USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

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

Yuxing Xiang, Rilin Huang, Naiqian Zheng, and Xin Jin, *Peking University*

https://www.usenix.org/conference/nsdi26/presentation/xiang-iceberg

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

ARTIFACT EVALUATED USENIX AVAILABLE
ARTIFACT EVALUATED USENIX FUNCTIONAL
ARTIFACT EVALUATED USENIX REPRODUCED

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

Yuxing Xiang Rilin Huang Naiqian Zheng Xin Jin
School of Computer Science, Peking University

Abstract. As the core of DNS services, DNS authoritative engines are responsible for answering DNS queries with DNS responses, where any bugs may lead to severe consequences. While it is critical to ensure the correctness of the engine, existing solutions fail to deliver both correctness guarantees and low manual effort when applied to large and complex implementations in use. The state-of-the-art solution, DNS-V, relies on extra manual specifications for scaling verification, which still incurs a prohibitive cost.

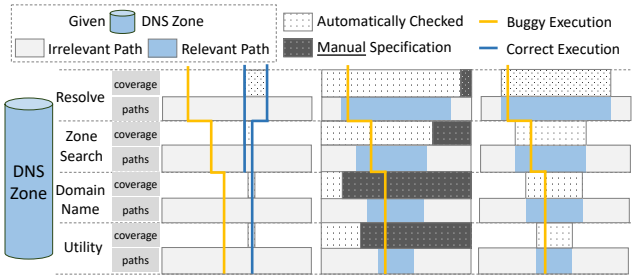
In this paper, we present Iceberg, an automated verification framework for DNS authoritative engines that holistically reduces manual effort. To achieve this, we propose *just-in-time (JIT) summarization*, a refinement-proof approach that utilizes invariants from DNS zones to enable the use of automated summaries throughout verification, especially for domain-specific DNS operations. In addition, we employ a set of techniques to further scale automation, including symbolic regions, summary optimization, and stub function interposing. We apply Iceberg to four open-source DNS engines, identifying 12 new bugs while keeping manual effort low.

1 Introduction

The Domain Name System (DNS) plays a crucial role in today's Internet. It provides name resolution, enabling the translation from human-readable domain names into IP addresses, and thus grants access to online services. At the center of DNS are authoritative engines, which are responsible for matching incoming DNS queries with locally configured authoritative DNS zone records and generating corresponding DNS responses. For the cloud providers and network operators that maintain their own authoritative engine implementations, it is of vital importance to ensure their correctness, *i.e.,* the engine composes the correct response for any query given the configured zone. In fact, past bugs in authoritative engines have caused major network outages and security risks, leading to significant business losses [15, 18–20].

Nevertheless, it is an enormous undertaking in practice to ensure correctness for DNS engines, which requires comprehensive analysis at the implementation level. Any form of such analysis is impeded by the complexity arising from handling numerous types of DNS records, the customizations and optimizations prevalent in applications, and the sheer scale of the implementations (*e.g.,* $O(10{,}000)$ lines of code in Bind 9 [10] are related to name resolution).

Existing work on DNS reliability fails to achieve both correctness guarantees and low manual effort for DNS engines.



(a) SCALE (b) DNS-V (AOT) (c) Iceberg (JIT)

Figure 1. Three testing/verification approaches. The hierarchy represents approximate module layers but not the actual software structure, and the size of the horizontal bars represents relative quantities. The testing method (a) has inferior coverage. The ahead-of-time summarization method (b) ensures full coverage but induces considerable manual effort. Our just-in-time summarization method (c) achieves full automation by reducing unnecessary coverage.

As shown in Figure 1(a), the testing approach represented by SCALE [48] cannot ensure full coverage and might miss bugs. Meanwhile, automated verification [57,58,60,63,70] has been successful in formally verifying complex software by adopting refinement proofs [56], which break the target software into manageable *layers* and derive the proof in steps by *lifting* layers to respective specifications. However, this approach is often labor-intensive due to the need for per-layer specifications (and thus effort) to scale the proof. The state-of-the-art solution, DNS-V [72], partially automates the refinement verification of one closed-source engine by replacing the manual specifications (dark shaded bars in Figure 1(b)) in some layers with summarization (light shaded bars). Still, the need for per-layer manual specifications persists, and properly designing, developing, and executing them not only requires expertise but is also time-consuming (§3.2).

