, the distribution flattens somewhat; yet even at 10,000 QPS, the vast majority of successful attacks complete within the first few offsets. In practical terms, this means that the cache can often be poisoned with as few as 3-4 spoofed packets (or 6-8 if targeting two ANS IPs), even under heavy load. This property reflects the accuracy of our PRNG state recovery procedure and demonstrates that our prediction strategy remains highly effective even under heavy concurrency, contributing directly to the high success rates of the RRset and RRset-ANY variants.

Load-dependent degradation. The only baseline experiment that failed entirely under maximum tested load was RR-QMA against BIND 9.18.37 at 10,000 QPS. We attribute this to the complexity of finding a representative domain for the same PRNG/thread of the target domain (see App. B) under high load. In contrast, in BIND 9.20 and 9.21 the threading assignment model ensures that queries sharing the same 4-tuple are always handled by the same PRNG instance (§ 4.1), eliminating the need to search for a representative domain; as a result, RR-QMA succeeds at moderate loads and RRset-ANY retains $>94\%$ success at 10,000 QPS. App. E provides an upper-bound attack success calculation for a system under load.

Variant-specific observations. The RRset-ANY variant consistently delivers the best resilience under load, owing to its ability to extract multiple consecutive PRNG outputs atomically. Our "Living off the Land" experiment (§ 5.7) demon-

strates that this attack can be mounted opportunistically using large-in-record-count RRsets served by legitimate third-party domains, eliminating the need for attacker-operated ANS for attacker domains. By leveraging existing infrastructure, an attacker both reduces their operational footprint and makes detection and attribution more difficult. Forwarder-mode experiments (§ 5.8) further confirm that the attack is also highly effective against BIND forwarders.

The RR-QMA variant remains applicable in scenarios where the RRset order is fixed rather than randomized, and is also effective against BIND 9.18.

Resilience to defense mechanisms. An important practical advantage of our technique is that, unlike prior post-Kaminsky port-inference attacks, it does not rely on generating or observing "unexpected" traffic such as ICMP errors or UDP packets to closed ports. Such traffic is often silently dropped by stateful L4 firewalls, preventing the side channels based on them from working. Our approach relies only on standard resolver query/response flows, so stateful firewall rules have no impact on the attack logic. Similarly, NAT devices that rewrite source ports do not prevent our technique from recovering the TXID (and in some cases the UDP port) – NAT simply increases the number of spoofed packets needed, roughly proportional to the NAT port range, but the attack remains viable. Finally, anomaly detection systems of the kind described by Afek et al. [2] – which monitor query patterns and traffic bursts – are less effective here because our queries are few in number, closely mimic legitimate traffic, and can be scheduled with realistic inter-packet timing, avoiding the large bursts characteristic of brute-force TXID guessing.

7 Related Work

7.1 DNS Cache Poisoning Against Resolvers and Forwarders

With the bailiwick rule introduction (ca. 1997), off-path DNS cache poisoning attacks against resolvers became reliant on successfully spoofing a legitimate ANS response. This in turn necessitated matching the expected resolver query's UDP source port and TXID, or exploiting fragmentation in the ANS response and spoofing its 2nd fragment.

7.1.1 TXID and UDP Port Attacks

The last time a prominent DNS resolver (BIND) was shown to be vulnerable to DNS cache poisoning attack requiring very few packets was in 2007, in the attacks against BIND 9 [25] and BIND 8 [24]. In that era, BIND used a fixed UDP source port, and only varied the TXID, which those attacks accurately predicted. Those vulnerabilities were promptly fixed by ISC.

Then in 2008, Kaminsky described a powerful DNS cache poisoning attack concept [23] which forced vendors to add UDP source port randomization to their DNS resolvers. Since then, a DNS cache poisoning attack (except fragmentation attacks) has to predict/enumerate over the 2^{30}-2^{32} space of TXID and port combinations.

Post-Kaminsky attacks typically try to predict/infer the UDP source port used by the resolver/forwarder. Specifically w.r.t. Linux, Man et al. exploit a side channel in the Linux ICMP rate limit mechanism [32] and a side channel in the Linux "next hop exception table" [33]. These attacks suffer from numerous drawbacks: (i) they rely on brute-forcing the TXID field, thus for every UDP port candidate, 65536 packets are needed to combine it with all possible TXID values. This makes these attacks less stealthy, i.e., they may trigger burst detection or spoofing detection; (ii) an L4 stateful firewall may silently drop the "unexpected" traffic used by these attacks to determine the UDP source port, e.g., ICMP messages and UDP traffic to closed ports. If so, this traffic does not arrive to the resolver/forwarder and the would-be effect of it is annihilated, resulting in a failure of the underlying attack technique. Moreover, such anomalous traffic patterns may be detected by anomaly-detection systems, e.g. Afek et al. [2]; (iii) they are completely ineffective if port rewriting is used, e.g., when the resolver/forwarder is behind a NAT; and (iv) nowadays, they are no longer in effect since they were fixed by the respective vendors.

In contrast, our technique can predict (up to very few candidates) the UDP port and TXID used by the resolver/forwarder without a need for brute-forcing; our attack does not make use of "unexpected" traffic as part of the underlying technique; our attack can still predict the TXID even when a NAT is present, and can thus still be used, albeit requiring more spoofed packets (typically a small factor times the NAT port range); and our attack is in full effect at the time of writing.

