Iceberg

Automated Verification of DNS Authoritative Engines via Just-in-Time Summarization

Yuxing Xiang, Rilin Huang, Naiqian Zheng, Xin Jin

Peking University

NSDI'26

DNS: critical infrastructure

- Every website visit, email, or API call starts with a DNS lookup.

DNS Authoritative Engines

BIND 9

CLOUDFLARE

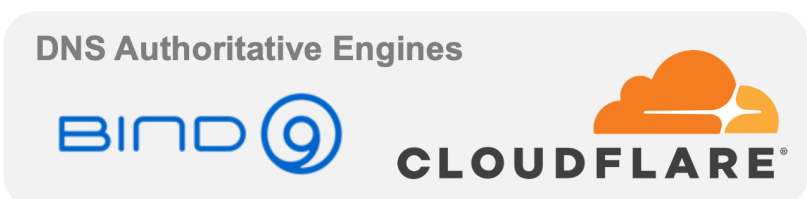
```
;; QUESTION SECTION:
;google.com.      IN  A

;; ANSWER SECTION:
google.com.   15  IN  A 142.250.197.46

;; Query time: 6 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Tue Apr 28 17:30:34 CST 2026
;; MSG SIZE  rcvd: 54
```

DNS: critical infrastructure

- Every website visit, email, or API call starts with a DNS lookup.
- At the heart of DNS are **authoritative engines** — software that answers queries by looking up records in configured zones.




```
;; QUESTION SECTION:
;google.com.      IN  A

;; ANSWER SECTION:
google.com.   15  IN  A 142.250.197.46

;; Query time: 6 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Tue Apr 28 17:30:34 CST 2026
;; MSG SIZE  rcvd: 54
```

DNS engine bugs cause real outages

DNS engine bugs cause real outages

Oct 2025

AWS us-east-1: DNS bug cascades into 15-hour Internet outage

How Tiny DNS Mistake That Crashed AWS: Inside the Outage That Shook the Internet

Aman Pathak | DevOps | AWS | K8s | Terraform | ML Follow 5 min read · Oct 21, 2025

aws

Summary of the Amazon DynamoDB Service Disruption in the Northern Virginia (US-EAST-1) Region

DNS engine bugs cause real outages

Oct 2025 — AWS us-east-1: DNS bug cascades into 15-hour Internet outage

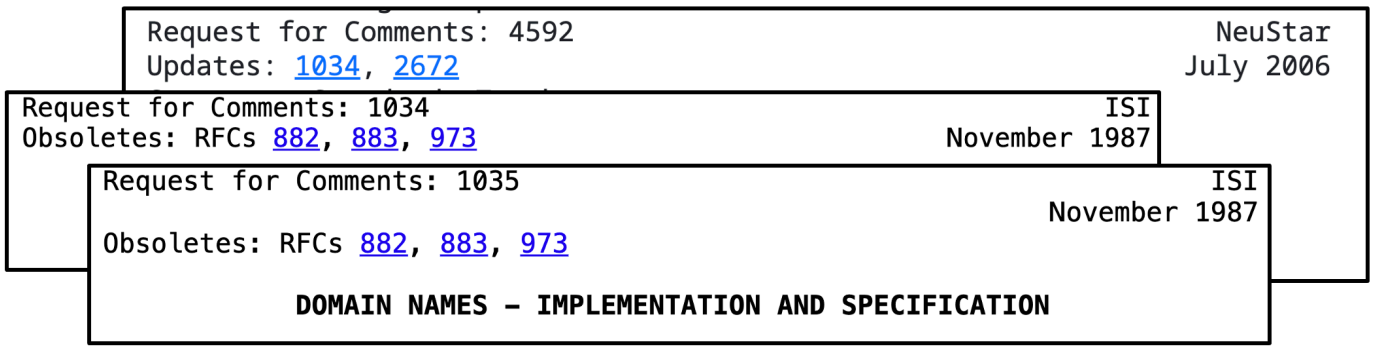
Oct 2021 — .club / .hsbc SERVFAIL for 3 hours — authoritative engine bug

Oct 2021 — Facebook 6-hour outage — DNS unreachable compounded by BGP issue

2018 / 2021 — Google, Slack — DNS-related outages

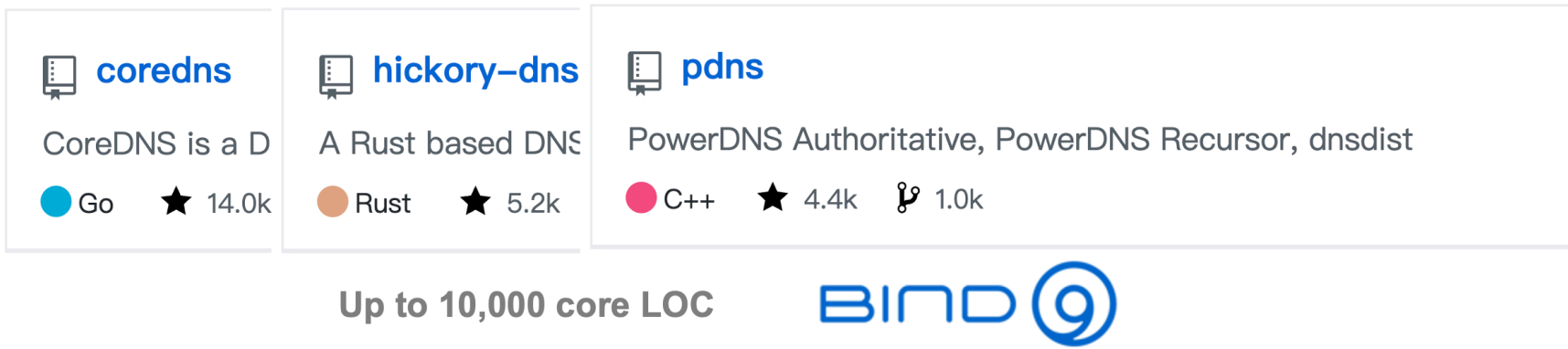
Why is correctness hard?

Why is correctness hard?



Complex protocol — dozens of RFCs; record types, wildcards, redirections, glue all interact

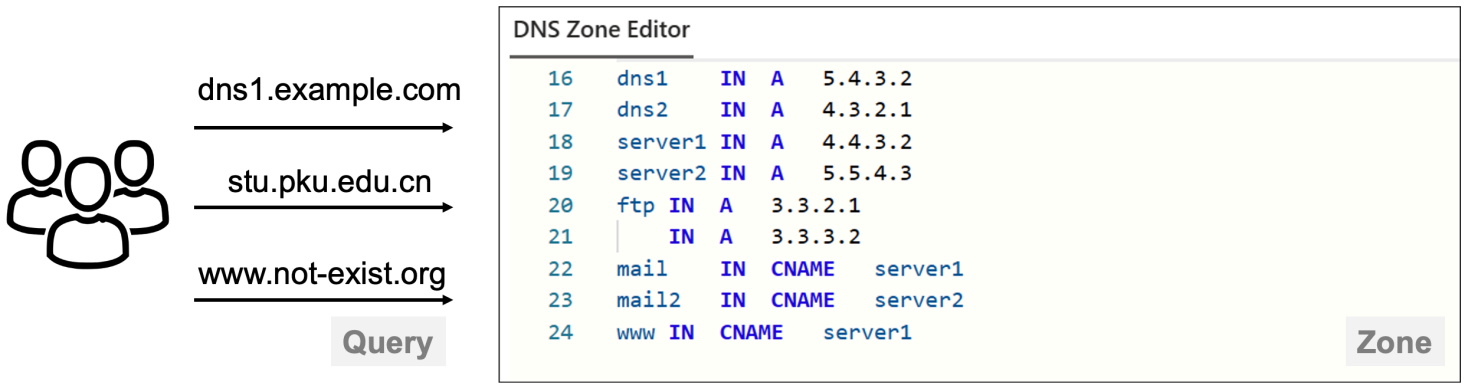
Why is correctness hard?



Complex protocol — dozens of RFCs; record types, wildcards, redirections, glue all interact

Large, diverse implementations — 2K–10K lines of core logic in Go / C / C++ / Rust

Why is correctness hard?

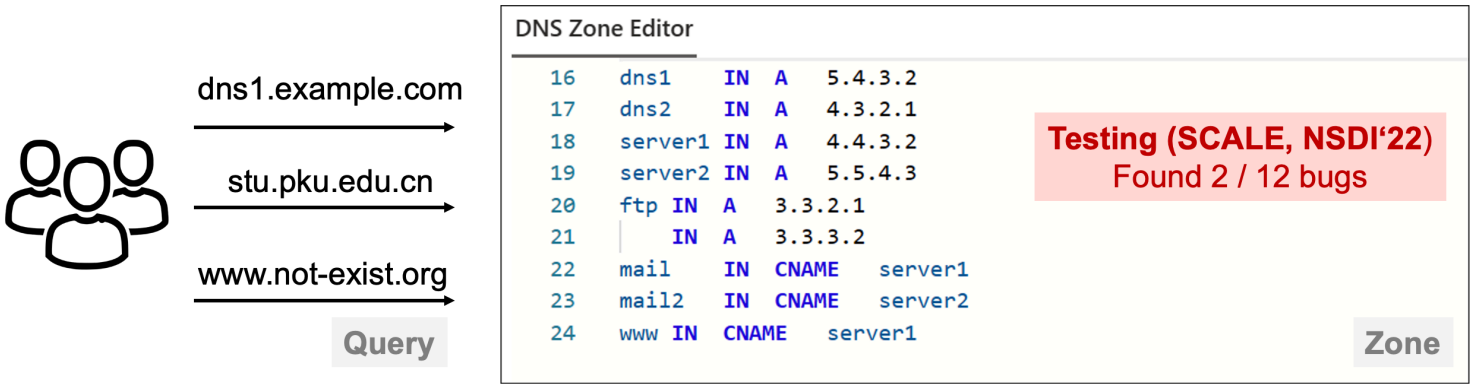


Complex protocol — dozens of RFCs; record types, wildcards, redirections, glue all interact

Large, diverse implementations — 2K–10K lines of core logic in Go / C / C++ / Rust

Testing is limited — query × zone input space is too large for full coverage

Why is correctness hard?