This limited use of summarization is attributable to it being conducted **ahead-of-time** (AOT), where the summary assumes nothing and must account for all code paths in a layer for every combination of its dependent states. Nevertheless, many such paths and state combinations are in fact impossible to reach in actual execution because of implicit *invariants* across layers. For example, any code that works with a C++ `vector` should be able to assume its length is not greater than its capacity (type invariant), or the correctness of modules might rely on some global states being constant (state invariant), where assuming otherwise leads to a great number of unreachable "error" paths and inflates summarization. In fact, AOT summarization is particularly inefficient for low-level modules where invariants are common, making

it impractical to summarize their implementation by simply enumerating paths ahead of time. As a result, DNS-V often falls back to relying on manual insights when carrying out the refinement proof, leading to prohibitive costs in application.

Technique. In this work, we propose Iceberg, an automated verification framework for DNS authoritative engines that holistically reduces manual effort during refinement through a novel **just-in-time** (JIT) summarization technique. As shown in Figure 1(c), the key idea behind JIT summarization is to encode only the logic relevant to the actual execution, rather than blindly collecting all code paths ahead of time.

JIT summarization is particularly suitable for verifying the correctness of DNS authoritative engines based on two observations (§3.3). First, DNS authoritative engines operate with pre-configured DNS zones, which provide rich state and type *invariants* in the form of fixed arguments, constant memory, and well-formed data structures. Second, summarization of *domain-specific DNS operations* benefits from these invariants with pruned execution paths and simplified summary representations. For example, domain name comparison using the RFC-advocated domain name representation [1] can be encoded much more easily if one of the compared names is concrete data. As a result, JIT summarization makes the summaries more efficient, empowering the use of summarization for *all* layers, and thus achieving low-effort refinement proof.

To build JIT summaries (§5), Iceberg performs top-down symbolic execution [49] on software modules (as LLVM IR), aggregating path constraints and effects into summaries. Concrete and symbolic values propagate through a unified state model to form expressions. During summarization, guided by a set of rules, Iceberg splits symbolic states [40, 46] into their concrete values, capturing them just in time as invariants. This design allows Iceberg to generate summaries just in time for only the relevant paths at each proof layer, skipping unnecessary paths while simplifying the rest.

Several other techniques complement JIT summarization to further scale verification (§6). First, Iceberg supports symbolic memory regions (§6.1) to efficiently summarize memory accesses. Second, it implements summary optimization passes (§6.2) to transform summaries into more solver-friendly forms. Third, Iceberg accepts stubs (§6.3) for interposing and tuning overall verification.

Key results. Iceberg is implemented with over 16,000 lines of Rust code. To show that Iceberg reduces manual effort, we apply it to verify four open-source DNS engines (§7): CoreDNS [24], Bind 9 [10], PowerDNS [26], and HickoryDNS [25], despite their major differences in terms of programming languages (Golang, C, C++, and Rust), key data structures (red-black tree, hashmap, multi-index container, and B-tree), and internal design. Iceberg has uncovered 12 new bugs across these implementations, 7 of which are acknowledged by developers (§8). These bugs include critical violations of DNS standards, while keeping the porting cost to one person-week and the spec-to-code ratio under 10%. We also discuss how Iceberg can be generalized to verify software in other domains in §9.

2 Background

2.1 DNS Terminologies

A *domain name* is a list of string labels separated by dots. A DNS *zone* consists of multiple domain names organized in a tree hierarchy, each associated with a set of *resource record*s (RR). The main components of an RR are the record name (`rname`), type (`rtype`), and data (`rdata`). For example, `<example.com, A, 8.8.8.8>` is an IPv4 record under the domain `example.com` with the IP address `8.8.8.8`. A DNS *query* contains a target domain name (`qname`) and a type (`qtype`). Upon receiving a query, the authoritative engine selects relevant zones and looks up RRs matching both `qname` and `qtype`. The *response* includes important fields such as the response code (`rcode`), authoritative answer flag (`aa`), and three sections of RRs: `answer`, `authority`, and `additional`.

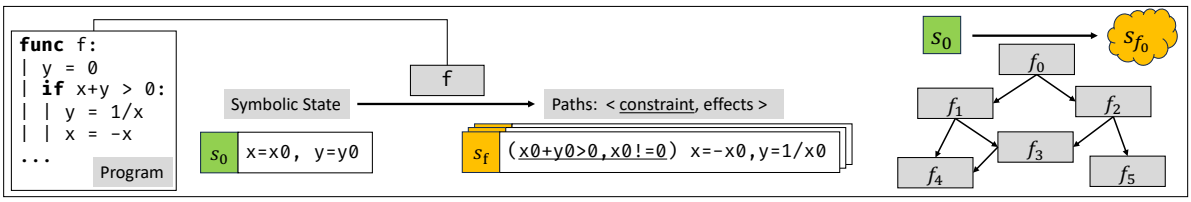
A collection of RFCs [1–8, 23] defines the precise rules for specifying correct DNS responses, with certain `qtype`, `rname`, and `rtype` combinations triggering special handling.