7.1.2 Fragmentation Attacks

Fragmentation attacks such as Herzberg et al. [16] and Zheng et al. [43] spoof a 2nd fragment, which requires predicting (or guessing) the 16-bit IPv4 ID value of the genuine 1st fragment of the DNS answer (as opposed to the 30-32 bits of the TXID and UDP port combinations in non-fragment attacks). Historically, this made them significantly more practical. However, modern DNS operations have effectively eliminated this attack surface. The current DNS best practices (e.g., Travis Palmer's DefCon 27 talk [39] and the 2020 DNS Flag Day [10]) recommend avoiding DNS fragmentation entirely by limiting the size of DNS UDP responses to e.g. 1232 bytes at the DNS level (advertised through EDNS). For IPv6, this alone suffices to prevent fragmentation since IPv6 mandates a minimal MTU of 1280 bytes (1232 at the DNS level). For IPv4, this can be coupled with instructing the operating system to ignore PMTU and fragment according to the local interface MTU (which is typically larger than 1280, e.g. 1500) – together with the DNS size limit, this ensures

that DNS answers are not fragmented at the origin. Major ANS implementations nowadays employ these measures, and specifically use Linux's `IP_PMTUDISC_OMIT` socket option to this end. In addition, Linux has hardened its fragment-ID generation in 2021: IPv6 fragment IDs became random, and IPv4 fragmentation attacks were further mitigated by enlarging the IPv4 ID table (considerably increasing the attack cost).

A more general approach is suggested by RFC 9715 which specifically states that "UDP requestors should drop fragmented DNS/UDP responses [...] to avoid cache poisoning attacks (at the firewall function)" [12, Section 3.2]. As such, existing fragmentation-based attacks are likely to fail due to having fragments getting detected and dropped by firewalls and anomaly detection systems, in contrast to our attack which does not rely on fragmentation, and which only needs a few spoofed packets.

7.2 Other DNS Cache Poisoning Attacks

Alharbi et al. describe a local DNS cache poisoning attack against DNS stub resolvers based on port exhaustion [3]. Gierlings et al. describe a port exhaustion attack via a malicious Javascript code running in a browser [13]. Klein describes a remote DNS cache poisoning attack against a (Linux) DNS stub resolver based on predicting the UDP source ports used by the stub resolver [30]. None of these attacks targets resolvers or forwarders.

7.3 Breaking Xoshiro128**

O'Neill describes a full internal state reconstruction attack against Xoshiro256** given four consecutive full PRNG outputs [38]. This attack can be easily adapted to Xoshiro128**. However, our RR-QMA attack variant relies on observing *non-consecutive, partial* PRNG outputs (the TXID field exposes the 16 least significant bits out of 32 bits of PRNG output, every 2^{nd} PRNG step). Our reconstruction attack can be generalized to any Xoshiro*NNN*** algorithm, given sufficient (not necessarily consecutive) partial PRNG outputs (8 or more least significant bits).

8 Conclusion

Securing protocol fields is difficult and full of pitfalls. To wit: we find that the most widely deployed open-source DNS server, BIND 9, suffers from a fundamental weakness in this exact area, even though there is no question about ISC's awareness of the problem and its importance. This flaw enables a highly efficient, few-packet cache poisoning attack with success rates approaching certainty under realistic Internet conditions, including heavy load (10,000 *cache miss* QPS), and the presence of stateful firewalls or DNS cookies.

The same weak PRNG that governs port and TXID selection is also used to shuffle RRsets, opening the door to a novel and largely unexplored attack vector: extracting PRNG state from the ordering of large RRsets. This vector enables a pure client-side prediction of both TXID and UDP port – something most prior attacks could not achieve – removing the requirement for attacker-controlled ANS for attacker domains, and significantly lowering operational risk.

We further show that many existing DNS security and privacy mechanisms offer no protection against this class of attacks. Specifically, DNS cookies, anomaly detection mechanisms such as suggested by Afek et al. [2], and qname minimization do not block our methods; in fact, qname minimization can even enhance certain attack variants by introducing near-atomicity to specially-crafted query batches.

Our findings demonstrate that PRNG state recovery -based DNS cache poisoning remains a relevant and impactful threat in 2025 – long after the community assumed such few-packet attacks had been eliminated. All vulnerabilities were responsibly disclosed to ISC and FreeBSD, leading to acknowledgment and two patches and CVEs.

9 Future Work

In this research, we were limited to testing the attack on our own servers, and a handful of open resolvers. An interesting research direction would be to ethically test the attack against production grade resolvers such as ones used in ISPs, enterprises and organizations. Since the crux of the attack is predicting the TXID and UDP source port of an outbound DNS query, the success of a full (and harmful) attack can be forecast from the "inert" version of the attack (merely predicting the query parameters and verifying by forcing a query to the researcher's ANS).

The main challenge we perceive in conducting this future research is access to production-grade resolvers, as this needs to be from within the specific network they serve (the ISP customer network, or the corporate/organization internal network). Obtaining access to these networks at a large scale is not trivial.

The proposed research can teach us about patch propagation, patch coverage, as well as fresh data about BIND 9's market share, perhaps even with a version breakdown. Running a periodic scan can show how these properties change and evolve over time.

Acknowledgments

We thank Oriyan Hermoni, Agam Ebel, Inbal Schussheim, and Noam Caspi for their careful reading of early drafts of this paper and for their many insightful comments that improved both the clarity and technical precision of this work. We thank Oriyan Hermoni for additionally helping us with the artifact testing and improvement. We are grateful to Ondřej Surý (ISC) for his professional collaboration during our coordi-

nated disclosure process and for developing and validating the corresponding BIND patches.

We also thank the anonymous reviewers and our shepherd from USENIX Security '26 for their constructive feedback and guidance, which significantly strengthened the final version of this paper.

Ethical Considerations

Disclosures

We disclosed the security vulnerabilities in BIND to ISC via email on August 19th, 2025. The PRNG vulnerability is tracked as CVE-2025-40780 [21]. ISC issued a patch in October 22nd, 2025 [4], replacing the insecure Xoshiro128** PRNG with a more cryptographically secure alternative (`arc4random()` where available, `uv_random()` elsewhere), see commit `6876753c7ccd`.[3] This patch is incorporated in BIND 9.18.41, 9.20.15 and 9.21.14.