Complex protocol — dozens of RFCs; record types, wildcards, redirections, glue all interact

Large, diverse implementations — 2K–10K lines of core logic in Go / C / C++ / Rust

Testing is limited — query × zone input space is too large for full coverage

Verification can guarantee correctness

www.cs.columbia.edu/~rgu/assets/media/certikos_layer.jpg

- **Formal verification** proves correctness against a **specification**, for **all** inputs

Verification can guarantee correctness

www.cs.columbia.edu/~rgu/assets/media/certikos_layer.jpg

- **Formal verification** proves correctness against a **specification**, for **all** inputs
- **Refinement proof**: prove by layers, with layer-wise specs

Verification can guarantee correctness

www.cs.columbia.edu/~rgu/assets/media/certikos_layer.jpg

- **Formal verification** proves correctness against a **specification**, for **all** inputs
- **Refinement proof**: prove by layers, with layer-wise specs

> **But:** Refinement proof is a lot of manual work!

Prior work partially automated this

DNS-V (SOSP'23) verified an in-production engine at Alibaba Cloud using **ahead-of-time (AOT) summarization**.

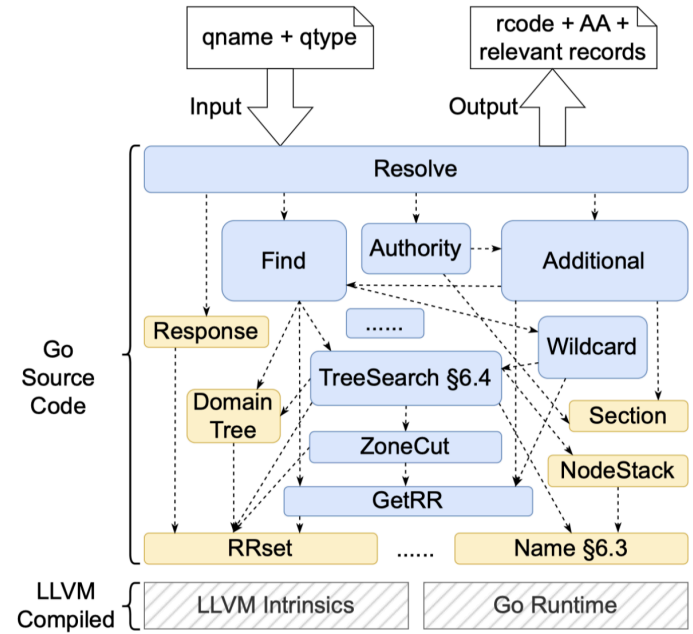


Figure from DNS-V (SOSP'23) talk slides

Prior work partially automated this

DNS-V (SOSP'23) verified an in-production engine at Alibaba Cloud using **ahead-of-time (AOT) summarization**.

- Higher-level modules → automatically **summarize** via symbolic execution.

```
if (nameLen != 0 && n0 == int("com")){
    match_result := EXACT;
    match_NodePtr := NODE("com.");
}                                          Path Condition
else if (nameLen != 0 && n0 != int("com")){
    match_result := NOMATCH;
    match_NodePtr := NULL_NODE;            Effect
}
else {
    match_result := WILDCARD;
    match_NodePtr := NODE(".");
}
                                           Specification
```

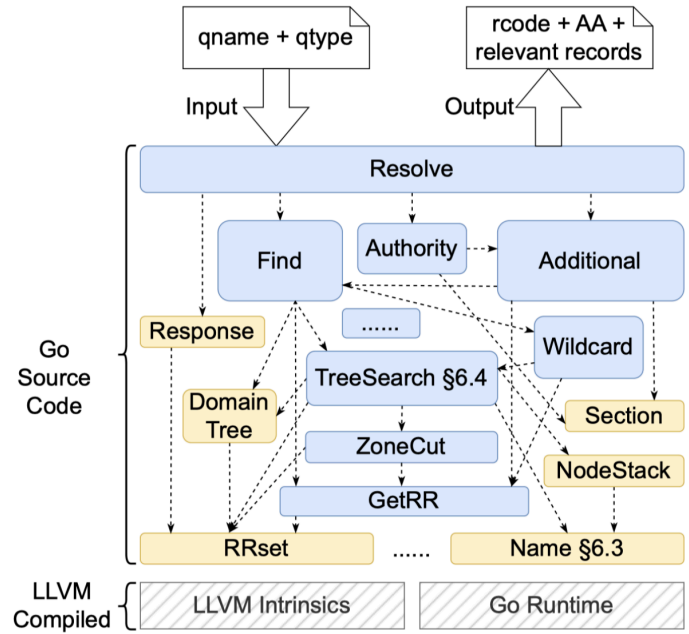


Figure from DNS-V (SOSP'23) talk slides

Prior work partially automated this

DNS-V (SOSP'23) verified an in-production engine at Alibaba Cloud using **ahead-of-time (AOT) summarization**.

- Higher-level modules → automatically **summarize** via symbolic execution.
- Low-level modules (byte ops, data structures) → still need **manual specifications**.

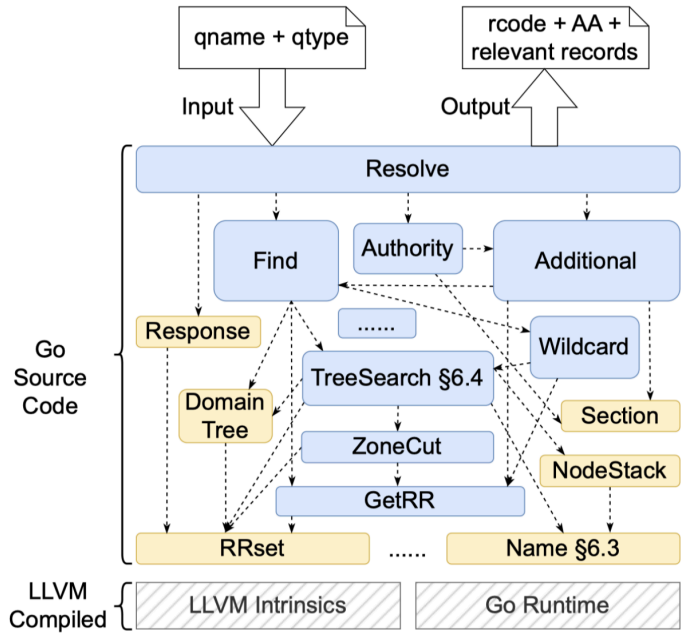


Figure from DNS-V (SOSP'23) talk slides

Manual effort: still too high

Manual effort: still too high

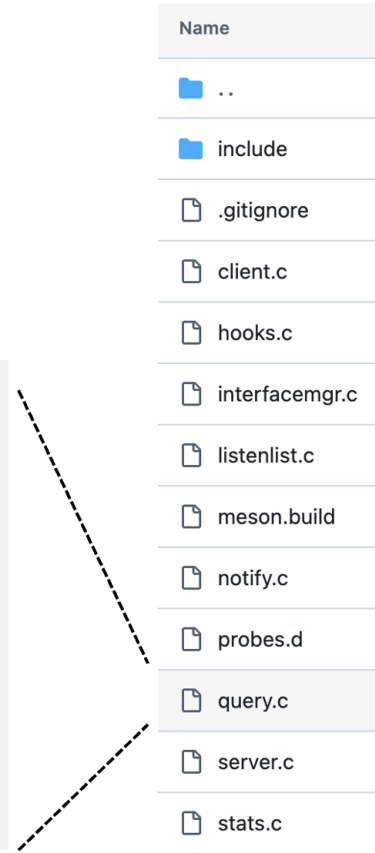
Manual Effort Estimation (% of functions)

CoreDNS	Bind 9	PowerDNS
~30 / 79 (38%)	~200 / 314 (64%)	~50 / 114 (44%)