Regarding the RRset-order randomness issue, ISC has publicly acknowledged that BIND's RRset shuffling was never intended to provide uniform randomness [20]. The documentation was updated accordingly in commits `46c88265daa4`[4] and `369c8dc388ca`[5] (included in releases 9.20.13 and 9.21.12), explicitly clarifying that `rrset-order random` does *not* guarantee a uniform distribution. Moreover, ISC decided to deprecate the `random` mode entirely for its future BIND versions: starting in BIND 9.21.14, the `random` option was fully removed from the software.[6]

We disclosed the FreeBSD vulnerability described in App. A to the FreeBSD Security Team via email on August 19th, 2025. This issue is tracked as CVE-2025-24934, and a patch was issued by the FreeBSD Security Team on October 22nd, 2025.[7]

Experiments with live systems without informed consent

In § 5.6 and § 5.7 we describe experiments which access 3rd-party systems. In both cases, the target domain we used was our own domain, so at no point was any 3rd-party domain at risk.

In § 5.6 we accessed five BIND 9.20 resolvers which we do not own and *simulated* an attack against them. As explained in § 5.6, we did *not* attempt to actually poison the cache of these servers, but rather we compared in offline the predicted outbound query's TXID and UDP source port to the values we observed in our own ANSes. All our queries to the resolvers were 100% standard-compliant DNS queries. These "attacker" queries triggered outbound queries from the resolver to our own ANS, thus we had complete control over the answer they receive and process as a result of our queries, and we could guarantee this answer is completely DNS compliant. Furthermore, we specified short TTL for our records in the response, thus making sure our added pressure on the cache is negligible. We took special care not to send more than 1 query per second to any server. Moreover, the total number of queries each server received was 220. The answers received from these servers were as expected and have not indicated any disruption of service. Finally, since our attack ends up in reconstructing the internal PRNG state of these resolvers, we used a specially crafted version of our attack script that does not log or print any data on the internal PRNG state recovered – it only prints the expected TXID and source port predicted for the next few queries, which we then compare to the actual query data that arrives at our ANS.

In § 5.7 we attacked our own resolver, by accessing 10 standard ANSes which we do not own, and one DRINK server (a server running a special kind of authoritative DNS name server software) we do not own. All our queries to these servers were 100% standard-compliant DNS queries, that are completely within the expected protocol and API for the specific services accessed. Furthermore, we took special care not to send more than 1 query per second to any server. Finally, the total number of queries each server received was approximately 10. The answers received from these servers were as expected and have not indicated any disruption of service.

Per the above safety measures and constraints, we consider the risk to 3rd-party systems extremely negligible, and as such our experiments are justified.

Stakeholder analysis

Beyond specifically the five open resolvers and 11 ANSes we experimented with (see above), we identify the following stakeholders:

- **BIND 9 operators:** the BIND 9 patch was clearly announced in the public ISC "bind-announce" mailing list.[8] And since ISC assigned the issue a CVE with a high CVSS score (8.6), and marked its severity as "High" in its accompanying security advisory [4], the issue is unlikely to go unnoticed by ISC BIND operators. We argue that this should compel any reasonable operator to upgrade ASAP. As the paper and the artifact will be released 3+ months after the BIND patch was released, BIND operators have a substantial window (3+ months) in which to deploy the patch, before the attack details become public.

[3] https://gitlab.isc.org/isc-projects/bind9/-/commit/6876753c7ccd67d445a6a2341219fe79cff6c77f

[4] https://gitlab.isc.org/isc-projects/bind9/-/commit/46c88265daa400b49c24abefa6272fb5cbe94cc0

[5] https://gitlab.isc.org/isc-projects/bind9/-/commit/369c8dc388caad0d4fa7e9da15a3a0cd62cd3b39

[6] https://downloads.isc.org/isc/bind9/9.21.14/doc/arm/html/notes.html#removed-features

[7] https://www.freebsd.org/security/advisories/FreeBSD-SA-25:09.netinet.asc

[8] https://lists.isc.org/pipermail/bind-announce/2025-October/001282.html

Therefore, we argue that the bigger operators are very likely to notice the patch via the ISC announcement email and/or via proactive patch/CVE monitoring, and most probably patch within the 3-month window, while the smaller operators are likely to automatically get patched by e.g. OS distribution package updates and upgrades, which pick up 3rd-party security patches on a regular basis, so again, they are likely to get the patch within the said window.

- **Non-DNSSEC domain owners and operators:** theoretically, domain owners can be affected by attacks mounted against un-patched BIND 9 resolvers, targeting their domains. From the argument we present for BIND 9 operators, it follows that in practice, we do not expect domain owners to be affected, due to prompt patching of BIND 9 by its operators.
- **End users/devices that resolve via a BIND 9 resolver/-forwarder:** these can be affected by attacks mounted against an un-patched BIND 9 resolver/forwarder, targeting domains they later request from the same resolver/forwarder. Again, we argue that in practice, by the time the attack becomes public, the BIND 9 resolver/forwarder will already be patched.

Open Science

We release a public artifact accompanying this paper, which contains the full implementation of two variants of our attack and additional documentation needed to reproduce our results. The artifact is available at the time of this paper's publication and is described in detail below.

Artifact Contents and Entry Points. Our artifact is hosted in Zenodo[9]. The artifact contains an attacker client implementing our client-side cache-poisoning attack variants (RRset and RRset-ANY). The client is written in C++20 and Python. A detailed overview and instructions appear in `README.md`.

Targets and Parameters. Experiments are intended for a controlled testbed where the evaluator runs a BIND 9 resolver (victim), an attacker-controlled authoritative nameserver (ANS), an attacker-controlled client, and a genuine ANS. The zone files, configurations, and instructions for each machine are detailed in the README file.

References

[1] D. Eastlake 3rd and M. Andrews. Domain Name System (DNS) Cookies. https://datatracker.ietf.org/doc/html/rfc7873, 2016. RFC 7873.

[2] Yehuda Afek, Harel Berger, and Anat Bremler-Barr. POPS: From History to Mitigation of DNS Cache Poisoning Attacks. In *Proceedings of the 34th USENIX Security Symposium*, Seattle, WA, August 2025. USENIX Association.

[3] F. Alharbi, J. Chang, Y. Zhou, F. Qian, Z. Qian, and N. Abu-Ghazaleh. Collaborative Client-Side DNS Cache Poisoning Attack. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1153–1161, April 2019.

[4] Darren Ankney. CVE-2025-40780: Cache poisoning due to weak PRNG. https://kb.isc.org/docs/cve-2025-40780, October 2025.