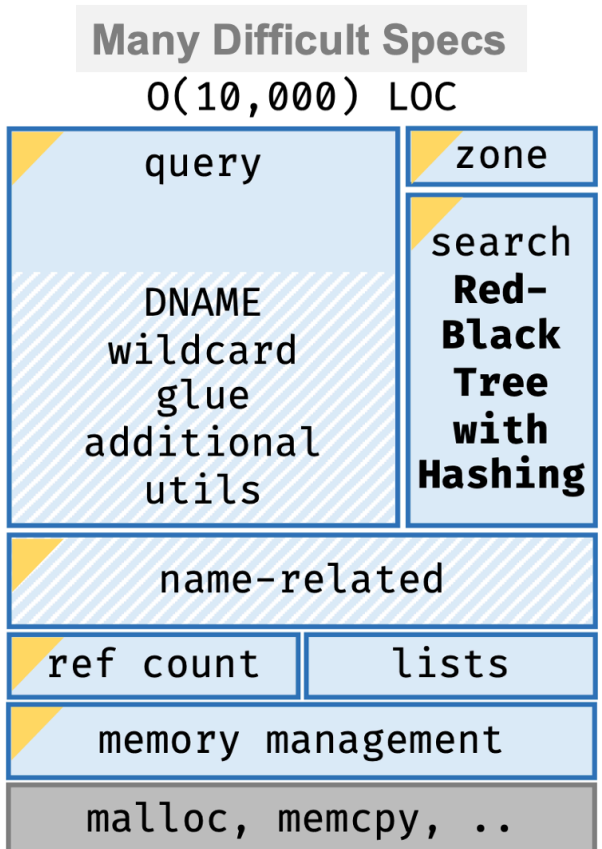
Bind 9 – entry point

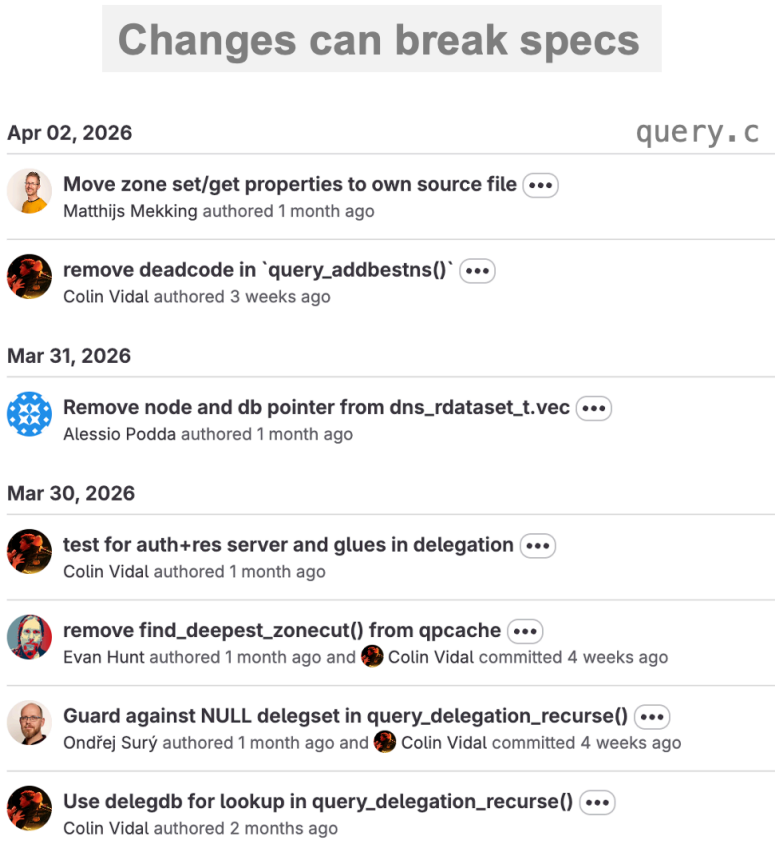
```
5364  /*%
5365   * Starting point for a client query or a chaining query.
5366   *
5367   * Called first by query_setup(), and then again as often as needed to
5368   * follow a CNAME chain.  Determines which authoritative database to
5369   * search, then hands off processing to query_lookup().
5370   */
5371  isc_result_t
5372  ns__query_start(query_ctx_t *qctx) {
5373    isc_result_t result = ISC_R_UNSET;
5374    CCTRACE(ISC_LOG_DEBUG(3), "ns__query_start");
5375    ...
5376  }
```

Core Engine Logic
O(10,000) LOC



Manual effort: still too high





Can we **guarantee** correctness with **low manual effort**?

Can we **guarantee** correctness with **low manual effort**?

Iceberg — automated verification via **just-in-time summarization**

Insight: invariants from DNS zones

- An engine always operates on **concrete domain names and records**

```
example.com.  3600  IN  SOA   ns.example.com. a
    2025081301 ; serial
    3600       ; refresh
    900        ; retry
    1209600    ; expire
    60         ; minimum
)

example.com.        3600  IN  NS    ns.example.

example.com.        3600  IN  A     192.0.2.1
example.com.        3600  IN  AAAA  2001:db8:10
ns.example.com.     3600  IN  A     192.0.2.2
ns.example.com.     3600  IN  AAAA  2001:db8:10

www.example.com.    3600  IN  CNAME example.com
```

Insight: invariants from DNS zones

- An engine always operates on **concrete domain names and records**
- Zone data propagates through the code as **invariants**:
  - Fixed function arguments
  - Constant memory contents
  - Bounded data structures

```
example.com.  3600  IN  SOA   ns.example.com. a
    2025081301 ; serial
    3600       ; refresh
    900        ; retry
    1209600    ; expire
    60         ; minimum
)

example.com.        3600  IN  NS    ns.example.

example.com.        3600  IN  A     192.0.2.1
example.com.        3600  IN  AAAA  2001:db8:10
ns.example.com.     3600  IN  A     192.0.2.2
ns.example.com.     3600  IN  AAAA  2001:db8:10

www.example.com.    3600  IN  CNAME example.com
```

Insight: invariants from DNS zones

- An engine always operates on **concrete domain names and records**
- Zone data propagates through the code as **invariants**:
  - Fixed function arguments
  - Constant memory contents
  - Bounded data structures

> These invariants can **dramatically simplify** what we need to summarize.

```
example.com.  3600  IN  SOA   ns.example.com. a
    2025081301 ; serial
    3600       ; refresh
    900        ; retry
    1209600    ; expire
    60         ; minimum
)

example.com.        3600  IN  NS    ns.example.

example.com.        3600  IN  A     192.0.2.1
example.com.        3600  IN  AAAA  2001:db8:10
ns.example.com.     3600  IN  A     192.0.2.2
ns.example.com.     3600  IN  AAAA  2001:db8:10

www.example.com.    3600  IN  CNAME example.com
```

Example: domain name comparison (Bind 9)

```
1   // RFC-compliant domain name format
2   typedef struct {
3     unsigned char      *ndata;    // binary data
4     unsigned int        labels;   // number of labels
5     unsigned char      *offsets;  // offset of labels
6   } name_t; /* other fields omitted */
7
8   namereln_t name_fullcomp(name_t *n1, name_t *n2) {
9     /* definitions omitted */
10    nlabels = 0;
11    ofs1    = n1->offsets;  //|
12    ofs2    = n2->offsets;  //| get offset pointers
13    l       = min(n1->labels, n2->labels);
14    ldiff   = n1->labels - n2->labels;
```

Example: domain name comparison (Bind 9)

```
1   // RFC-compliant domain name format
2   typedef struct {
3     unsigned char      *ndata;    // binary data
4     unsigned int        labels;   // number of labels
5     unsigned char      *offsets;  // offset of labels
6   } name_t; /* other fields omitted */
7
8   namereln_t name_fullcomp(name_t *n1, name_t *n2) {
9     /* definitions omitted */
10    nlabels = 0;
11    ofs1    = n1->offsets;  //|
12    ofs2    = n2->offsets;  //| get offset pointers
13    l       = min(n1->labels, n2->labels);
14    ldiff   = n1->labels - n2->labels;
```

Example: domain name comparison (Bind 9)

```
23        c2    = *p2++;     //| and move pointer
24        cdiff = (int)c1 - (int)c2;          //|
25        c1    = cdiff < 0 ? c1: c2;         //|
26        diff  = ascii_lowercmp(p1, p2, c);  //| compare
27        if (diff != 0 || cdiff != 0) goto done;
28        nlabels++;
29      }
30      if (ldiff < 0)      return namereln_contains;
31      else if (ldiff > 0) return namereln_subdomain;
32      else                return namereln_equal;
33    done:
34      if (nlabels > 0)    return namereln_commonancestor;
35      return namereln_none;
36    }
```

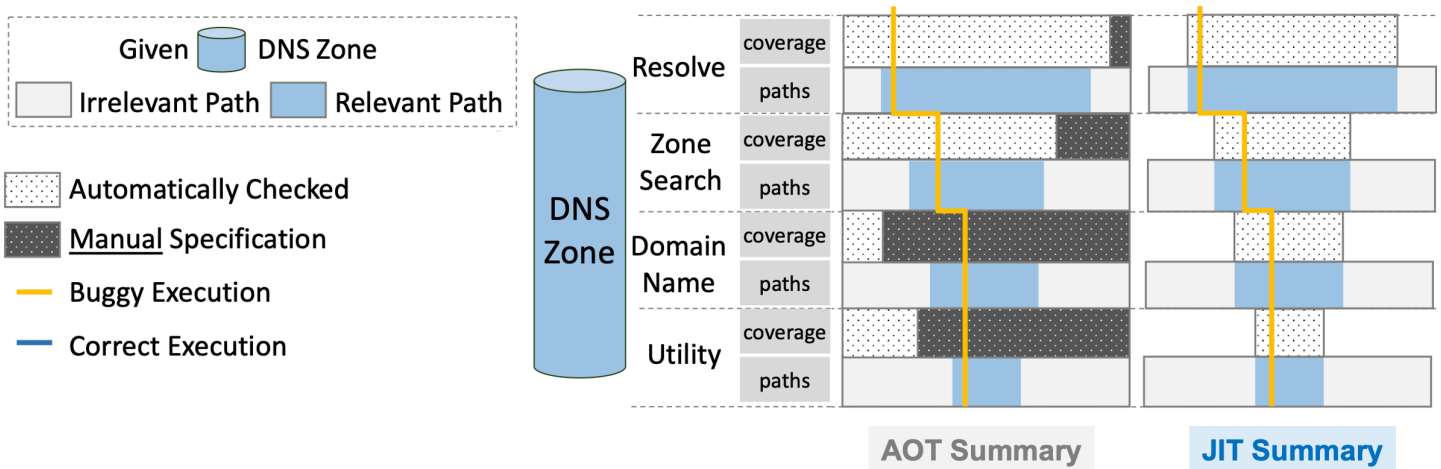
- **Summarizing in general:** loop count symbolic, memory accesses depend on offset array, many interleaving paths.

Example: domain name comparison (Bind 9)

```
20        p1    = &n1->ndata[*ofs1];  //| go to the
21        p2    = &n2->ndata[*ofs2];  //| next label
22        c1    = *p1++;      //| get label size
23        c2    = *p2++;      //| and move pointer
24        cdiff = (int)c1 - (int)c2;            //|
25        c1    = cdiff < 0 ? c1: c2;           //|
26        diff  = ascii_lowercmp(p1, p2, c);    //| compare
27        if (diff != 0 || cdiff != 0) goto done;
28        nlabels++;
29      }
30      if (ldiff < 0)      return namereln_contains;
31      else if (ldiff > 0) return namereln_subdomain;
32      else                return namereln_equal;
33    done:
```

- **Summarizing in general:** loop count symbolic, memory accesses depend on offset array, many interleaving paths.
- **With zone invariant:** (n1 = "example.com") loop bounded, memory accesses resolve to constants, paths collapse.

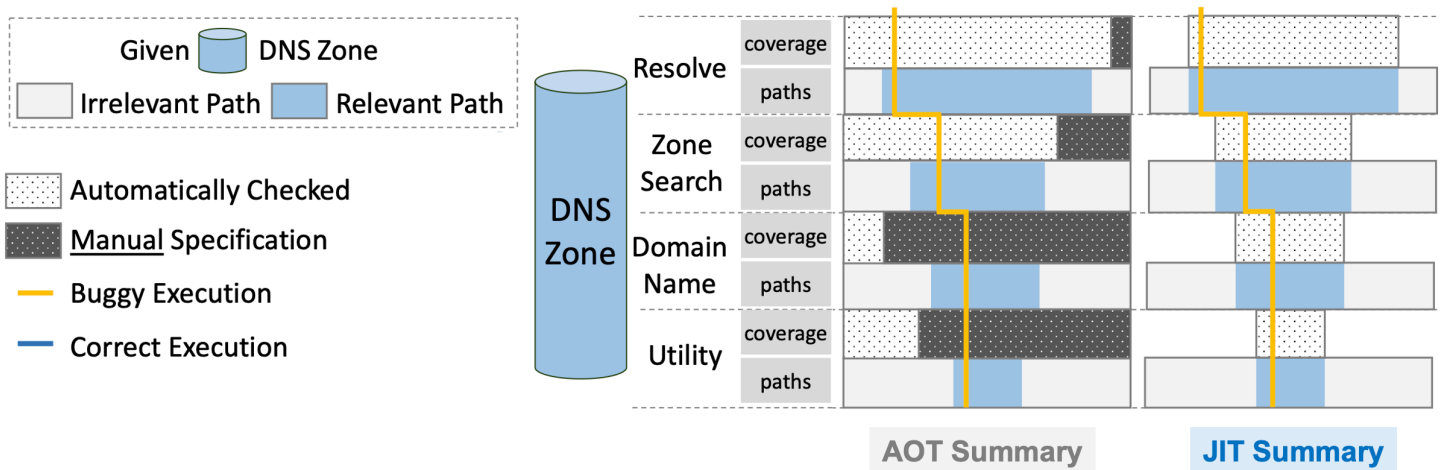
AOT vs. JIT summarization



Ahead-of-time (AOT) Summary

- Accounts for **all** inputs
- Many paths unreachable due to invariants
- Summaries unmanageable → manual specs

AOT vs. JIT summarization



Ahead-of-time (AOT) Summary

- Accounts for **all** inputs
- Many paths unreachable due to invariants
- Summaries unmanageable → manual specs

Just-in-time (JIT) Summary

- Summarizes **in context** of actual call
- Zone data prunes impossible paths
- Summaries stay small → **more automated**

JIT summaries: building top-down

- **Symbolic execution** on LLVM IR, top-down
- At each function call → build summary on the fly with zone invariants

JIT summaries: building top-down

- **Symbolic execution** on LLVM IR, top-down
- At each function call → build summary on the fly with zone invariants

```
5372  ns__query_start(query_ctx_t *qctx) {
          ...
          name_fullcomp(query, “example.com”)
```

```
1   namereln_t name_fullcomp(name_t *n1, name_t *n2) {
2     /* definitions omitted */
3     nlabels = 0;
4     ofs1    = n1->offsets;  //|
5     ofs2    = n2->offsets;  //| get offset pointers
6     l       = min(n1->labels, n2->labels);
7     ldiff   = n1->labels - n2->labels;
8     ofs1   += n1->labels;   //|
9     ofs2   += n2->labels;   //| start from the right
10    while (l-- > 0) {
11      ofs1--;                     //|
```

JIT summaries: building top-down

- **Symbolic execution** on LLVM IR, top-down
- At each function call → build summary on the fly with zone invariants

```
5372  ns__query_start(query_ctx_t *qctx) {
          ...
          name_fullcomp(query, “example.com”)
```

When the code dereferences a pointer into zone data

→ **state splitting**: substitute symbolic (AOT) with concrete value (JIT)

```
6     l       = min(n1→labels, n2→labels);
7     ldiff   = n1→labels - n2→labels;
8     ofs1   += n1→labels;   //|
9     ofs2   += n2→labels;   //| start from the right
10    while (l-- > 0) {
11      ofs1--;                    //|
12      ofs2--;                    //|
13      p1    = &n1→ndata[*ofs1];  //| go to the
14      p2    = &n2→ndata[*ofs2];  //| next label
15      c1    = *p1++;     //| get label size
```

JIT summaries: building top-down

- **Symbolic execution** on LLVM IR, top-down
- At each function call → build summary on the fly with zone invariants

```
5372  ns__query_start(query_ctx_t *qctx) {
          ...
          name_fullcomp(query, “example.com”)
```

When the code dereferences a pointer into zone data

→ **state splitting**: substitute symbolic (AOT) with concrete value (JIT)

```
14        p2    = &n2->ndata[*ofs2];  //| next label
15        c1    = *p1++;      //| get label size
16        c2    = *p2++;      //| and move pointer
17        cdiff = (int)c1 - (int)c2;            //|
18        c1    = cdiff < 0 ? c1: c2;           //|
19        diff  = ascii_lowercmp(p1, p2, c);    //| compare
20        if (diff != 0 || cdiff != 0) goto done;
21        nlabels++;
22      }
23      if (ldiff < 0)      return namereln_contains;
```

JIT summaries: building top-down

- **Symbolic execution** on LLVM IR, top-down
- At each function call → build summary on the fly with zone invariants

```
5372  ns__query_start(query_ctx_t *qctx) {
          ...
          name_fullcomp(query, “example.com”)
```

When the code dereferences a pointer into zone data

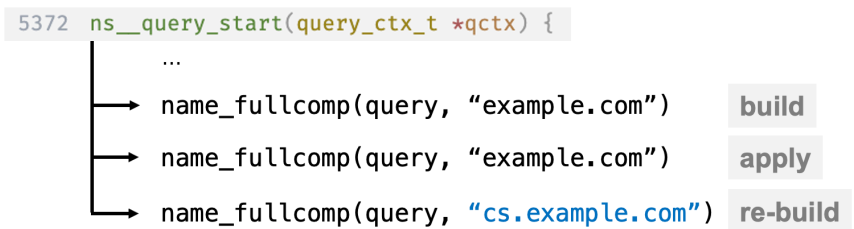
→ **state splitting**: substitute symbolic (AOT) with concrete value (JIT)

```
13        p1    = &n1->ndata[*ofs1];  //| go to the
14        p2    = &n2->ndata[*ofs2];  //| next label
15        c1    = *p1++;      //| get label size
16        c2    = *p2++;      //| and move pointer
17        cdiff = (int)c1 - (int)c2;            //|
18        c1    = cdiff < 0 ? c1: c2;           //|
19        diff  = ascii_lowercmp(p1, p2, c);    //| compare
20        if (diff != 0 || cdiff != 0) goto done;
21        nlabels++;
22      }
23      if (ldiff < 0)       return namereln_contains;
```

JIT state splitting: Replace symbolic values with concrete values, triggered by a set of rules (*see our paper*).

JIT summaries: applying and re-summarization

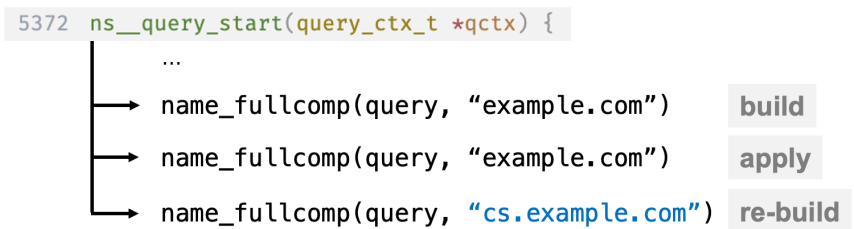
- **Symbolic execution** on LLVM IR, top-down




```
name_compare() # Summary tree (pseudo notation)
  └[Root]
    └[Specialize| n1 is "example.com" ]
      ├[Assume| query == "example.com" ]
      │ └[Ret| return EQUAL ]
      └[Assume| query ≠ "example.com" ]
        └[Ret| return PARTIAL / NONE ]
```

JIT summaries: applying and re-summarization

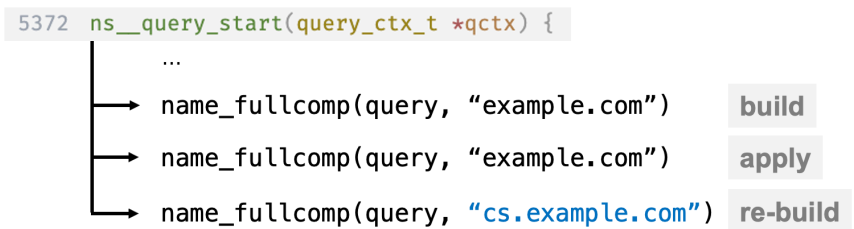
- **Symbolic execution** on LLVM IR, top-down
- First call with **"example.com"** → build summary




```
name_compare() # Summary tree (pseudo notation)
  └[Root]
    └[Specialize| n1 is "example.com" ]
      ├[Assume| query == "example.com" ]
      │ └[Ret| return EQUAL ]
      └[Assume| query ≠ "example.com" ]
        └[Ret| return PARTIAL / NONE ]
```