[5] David Blackman and Sebastiano Vigna. xoshiro128**. https://prng.di.unimi.it/xoshiro128starstar.c, 2018.

[6] Stephane Bortzmeyer. A dynamic experimental DNS server, just for fun. https://www.bortzmeyer.org/drink.html, May 2022.

[7] Stephane Bortzmeyer, Ralph Dolmans, and Paul E. Hoffman. DNS Query Name Minimisation to Improve Privacy. RFC 9156, November 2021.

[8] Tianxiang Dai, Philipp Jeitner, Haya Shulman, and Michael Waidner. From IP to transport and beyond: cross-layer attacks against applications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pages 836–849, New York, NY, USA, 2021. Association for Computing Machinery.

[9] Jacob Davis and Casey Deccio. A Peek into the DNS Cookie Jar. In *Passive and Active Measurement (PAM)*, pages 302–316. Springer, 2021.

[10] DNS-OARC. DNS flag day 2020. https://www.dnsflagday.net/2020/, 2020.

[11] FreeBSD. FreeBSD Manual Pages (connect). https://man.freebsd.org/cgi/man.cgi?query=connect, 2016.

[12] Kazunori Fujiwara and Paul A. Vixie. IP Fragmentation Avoidance in DNS over UDP. RFC 9715, January 2025.

[13] Matthias Gierlings, Marcus Brinkmann, and Jörg Schwenk. Isolated and exhausted: attacking operating systems via site isolation in the browser. In *Proceedings of the 32nd USENIX Conference on Security Symposium*, SEC '23, USA, 2023. USENIX Association.

[14] Austin Group. The Open Group Base Specifications Issue 8 IEEE Std 1003.1-2024 (connect). https://pubs.opengroup.org/onlinepubs/9799919799/functions/connect.html, 2024.

[9] https://doi.org/10.5281/zenodo.17762025

[15] Olafur Guomundsson. Looking at DNS traces: What do we know about resolvers? https://archive.icann.org/en/meetings/siliconvalley2011/bitcache/Conclusions%20from%20DNS%20Traces%20-%20Olafur%20Gudmunsson,%20Shinkuro-vid=23075&disposition=attachment&op=download.pdf, 2011.

[16] Amir Herzberg and Haya Shulman. Fragmentation Considered Poisonous. *CoRR*, abs/1205.4011, 2012.

[17] Geoff Huston. ICANN DNS Resolver Symposium. https://www.potaroo.net/ispcol/2021-12/dns-sym.pdf, 2021.

[18] Geoff Huston. DNSSEC validation: Performance killer? https://blog.apnic.net/2022/08/22/dnssec-validation-performance-killer/, 2022.

[19] ICANN. ITHI M11: Resolver Behavior – DNSSEC Validation. https://ithi.research.icann.org/graph-m11.html, 2025. Accessed: 2025-10-25.

[20] ISC. BIND9 Issue #5485: RRset-order random distribution clarification. https://gitlab.isc.org/isc-projects/bind9/-/issues/5485. Accessed: 2025-10-25.

[21] ISC. CVE-2025-40780: BIND9 predictable PRNG vulnerability. https://kb.isc.org/docs/cve-2025-40780. Accessed: 2025-10-25.

[22] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris. DNS Performance and the Effectiveness of Caching. In *IEEE/ACM Transactions on Networking*, 2002.

[23] Dan Kaminsky. Black-Ops 2008 – It's The End Of The Cache As We Know It. In *Black Hat USA*, August 2008.

[24] Amit Klein. BIND 8 DNS Cache Poisoning. https://dl.packetstormsecurity.net/papers/attack/BIND_8_DNS_Cache_Poisoning.pdf, 2007.

[25] Amit Klein. BIND 9 DNS Cache Poisoning. https://citeseerx.ist.psu.edu/pdf/0c1e863b6698808b724def8793d7cba023494808, 2007.

[26] Amit Klein. Windows DNS Server Cache Poisoning. https://dl.packetstormsecurity.net/papers/attack/Windows_DNS_Cache_Poisoning.pdf, 2007.

[27] Amit Klein. OpenBSD DNS Cache Poisoning and Multiple O/S Predictable IP ID Vulnerability. https://dl.packetstormsecurity.net/papers/attack/OpenBSD_DNS_Cache_Poisoning_and_Multiple_OS_Predictable_IP_ID_Vulnerability.pdf, February 2008.

[28] Amit Klein. PowerDNS Recursor DNS Cache Poisoning. https://dl.packetstormsecurity.net/papers/attack/PowerDNS_recursor_DNS_Cache_Poisoning.pdf, 2008.

[29] Amit Klein. Hijacking DNS, October 2016. The Hebrew University Cyber Security Center Retreat.

[30] Amit Klein. Cross Layer Attacks and How to Use Them (for DNS Cache Poisoning, Device Tracking and More). In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 927–944, Los Alamitos, CA, USA, May 2021. IEEE Computer Society.

[31] Matthew Luckie, Robert Beverly, Tor Anderson, Ken Keys, and Casey Claffy. Network Hygiene, Incentives, and Regulation: Deployment of Source Address Validation in the Internet. Technical report, University of Waikato, 2019.

[32] Keyu Man, Zhiyun Qian, Zhongjie Wang, Xiaofeng Zheng, Youjun Huang, and Haixin Duan. DNS Cache Poisoning Attack Reloaded: Revolutions with Side Channels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, pages 1337–1350, New York, NY, USA, 2020. Association for Computing Machinery.

[33] Keyu Man, Xin'an Zhou, and Zhiyun Qian. DNS Cache Poisoning Attack: Resurrections with Side Channels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, pages 3400–3414, New York, NY, USA, 2021. Association for Computing Machinery.

[34] Gilad Moav, Yehuda Afek, Anat Bremler-Barr, and Amit Klein. DNS FLaRE: A Flush-Reload Attack on DNS Forwarders. In *34th USENIX Security Symposium (USENIX Security '25)*, pages –, Seattle, WA, USA, August 2025. USENIX Association. Open access; USENIX Security '25, August 13-15, 2025.

[35] P. Mockapetris. Domain Names - Concepts and Facilities. RFC 1034, November 1987. https://www.rfc-editor.org/rfc/rfc1034.

[36] P. Mockapetris. Domain Names - Implementation and Specification. RFC 1035, November 1987. https://www.rfc-editor.org/rfc/rfc1035.

[37] Ondrej Sury. Don't set load-balancing socket option on the UDP connect sockets . https://gitlab.isc.org/isc-projects/bind9/-/commit/b6b7a6886a8ac66bc3932158740998a3bf2da014, 2022.

[38] Melissa E. O'Neill. A Quick Look at Xoshiro256**. https://www.pcg-random.org/posts/a-quick-look-at-xoshiro256.html, May 2018.

[39] Travis Palmer and Brian Somers. "FIRST-TRY" DNS CACHE POISONING WITH IPV4 AND IPV6 FRAGMENTATION. https://media.defcon.org/DEF%20CON%2027/DEF%20CON%2027%20presentations/DEFCON-27-Travis-Palmer-First-try-DNS-Cache-Poisoning-with-IPv4-and-IPv6-Fragmentation.pdf, 2019.

[40] George W. Snedecor, R. A. Fisher, and F. Yates. Statistical Tables for Biological, Agricultural and Medical Research. *Journal of the Royal Statistical Society*, 102:298, 1939.

[41] Trickest. Public List of Open DNS Resolvers. https://github.com/trickest/resolvers/blob/main/resolvers.txt, 2024. Accessed: 2025-08-08.

[42] Masanori Yajima, Daiki Chiba, Yoshiro Yoneya, and Tatsuya Mori. Measuring Adoption of DNS Security Mechanisms with Cross-Sectional Approach. In *IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2021.

[43] Xiaofeng Zheng, Chaoyi Lu, Jian Peng, Qiushi Yang, Dongjie Zhou, Baojun Liu, Keyu Man, Shuang Hao, Haixin Duan, and Zhiyun Qian. Poison Over Troubled Forwarders: A Cache Poisoning Attack Targeting DNS Forwarding Devices. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 577–593. USENIX Association, August 2020.

A The FreeBSD `SO_REUSEPORT_LB` Security Vulnerability

The POSIX/IEEE/Open Group standard documentation for UDP `connect()` [14] states that "if the initiating socket is not connection-mode, then *connect()* shall set the socket's peer address, and no connection is made. For SOCK_DGRAM sockets, the peer address identifies where all datagrams are sent on subsequent *send()* functions, and limits the remote sender for subsequent *recv()* functions." This is also reflected in the FreeBSD documentation for `connect()` [11].

During our experiments with FreeBSD, we noticed a violation of the guarantee to only allow incoming datagrams into the socket from the remote host and port specified in the `connect()` call. This violation occurs when the socket is set to "load balancing" mode prior to an explicit `bind()` for the port, or prior to the implicit bind in a call to `connect()`, via a call to `setsockopt(...,SOL_SOCKET,SO_REUSEPORT_LB,...)`. When this happens, FreeBSD allows datagrams from *any* remote host and port, to the socket, after the `connect()` call.

The attacker can, therefore, send the poisonous DNS response from any source IP address (for example, the attacker's own IP address) and from any port (for example, a non-privileged port, higher than 1024), instead of only from the (spoofed) ANS IP address, and from (privileged) port 53. The implication is that the entire attack can now be performed by a fully unprivileged user, with no raw-socket capability and no permission to bind to low-numbered ports. This significantly lowers the attacker bar and expands the threat model—for example enabling malware or insiders with only unprivileged user accounts to poison a local BIND 9.18 forwarder running on FreeBSD.

The `SO_REUSEPORT_LB` socket option is FreeBSD-specific. The FreeBSD documentation for this socket option does not mention any exceptional behavior in `connect()` as a result of using `SO_REUSEPORT_LB`. The exceptional behavior was not observed for the standard socket options `SO_REUSEPORT` and `SO_REUSEADDR`, neither in FreeBSD nor in Linux.

We noticed that BIND 9.18 sets `SO_REUSEPORT_LB` on its outbound sockets, which use `connect()`. While this makes little sense, since there is no need to load-balance outbound sockets, it is not a bug in BIND per-se (assuming correct behavior of `connect()`).

However, in this case, `connect()`'s behavior is incorrect and insecure, and as such it has implication on the entire system. In short, given a "classic" remote DNS cache poisoning attack against BIND 9.18 (as described in this paper), it allows remotely poisoning the cache of BIND 9.18 running on FreeBSD, *without source IP address spoofing and without binding to privileged ports*. Here are three concrete scenarios where this is advantageous:

- **Unprivileged attacker scenario.** Assume a network topology wherein one company branch office (internal network) has a BIND 9.18 forwarder forwarding queries to a central resolver in the company HQ, over private network connections (VPN). Consider an attacker in the branch office who does not have administrator or root privileges (or `CAP_NET_RAW`/`CAP_NET_ADMIN` capability). This can be malware running without root permissions, or an attacker with an unprivileged account. Such an attacker cannot spoof packets from the machine he/she has limited access to (and no packets, spoofed or otherwise, can arrive at the BIND 9.18 forwarder from the Internet!). But now the attack requires neither source address spoofing nor binding to privileged ports, so the attacker can mount a successful attack against the BIND 9.18 forwarder running on FreeBSD, because other than source address spoofing and binding to privileged ports, our DNS cache poisoning attack needs no privileged operations (just standard TCP/UDP sockets).

- **General attack considerations.** Even privileged attackers, e.g. attacking Internet-facing BIND 9.18 resolvers from their own machines, may be limited by network source address verification filtering. For these attackers, doing away with the need to spoof the source address

makes their operational logistics much easier.

- **Reducing the entropy of authoritative server choice.** Having multiple ANSes (IP addresses) for a domain increases the entropy the attacker faces when attempting to spoof the answer from the ANS, as he/she may not be able to accurately predict which ANS the resolver will access. For BIND 9.18 on FreeBSD, however, the attacker does not need to spoof the ANS's IP address in order to respond to the query, thus the entropy usually gained by having multiple name servers is reduced to zero.