JIT summaries: applying and re-summarization

- **Symbolic execution** on LLVM IR, top-down
- First call with **"example.com"** → build summary
- Second call with **"example.com"** again → **apply** existing summary




```
name_compare() # Summary tree (pseudo notation)
  └[Root]
    └[Specialize| n1 is "example.com" ]
      ├[Assume| query == "example.com" ]
      │ └[Ret| return EQUAL ]
      └[Assume| query ≠ "example.com" ]
        └[Ret| return PARTIAL / NONE ]
```

JIT summaries: applying and re-summarization

- **Symbolic execution** on LLVM IR, top-down
- First call with **"example.com"** → build summary
- Second call with **"example.com"** again → **apply** existing summary
- Next call with **"cs.example.com"** → not yet covered → **re-summarize and merge**

```
5372 ns__query_start(query_ctx_t *qctx) {
        ...
        name_fullcomp(query, “example.com”)      build
        name_fullcomp(query, “example.com”)      apply
        name_fullcomp(query, “cs.example.com”)   re-build
```

```
name_compare() # Summary tree (pseudo notation)
  └[Root]
    ├[Specialize| n1 is "example.com" ]
    │ ├[Assume| query == "example.com" ]
    │ │ └[Ret| return EQUAL ]
    │ └[Assume| query ≠ "example.com" ]
    │   └[Ret| return PARTIAL / NONE ]
    └[Specialize| n1 is "cs.example.com" ]
      ├[Assume| query == "cs.example.com" ]
      │ └[Ret| return EQUAL ]
      └[Assume| query ≠ "cs.example.com" ]
        └[Ret| return PARTIAL / NONE ]
```

JIT summaries: applying and re-summarization

- **Symbolic execution** on LLVM IR, top-down
- First call with **"example.com"** → build summary
- Second call with **"example.com"** again → **apply** existing summary
- Next call with **"cs.example.com"** → not yet covered → **re-summarize and merge**

```
5372 ns__query_start(query_ctx_t *qctx) {
        ...
        name_fullcomp(query, “example.com”)        build
        name_fullcomp(query, “example.com”)        apply
        name_fullcomp(query, “cs.example.com”)     re-build
```

- Summary grows just in time to cover necessary behaviors

```
name_compare() # Summary tree (pseudo notation)
  └[Root]
    ├[Specialize| n1 is "example.com" ]
    │ ├[Assume| query == "example.com" ]
    │ │ └[Ret| return EQUAL ]
    │ └[Assume| query ≠ "example.com" ]
    │   └[Ret| return PARTIAL / NONE ]
    └[Specialize| n1 is "cs.example.com" ]
      ├[Assume| query == "cs.example.com" ]
      │ └[Ret| return EQUAL ]
      └[Assume| query ≠ "cs.example.com" ]
        └[Ret| return PARTIAL / NONE ]
```

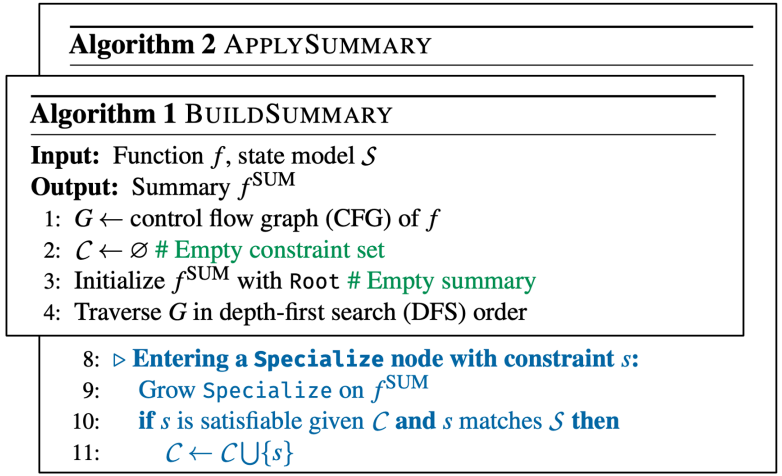
Formal model: check out our paper!

JIT summaries are **trees of constraints and effects**,
managed by two core algorithms:

Formal model: check out our paper!

JIT summaries are **trees of constraints and effects**, managed by two core algorithms:

- **BuildSummary** — top-down symbolic execution with JIT state splitting
- **ApplySummary** — match existing summary or trigger re-summarization



Algorithm 2 ApplySummary

Algorithm 1 BuildSummary

Input: Function f, state model $\mathcal{S}$
Output: Summary f^{SUM}

1: $G \leftarrow$ control flow graph (CFG) of f
2: $\mathcal{C} \leftarrow \varnothing$ # Empty constraint set
3: Initialize f^{SUM} with `Root` # Empty summary
4: Traverse G in depth-first search (DFS) order

8: ▷ **Entering a `Specialize` node with constraint s:**
9: Grow `Specialize` on f^{SUM}
10: **if** s is satisfiable given $\mathcal{C}$ **and** s matches $\mathcal{S}$ **then**
11: $\mathcal{C} \leftarrow \mathcal{C} \bigcup \{s\}$

Formal model: check out our paper!

JIT summaries are **trees of constraints and effects**, managed by two core algorithms:

- **BuildSummary** — top-down symbolic execution with JIT state splitting
- **ApplySummary** — match existing summary or trigger re-summarization
- Operates on a **unified state/memory model** across all proof layers.

State

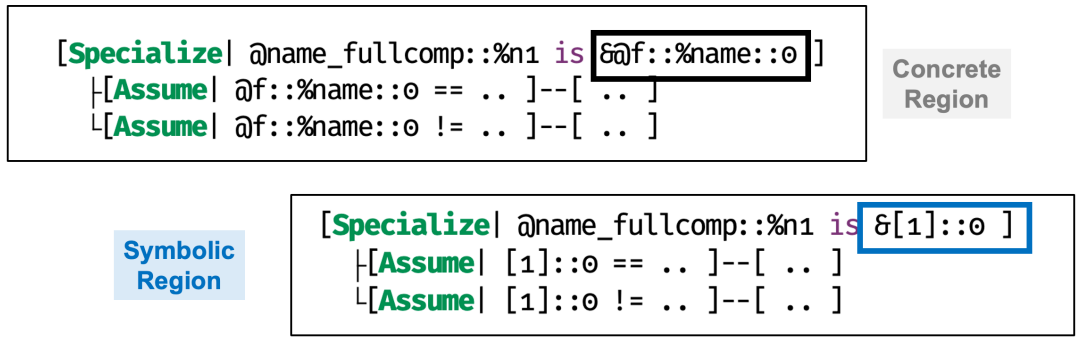
Effect	Definition	Example
Root	Root placeholder	[Root]
Assume	Path constraint	[Assume| x > 0]
Specialize	JIT state splitting	[Specialize| x is 1]
Panic	Runtime error	[Panic| "..."]
Return	Binding a return value	[Return| 2w64]
Store	Memory store	null -> Region::0
Malloc	Heap allocation	[0] := malloc ptr
Free	Heap deallocation	free [0]

ValueState

MemoryState ::= Region :: *int* — Region with offsets
Region ::= *@func_ident* :: *%ident* — Stack region
| Global :: *%ident* — Global region
| Const :: *%ident* — Constant region
| Heap :: *%ident* — Heap region

Further scale verification

Further scale verification



Symbolic regions — reuse summaries across concrete memory addresses.

Further scale verification

```
[Specialize| @name_fullcomp::%n1 is &[1]::0 ]
  ├[Assume| [1]::0 == .. ]--[ .. ]
  └[Assume| [1]::0 != .. ]--[ .. ]
```

Summary Optimization Passes

- `MinimizeAssume`
- `MergeAssume`
- `...`

Symbolic regions — reuse summaries across concrete memory addresses.

Summary optimization — compiler-like passes that optimize summaries.

Further scale verification

```
1 //unique_ptr<DNSPacket> PacketHandler::doQuestion(DNSPacket& p)
2 fn stub(ret: Ptr, h: Ptr, p: Ptr) -> Bool {
3   let qtype = p |> DnsPacket.qtype |> *Int<16>; // read qtype
4   let _ = debug(qtype);                        // log qtype
5   specialize(h |> PacketHandler.log_dns_details |> *Int<8>)
6   && qtype != 0w16 && qtype < 249w16 // qtype not EMPTY | META
7 }
```

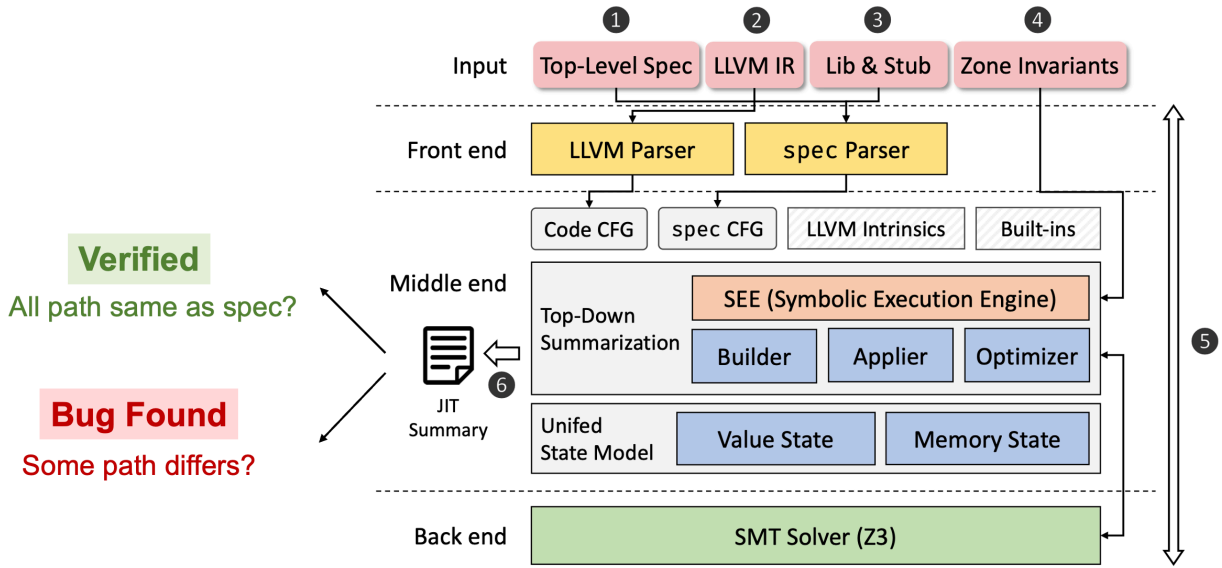
Stub Function
(PowerDNS)

Symbolic regions — reuse summaries across concrete memory addresses.