It should be noted that only BIND 9.18 sets the `SO_REUSEPORT_LB` option on its outbound UDP sockets. In BIND 9.20 and above this logic is deliberately removed, reasoning that "this socket option makes only sense for the listening sockets" [37]. In other words, BIND 9.20 and 9.21 are not affected, but not due to any recognition that setting `SO_REUSEPORT_LB` on connected UDP sockets is a security issue, but rather, due to recognition that the setup makes no sense for outbound sockets. To wit, this logic remains in BIND 9.18 to this day.

B Adapting the RR-QMA Attack to BIND 9.18

In this appendix we explain how to adapt the RR-QMA attack to BIND 9.18. We first explain about BIND 9.18's threading model and buckets, then we describe the adaptation, and finally we elaborate on the specific issue of a halting condition for the PRNG instance search.

B.1 BIND 9.18's Threading Model

Unlike later versions of BIND, BIND 9.18 has two types of threads: listener threads and worker threads. Upon startup, BIND 9.18 determines the number of logical CPU cores ("threading cores") and instantiates that many copies of its listener threads and worker threads. Each thread maintains a dedicated instance of the PRNG.

Incoming datagrams are dispatched to *listener threads* by the operating system based on a hash of their transport-level 4-tuple (source IP address, source port, destination IP address, destination port). Listener threads parse these datagrams and forward the queries to *worker threads*, selected via a keyed hash of the query name. Worker threads issue outbound queries – this is where TXID and UDP source port generation occurs. Once a response is received, it is returned to the originating listener thread, which constructs the final answer (including RRset order shuffling). Because each thread type maintains its own independent PRNG instance, PRNG outputs (e.g., TXIDs and ports) generated in worker threads cannot be predicted from RRset orders produced by listener threads.

B.2 BIND 9.18's Internal Queues

In BIND 9.18, internal queues known as "buckets" are used to dispatch queries from listener threads to worker threads. Each incoming query name is hashed and deterministically assigned to one of these buckets. The buckets act as FIFO queues and serve as the mechanism for routing queries between listeners and workers.

Specifically, the assignment is done by casting the hash of the query name into the range $[0, m-1]$, where m is the total number of buckets. Each bucket is owned by a single worker thread, and each worker typically manages multiple buckets (historically 32 until version 9.18.33, reduced to 2-3 since 9.18.35). A listener thread places the query into the appropriate bucket, and the owning worker thread processes queries in strict FIFO order.

B.3 Adapting the RR-QMA Attack to BIND 9.18

In the initial phase, the attacker breaks the PRNG seed for each worker thread, associating one representative domain per thread. A *representative domain* is a domain name that maps to a specific resolver worker thread and is therefore resolved using a known PRNG instance. This mapping is later used to help identify which thread is responsible for resolving the target domain. This ensures that the attacker can accurately predict the TXID and UDP port used for the subsequent target query.

Mapping all worker threads is done in a series of distinct rounds, where in each round the attacker reconstructs the PRNG state of a single worker thread. In each round, the attacker sends 20 queries with distinct query types for a single FQDN in an attacker domain. Because BIND 9.18 assigns queries to worker threads based solely on the query name, using the same name guarantees that all queries map to the same BIND *worker* thread and PRNG instance. By analyzing the resolver's outbound queries to the attacker's ANS, the attacker reconstructs the worker thread's 128-bit PRNG internal state. The attacker conducts multiple rounds, each with distinct query names, until all worker threads' PRNG states are identified. Each newly reconstructed PRNG state is compared to previously found states, advanced several million steps forward. If the new state matches any state in the series obtained from an already broken PRNG (worker thread), the attacker identifies it as belonging to an already discovered thread; otherwise, it is deemed to belong to an unseen-before thread. The attacker continues until no new threads are discovered after a sufficient number of additional rounds, ensuring with high confidence that all worker threads have been identified (see App. B.4).

Next, for each identified worker thread, the attacker must find a *validated pair* – two representative domains served by the thread but mapped to two distinct internal buckets

in BIND. To achieve this, the attacker generates a sequence (stream) of candidate domains mapped to the targeted worker thread. Each candidate domain pair is tested by sending alternating queries (e.g., `x1,x2,x1,x2,...`) and analyzing the order of outbound queries the resolver sends to the attacker's ANS. Queries are sorted by their original transmission order, and the PRNG state is rolled forward until matching the observed TXID from the first query. Subsequent queries are checked by advancing the PRNG twice per query and verifying consistency with observed TXIDs. If the observed order differs from the predicted order, it indicates that the domains map to distinct buckets (since FIFO order is always maintained if they belong to the same bucket) and a *validated pair* is found. Otherwise, the attacker discards the 2nd domain candidate and proceeds with the next candidate.

Subsequently, the attacker identifies which validated domain pair corresponds to the same worker thread as the target domain `www.target.example` (which we now refer to as `g` for brevity). For each candidate domain pair e.g. (`x1`,`x2`) the attacker sends three structured sets of 20 queries each: `x1,g,g,...,g,x1`, then `x2,g,g,...,g,x2`, and finally `x1,g,g,...,g,x2`. Each set includes exactly 18 queries for the domain `g`, resulting in precisely 36 PRNG invocations (18 TXIDs and 18 UDP source ports) for the `g` queries. By aligning the PRNG state with the TXID and UDP source port observed for the initial query and advancing it through subsequent queries, the attacker measures PRNG step distances between the `x` queries in each set. If `g` belongs to the same bucket as `x1` then it is guaranteed to form a gap of 36 PRNG steps in the first set. Likewise with `x2` and the second set. Finally if `g` is served by the same thread but does not share a bucket neither with `x1` nor with `x2` then it is not guaranteed that `g` forms a gap even if it is served by the same thread; however, if a large gap (36 steps) between `x1` and `x2` is observed we can assume that `g` formed it and thus `g` is served from the same thread. Therefore, a gap exactly corresponding to 36 PRNG steps in any set indicates the domain pair `x1,x2` shares the worker thread with the target domain `g`. Once the pair is found, the attacker marks the representative domain (e.g. `x1`) associated with the target domain's worker thread.