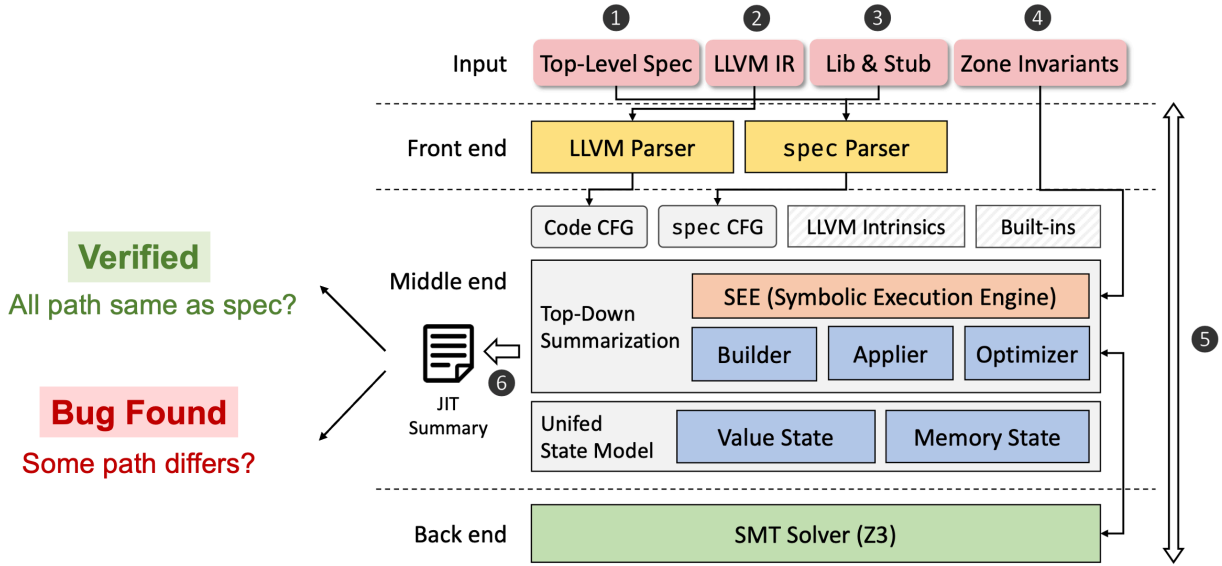
Summary optimization — compiler-like passes that optimize summaries.

Stub functions — lightweight interposition to customize verification.

Iceberg overview

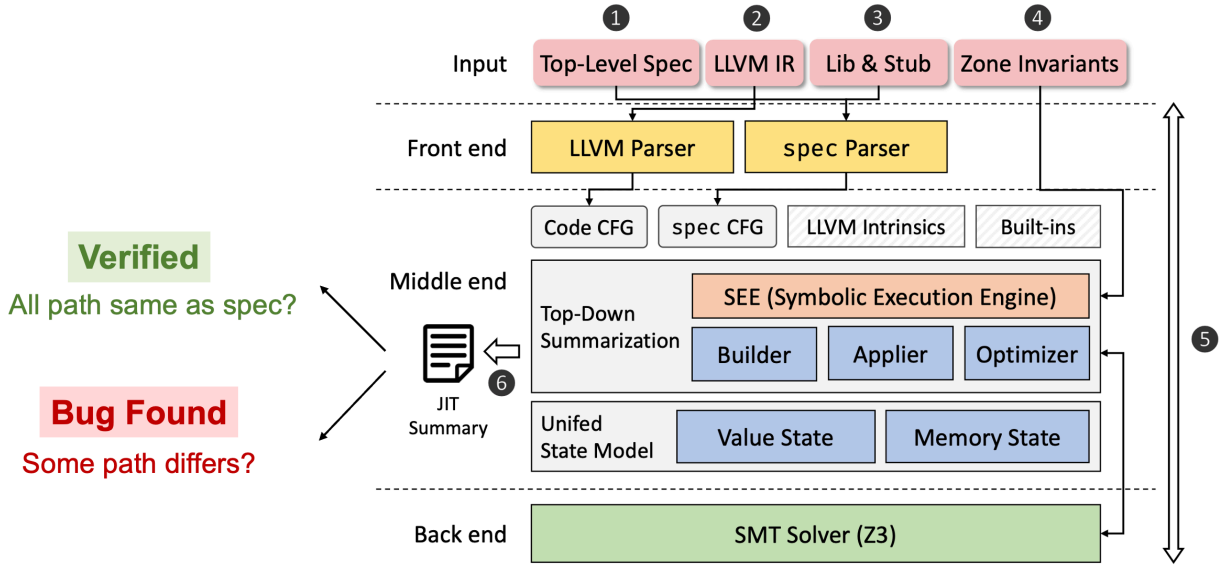


Iceberg overview



Push-button verification — no per-layer specifications needed.

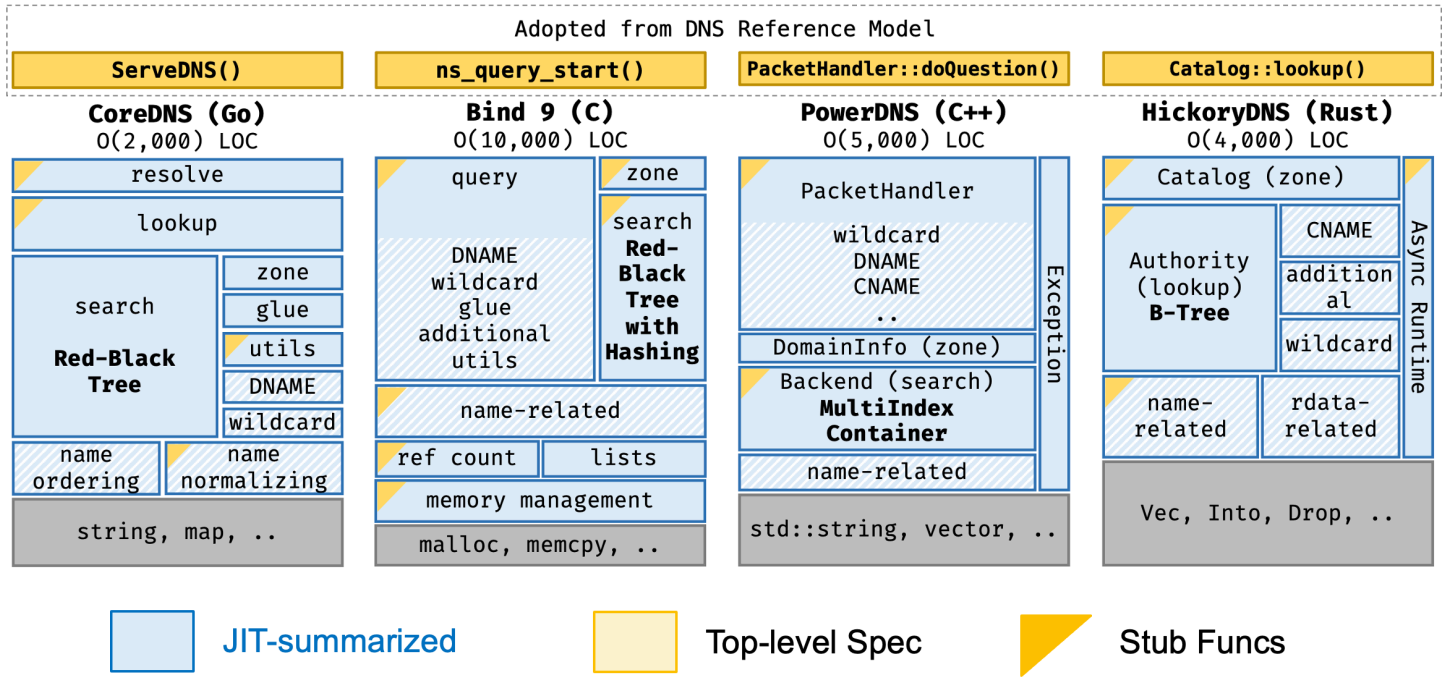
Iceberg overview



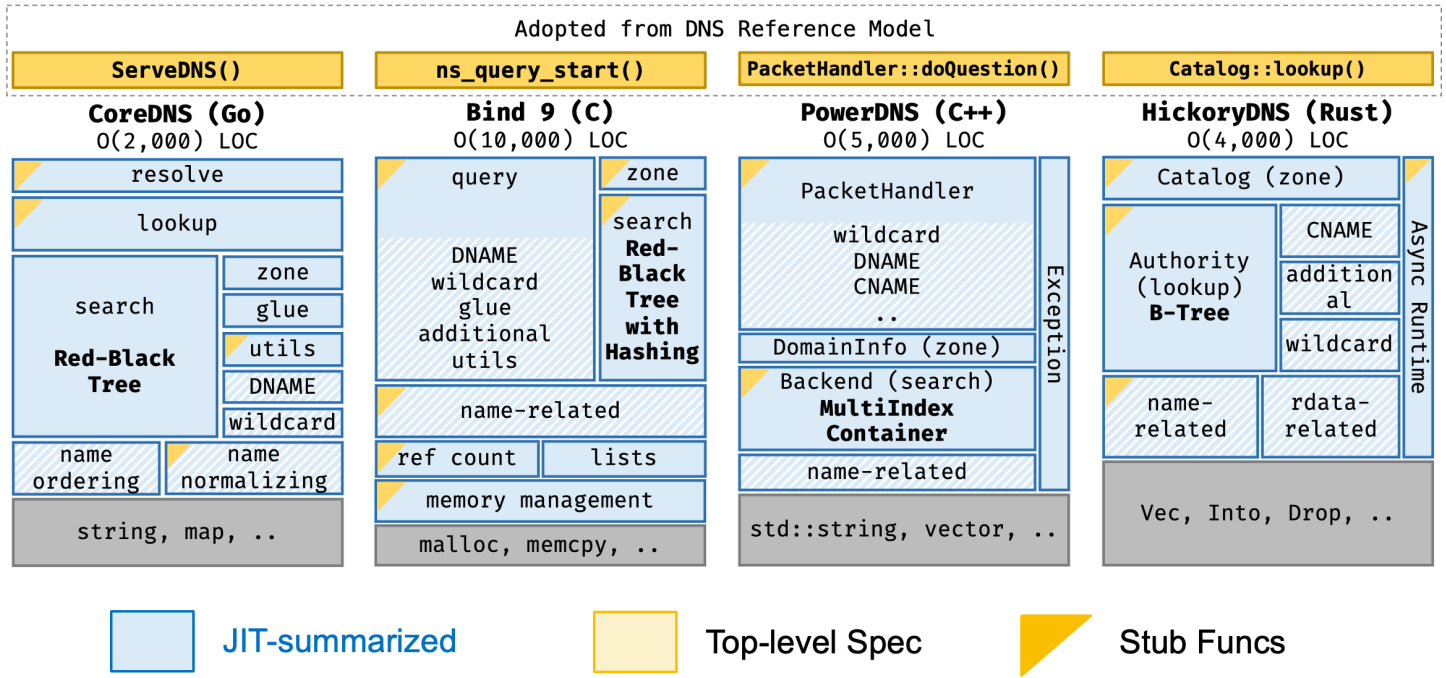
Push-button verification — no per-layer specifications needed.

Implementation: 16,000 Rust LOC, based on the Z3 SMT solver

Practical verification

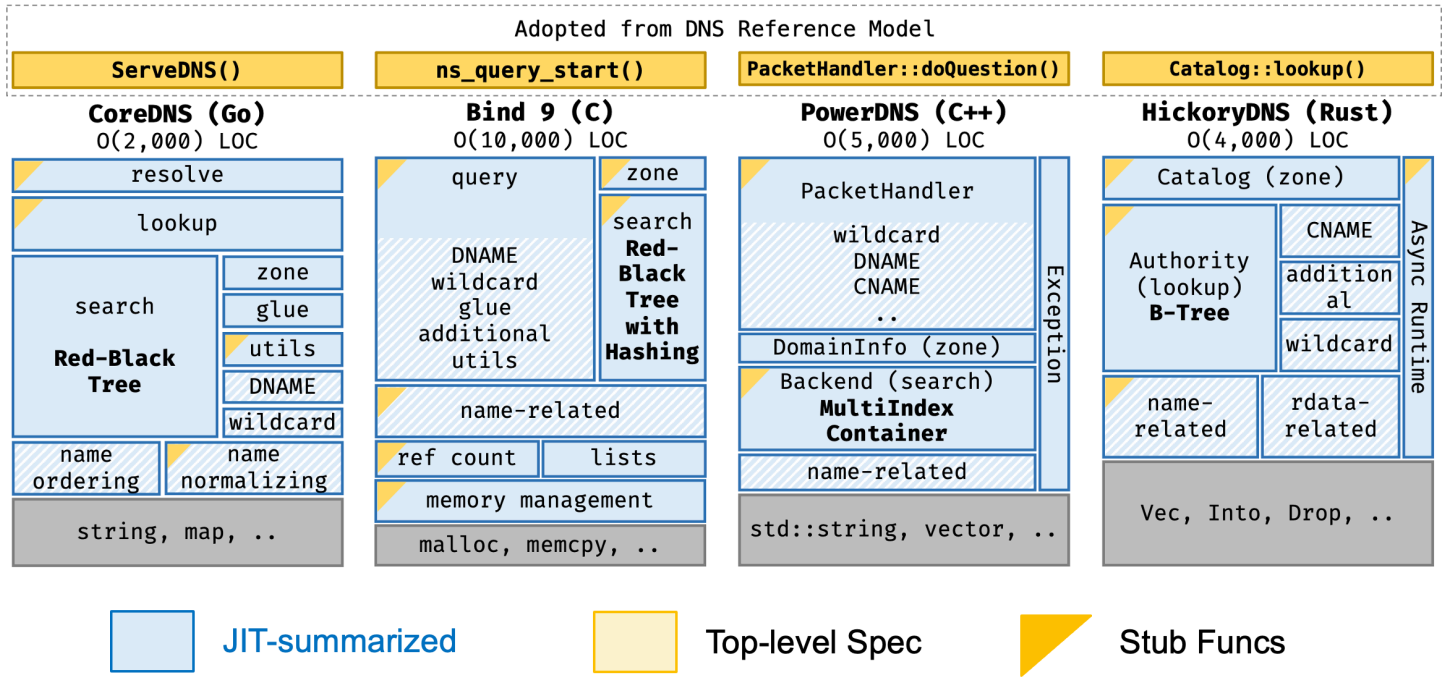


Practical verification



Scope: core lookup logic (query → zone → response). Excludes I/O, networking, concurrency.

Practical verification

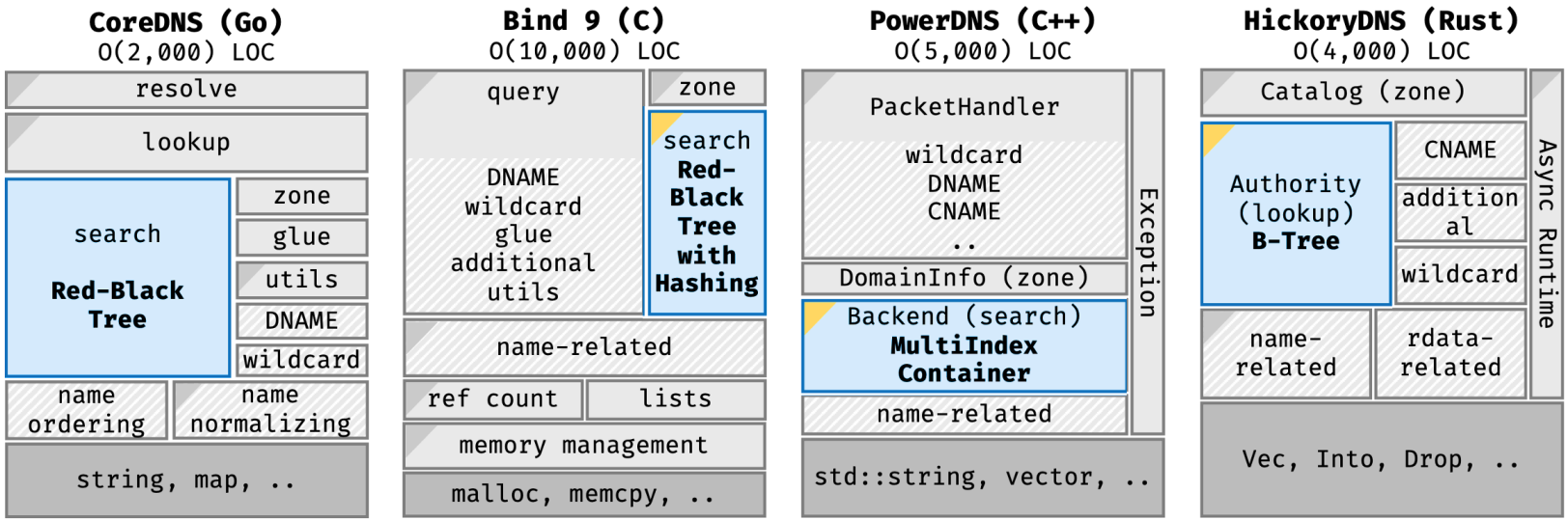


Scope: core lookup logic (query → zone → response). Excludes I/O, networking, concurrency.

Top-level spec: DNS reference model derived from 7+ RFCs, tailored per engine.