Finally, the attacker performs cache poisoning as described in § 4.5. The representative domain for the target domain's worker thread is queried first to synchronize with the PRNG state. The target domain is queried immediately afterward. This ensures that the PRNG instance used for the target domain is the same one whose state has just been recovered, which is crucial for being able to predict the actual TXID and UDP port values. With the PRNG state known, the attacker crafts and injects a spoofed response matching the expected TXID and UDP port, which the resolver accepts and caches.

This attack variant does not depend on RRset randomization. Instead, it exploits BIND 9.18's deterministic mapping of queries to worker threads and buckets, combined with the predictability of the thread-local PRNG. It assumes control over an ANS and visibility into queries directed at it. Specifically, the TXID is observed and used, but optionally the UDP source port may be needed (see App. D).

B.4 A Halting Condition for the PRNG Instance Search

An important technical detail omitted from the above discussion is the halting condition for the PRNG instance search. Since the attacker does not know how many worker threads the target resolver runs, the attacker has to somehow decide when to stop trying to find new PRNG instances. Any such strategy is bound to have false negatives, even if individual state recovery attempts are accurate, so the challenge is to ensure the false negative rate is low enough. A simple and effective strategy is, given n observed unique PRNG instances, to keep polling R_n more – if a new instance is discovered, restart the strategy (with $n+1$ and R_{n+1}). If no new instances are found, determine that there are only n instances and stop.

Thus the challenge becomes: given n observed unique threads (PRNGs), we would like to calculate R_n – the number of iterations in which we obtain PRNG state from a random unknown thread which conforms to the n observed unique PRNGs that are needed in order to determine, with fail probability no higher than p, that there are no more unseen PRNGs.

The failure probability for a situation wherein there are actually $n+m$ PRNGs is $(\frac{n}{n+m})^{R_n}$. Clearly the worst case is $m=1$, therefore we need $(1-\frac{1}{n+1})^{R_n} \leq p$. If n is large, we can use the standard approximation $(1-\frac{1}{n+1})^{n+1} \approx e^{-1}$, to obtain $e^{-\frac{R_n}{n+1}} \leq p$ and finally $R_n \geq (n+1)(-\ln(p))$. In other words, if we already observed n PRNGs, and we now further observe $(n+1)(-\ln(p))$ same PRNGs, we can conclude that there are only n PRNGs with error probability bounded by p.

For example, with $n=4$ threads and a desired $P_{\text{fail}} \leq 0.01$, we require:

$$R_4 \geq 5 \cdot \ln(10^2) \approx 5 \cdot 4.605 \approx 23,$$

i.e., 23 independent PRNG-breaking rounds to drive the total failure probability below 0.01.

C Inferring Resolver's Ephemeral Port Range Using DRINK

C.1 Description

BIND inherits its outbound UDP source port range directly from the operating system's ephemeral port range. By default, Linux uses 32768-60999 and FreeBSD uses 49152-65535. Accordingly, the DRINK-based port range inference phase is only required when the operating system or its configuration is unknown. If the default range can be assumed, or if the attacker can infer it by other means (e.g., prior observation), explicit inference is unnecessary (see also § 4.4).

In client-side-only attacks, the attacker cannot directly observe the resolver's outbound UDP ports. Instead, this limitation can be overcome by querying ANSes that reflect the resolver's source port in their DNS responses. Our experiments use an ANS running the DRINK server software [6], but other equivalent mechanisms could be employed with minimal adaptation.

Using these services, the attacker faces the challenge that these queries return only the UDP source ports, but not the corresponding TXIDs (in contrast to the use case in App. D). To overcome this, the attacker issues a sequence of specially crafted queries. Specifically, the attacker sends alternating queries from a single IP-port pair, forming a structured sequence $r,u,r,u,r,u,\ldots,u,r$, where:

- r is an RRset-order-revealing query to a cached RRset of size 23+, allowing the attacker to reconstruct 32 PRNG state bits from the observed RRset order, which are used to re-sync the PRNG state to the one used for the r query.
- u is a UDP-port-revealing query to a UDP-echoing service such as a DRINK server.

Because queries from the same IP-port pair consistently map to the same resolver worker thread (in BIND 9.20 and 9.21), each observed UDP source port from u queries corresponds to a PRNG output from the same PRNG instance. However, due to potential interleaved queries from other resolver clients, each u query may follow an unknown number of PRNG invocations.

The attacker addresses this by using the surrounding RRset order -revealing r queries to constrain the PRNG state between successive u queries. Specifically, each u query is "sandwiched" between two r queries, significantly reducing the set of possible PRNG states. Thus, for each observed UDP source port (from the DRINK responses), the attacker obtains a limited set of candidate PRNG values (in practice we cap this to 2000 values to further limit the search space; we observed no negative impact on our results due to this shortcut). By repeating this structured querying multiple times, the attacker accumulates multiple sets of candidates.

BIND calculates a UDP source port p from a PRNG output x as:

$$p = b + \mathrm{cast}(x,n)$$

where $[b, b+n-1]$ is the port range from which BIND picks the port, and $\mathrm{cast}(x,n)$ is the function used by BIND to map the PRNG output into $[0,\ldots,n-1]$.

To infer the port range, the attacker proceeds as follows:

1. Compute the set of possible range sizes based on observed ports. Let $p_{\min}$ and $p_{\max}$ be the minimal and maximal port numbers observed, respectively. Let $n_{\min} = \max(p_{\max} - p_{\min} + 1,\ 10000)$ and $n_{\max} = 65536$.
2. For each candidate range size n':
   - For each candidate PRNG output x and its corresponding observed port p, compute $b' = p - \mathrm{cast}(x,n')$. Discard negative b' values.
   - Collect these values into a set B_u for each u query.
   - If $\bigcap B_u = \{b'\}$ is a singleton for some n', conclude $n = n'$, $b = b'$.