Verification experience

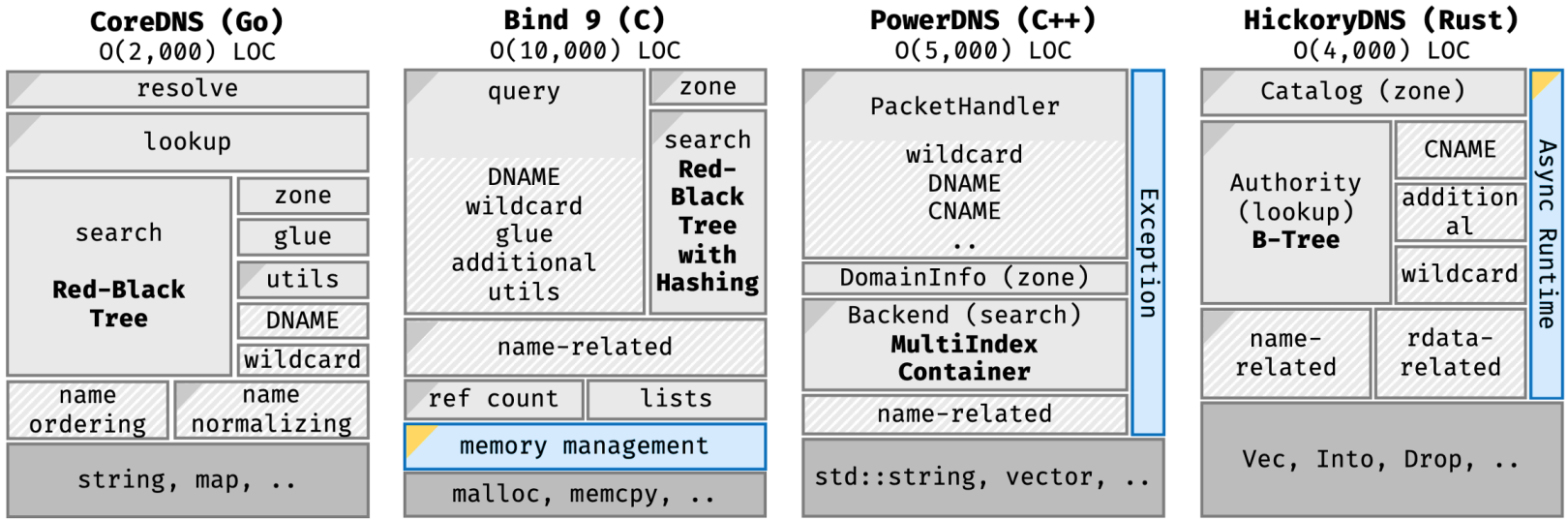
Verification experience



Diverse data structures — red-black trees, B-trees, hash tables, multi-index containers.

JIT doesn't model the data structure — it state-splits on the *concrete zone layout*. The same mechanism handles all four, with no per-data-structure specs.

Verification experience



Language-specific IR patterns — C++ exceptions, Rust async state machines.

Complex compilation artifacts in LLVM IR. JIT explores only *reachable* states top-down — no manual modeling.

Evaluation setup

Zones: 3 categories for thorough coverage

Evaluation setup

Zones: 3 categories for thorough coverage

Category	Size	Count	Source
simple	2–4 records	1,000	SCALE
complex	10–12 records	100	randomly generated
real	137 & 393 records	2	public CZDS data

Evaluation setup

Zones: 3 categories for thorough coverage

Category	Size	Count	Source
simple	2–4 records	1,000	SCALE
complex	10–12 records	100	randomly generated
real	137 & 393 records	2	public CZDS data

Baseline: SCALE (NSDI'22) — state-of-the-art symbolic test generator for DNS engines

Evaluation setup

Zones: 3 categories for thorough coverage

Category	Size	Count	Source
simple	2–4 records	1,000	SCALE
complex	10–12 records	100	randomly generated
real	137 & 393 records	2	public CZDS data

Baseline: SCALE (NSDI'22) — state-of-the-art symbolic test generator for DNS engines

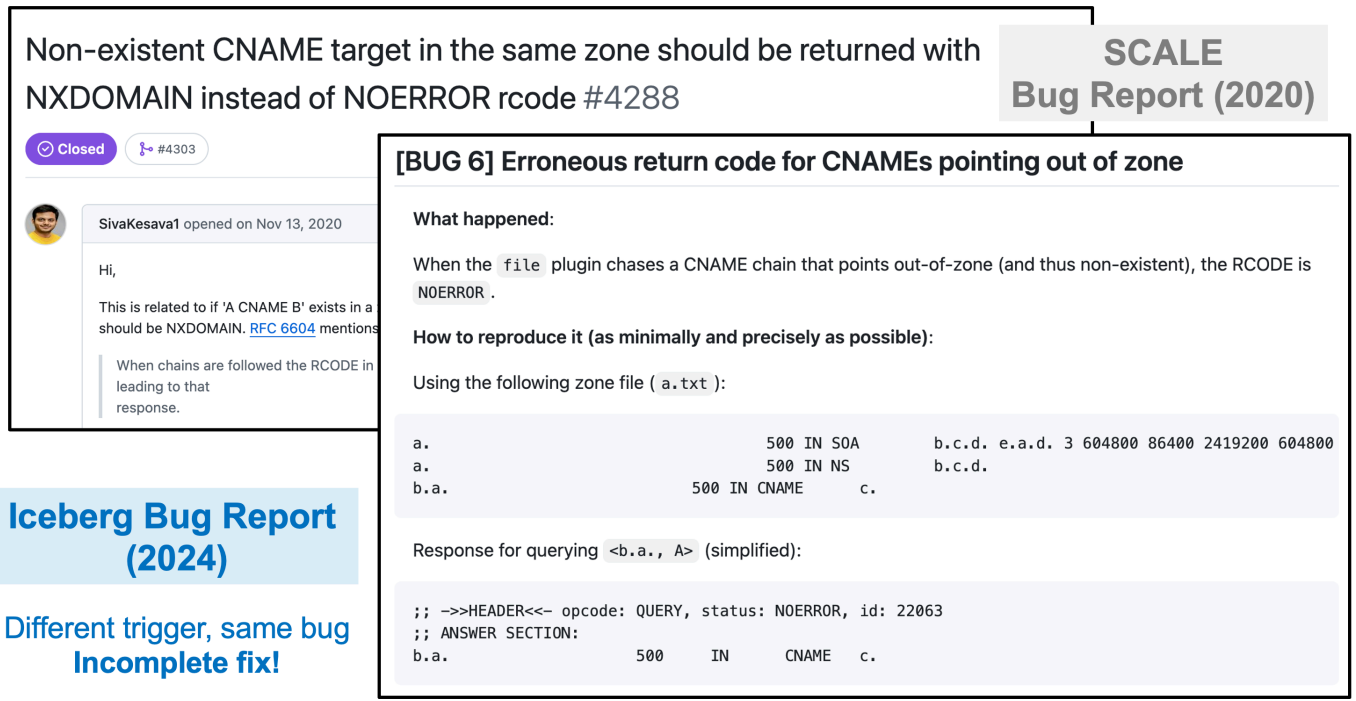
Metrics: bugs found / manual effort / verification time

Bugs Found

Index	Impl	Category	Description	Status	SCALE
1	CoreDNS	Wrong Answer & Flag	Incorrect response when `DNAME` redirects to a zonecut.	☑	✗
2	CoreDNS	Wrong Answer	Missing `SOA`/`NS` records when `CNAME` redirects to a zone apex.†	☑	✗
3	CoreDNS	Wrong Additional	Missing records for wildcard `MX`s.	✓	✗
4	CoreDNS	Wrong Additional	Extraneous glue records for `MX` queries.	✓	✗
5	CoreDNS	Wrong Additional	Duplicate records during the additional section handling.	✓	✗
6	CoreDNS	Wrong `rcode`	Erroneous return code for `CNAME`s pointing out of zone.†	☑	✗
7	CoreDNS	Wrong Additional	Extraneous glue records for non-referral `NS` queries.	✓	✓
8	CoreDNS	Wrong Additional	Missing additional records after `DNAME` substitution.	☑	✗
9	PowerDNS	Wrong Answer	Extraneous records when matching wildcard `CNAME`s.	✓	✓
10	PowerDNS	Wrong Authority	Extraneous `SOA` records after `DNAME` redirection.	☑	✗
11	HickoryDNS	Wrong Answer & Flag	Incorrect response for a non-existent `SOA` query.	☑	✗
12	HickoryDNS	Wrong Additional	Wrong record placement for `CNAME`-related queries.	☑	✗

- **SCALE** found only **2 of 12** on the same zones.

Bugs Found



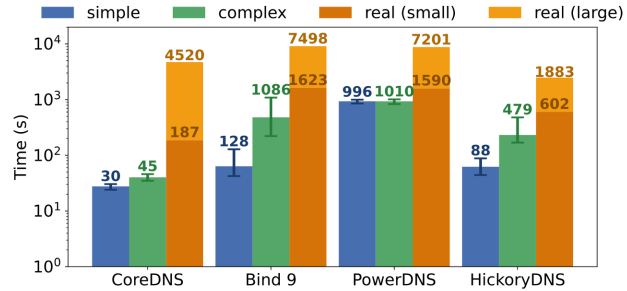
- Some bugs were in *incomplete fixes* for issues SCALE had previously reported.

Manual effort & overhead

	top	lib	stub	spec-to-code ratio
CoreDNS	790	233	192	9.6%
Bind 9	860	108	752	7.5%
PowerDNS	752	849	290	5.8%
HickoryDNS	731	249	314	7.8%

Manual Effort

spec-to-code ratio measured w/ top specs



Verification Overhead

Manual effort

Spec-to-code ratio: **5.8%–9.6%**

Upfront porting: **~1 person-week**

Version retrofitting: **~1 person-hour**

Verification time

Simple zones: **minutes**

Complex zones: **~30 min**

Production zones: **few hours**

All verification runs terminate on a standard cloud server (Alibaba Cloud ecs.c9i.4xlarge).

Beyond DNS

Many systems operate on **pre-configured dataplane**:

Beyond DNS

Many systems operate on **pre-configured dataplane**:

- Network switches → forwarding tables
- SDN controllers → flow rules
- Database query engines → schemas and indexes

Beyond DNS

Many systems operate on **pre-configured dataplane**:

- Network switches → forwarding tables
- SDN controllers → flow rules
- Database query engines → schemas and indexes

> Whenever behavior is shaped by fixed configuration, JIT summarization can exploit those invariants.

Beyond DNS

Many systems operate on **pre-configured dataplane**:

- Network switches → forwarding tables
- SDN controllers → flow rules
- Database query engines → schemas and indexes

> Whenever behavior is shaped by fixed configuration, JIT summarization can exploit those invariants.

Summary

- **Iceberg** uses JIT summarization to automatically verify DNS engines.
- **4 engines** verified, **12 bugs** found, **1 person-week** porting cost.

Summary

- **Iceberg** uses JIT summarization to automatically verify DNS engines.
- **4 engines** verified, **12 bugs** found, **1 person-week** porting cost.

Thanks for listening!