This method is highly reliable even under load. The attacker can precisely bound the false positive probability by adjusting the number of observations and candidate PRNG steps.

C.2 Analysis

In the DRINK-based ephemeral port range inference, false positives occur if all observed ports are consistent with an incorrect (b,n) pair.

Let n be the (unknown) ephemeral-port range size, X the number of `isc_random_uniform` candidates per a readout u, and K the number of independent u readouts. The attacker first chooses a minimal X that yields sufficiently high probability that the *correct* PRNG states for *all* observed UDP ports will appear within X PRNG steps of the PRNG state extracted from the RRset order of the preceding query. Modeling the port selection retry in a multi-threaded environment is difficult, so we resort to empirical results: we find that setting $X = 2000$ provides de-facto 100% success for a rate of 10,000 outbound queries per second, thus we use this value.

Next the attacker needs to determine the optimal K. For a given n, the probability that a specific b in the valid range $0 \le b \le 65536 - n$ appears at least once in a single u is bounded from above (because the ranges of b candidates, $[\mathrm{port}_j - (n-1), \mathrm{port}_j]$ do not perfectly overlap across the observations) by:

$$Prob(b \text{ in a specific } u) \;=\; 1 - \left(1 - \frac{1}{n}\right)^X \;\approx\; 1 - e^{-X/n}.$$

Requiring appearance in *all* K readouts gives

$$p_b = \left(1 - e^{-X/n}\right)^K.$$

Naturally we choose K sufficiently large so that p_b is small, thus we will be able to use the approximation $(1-p_b)^{\frac{1}{p_b}} \approx e^{-1}$.

Over all $65536 - n + 1$ possible b values, the probability that *some* b appears in all K readouts (i.e., a false positive for that n) is:

$$\begin{aligned} Prob(\mathrm{FP} \mid n) \;&=\; 1 - (1-p_b)^{65536-n+1} \\ &\approx\; 1 - \exp\Big(-(65536-n+1)\left(1 - e^{-X/n}\right)^K\Big). \end{aligned}$$

It is easy to see that $Prob(\mathrm{FP} \mid n)$ is monotonically decreasing in n. Therefore we can bound $Prob(\mathrm{FP} \mid n)$ from above by $Prob(\mathrm{FP} \mid n_{\min})$. Enumerating M candidate n values (with

$M = 65536 - n_{\min}$ when $n \in [n_{\min}, 65535]$) and treating them as approximately independent, the overall false positive probability is bound from above by:

$$Prob(\text{FP}_{\text{overall}}) \approx 1 - \exp\Big(-M\,(65536 - n_{\min} + 1) \times \Big(1 - e^{-X/n_{\min}}\Big)^K \Big).$$

Numerical Example. With $X = 2000$, $n_{\min} = 10000$, $K = 16$, we have $65536 - n_{\min} + 1 = 55537$, $M = 65536 - 10000 = 55536$, and

$$(1 - e^{-X/n_{\min}})^K \approx 2.1476 \times 10^{-6},$$

so

$$Prob(\text{FP}_{\text{overall}}) \approx 1 - \exp\Big(-55536 \times 55537 \times 2.1476 \times 10^{-6} \Big) \approx 2.09 \times 10^{-2}.$$

This is a coarse upper bound. In practice, we encounter near-n values of n_{min}, so the probability is actually much lower. Thus both per-n and overall false positive probabilities remain small in practice.

D Inferring Resolver's Ephemeral Port Range in RR-QMA Attacks

Resolvers may be configured with nonstandard ephemeral UDP port ranges, making accurate range identification essential for predicting outbound ports and mounting reliable cache poisoning attacks. App. C covers the more complex client-side case, whereas here we focus on the simpler server-side case where the attacker has authoritative visibility.

If the attacker controls an ANS, then the attacker can directly observe both the TXIDs and the UDP source ports of incoming resolver queries. After reconstructing the PRNG state, the attacker correlates it with the observed ports to determine the resolver's port range.

To validate a candidate port range n, the attacker checks whether the differences between observed ports match the differences in casted PRNG outputs, i.e.,

$$\text{port}_{j+1} - \text{port}_j = \text{cast}(x_{j+1}, n) - \text{cast}(x_j, n)$$

must hold for multiple successive queries. Once a consistent n is found, the base of the port range is derived as

$$b = \text{port}_j - \text{cast}(x_j, n).$$

To ensure robustness, the attacker repeats this across several observations to confirm that the same pair (b, n) appears consistently. This guards against false positives caused by unrelated PRNG steps (e.g., from other concurrent queries). If the set of pairs is inconsistent, the attacker discards it and restarts with new observations.

This method remains effective even under resolver load. As long as multiple consistent observations are collected, the risk of false positives remains negligible. Thus, the RR-QMA attack variant can be reliably deployed even against resolvers with customized UDP source port ranges.

E Impact of UDP unavailable ports

When the UDP port number generated by BIND's PRNG is already in use, BIND retries with another random value. This retry disrupts the expected PRNG sequence, and in many contexts – particularly when precise PRNG state tracking is required – this behavior undermines our prediction accuracy. The effect is exacerbated on heavily loaded machines, where the high number concurrent outstanding queries increases the probability of collisions. On average, the number of unavailable ports at any moment can be estimated as $L \cdot RTT_{\text{ANS}}$ where L is the outbound query load in QPS units and RTT_{ANS} is the round-trip time between the resolver and the load ANS. The probability that a generated UDP port is unavailable is therefore $\frac{L \cdot RTT_{\text{ANS}}}{R_{\text{ephemeral}}}$ where $R_{\text{ephemeral}}$ is the size of the ephemeral port range configured on the system. On Linux, $R_{\text{ephemeral}}$ is typically 28232 (ports 32768-60999). At high L values or with larger RTT_{ANS}, this probability becomes non-negligible, making retries more frequent and the PRNG step sequence less predictable. It bounds from above the success rate of a single attack attempt. additionally, it necessitates larger search window X in the optional client port range inference step (App. C), to compensate for the occasional skipped or replaced PRNG outputs.