2.2 Automated Testing & Verification

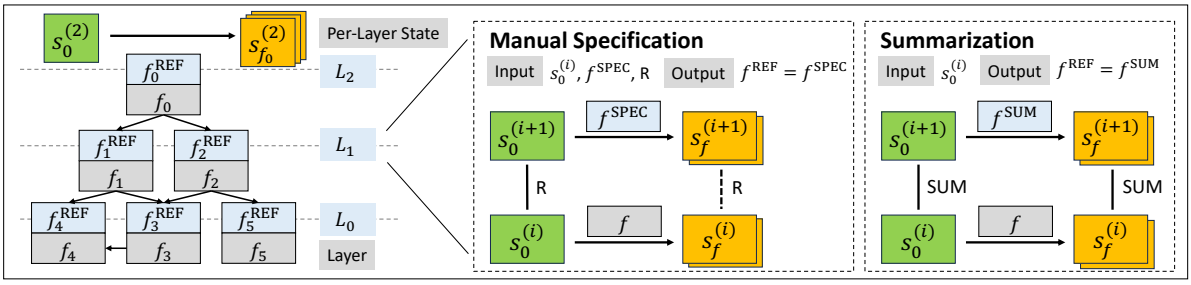
Symbolic execution. Symbolic execution [49] is a well-established approach for automating the reasoning about program behaviors. As shown in Figure 2(a), it systematically explores all viable execution paths by executing with symbolic states, capturing path constraints and side effects as symbolic expressions after execution. Reasoning about the program can then be achieved by checking the resulting paths.

This technique serves as a building block for software testing and verification. For testing, symbolic execution over an abstract model of the program helps produce higher-quality test cases [48]. For verification, verifiers can ensure correctness by proving that the desired properties hold in every possible state after symbolic execution.

Refinement proof. Symbolic execution alone scales poorly for large programs, due to the exponential growth in execution paths (*i.e.,* path explosion [29]). Verifying large systems is typically accomplished using refinement proofs [56]. As shown in Figure 2(b), the full program is first organized into *layers*. A code module f in a lower layer is then *lifted* to a refined representation f^{REF} via either manual specification or automatic summarization. For the manual specification, users are required to devise the target specification f^{SPEC} as well as the simulation relation R that lifts symbolic states to the next layer, and the verifier helps prove or deny this refinement proof by checking whether R holds after execution. For summarization, symbolic execution handles the lifting of symbolic states (via SUM), and the path collection produced by symbolic execution f^{SUM} (known as a *summary*) is taken as (by definition) the refinement of f directly.



(a) Symbolic execution.



(b) Refinement proof.

Figure 2. Illustration of symbolic execution and refinement proof. Naive symbolic execution (a) for full-program analysis leads to path explosion. With refinement proof (b), program f is lifted to either manual specifications (f^{SPEC}) or automatic summaries (f^{SUM}).

While manual specifications are typically more concise than summaries due to the manual expertise involved, the latter can be fully automated. Refinement proof has been successfully applied to verify many large systems [39, 43–45, 52–54, 57, 60, 65, 65, 71, 72], some of which suggest the use of summarization to replace manual specifications.

3 Motivation

We seek a method that both ensures the correctness of DNS authoritative engines and requires low manual effort. This section explains our goals, demonstrates the challenges faced by existing solutions, and highlights the opportunities.

3.1 Our Goals

Goal 1: Correctness. Our first goal is to ensure *correctness* [72] of DNS authoritative engines: *i.e.,* the engine composes correct responses for any queries given configured zones. To be exact, we aim for two properties:

- Safety: the DNS authoritative engine should not produce any runtime errors when handling any query.
- Functional correctness: the composed response should be correct for any query, up to a top-level specification that complies with both standard RFCs and application-specific customizations.

Goal 2: Low manual effort. Existing implementations [10, 24–26] are written in various programming languages (Go, C, C++, Rust, *etc.*), use complex control flows and data structures, and can involve up to 10,000 lines of code (LOC). Furthermore, these implementations are subject to frequent iterations due to changing needs. To make verification practically feasible for such systems, the manual effort must be kept low: the method should accommodate large and diverse implementations with minimal upfront porting work, and handle evolving versions with little retrofitting overhead.

Table 1. Estimated manual effort across implementations, measured as the percentage of low-level functions over all verified functions.

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

3.2 Challenges with DNS-V

Given our first goal of correctness guarantee, we focus our discussion on the verification approach, as testing [16, 22, 48] can only identify but not eliminate the existence of bugs. In pursuit of the second goal, the DNS-V [72] methodology emerges as the most promising, best described as refinement with **ahead-of-time** (AOT) summarization. Essentially, the user first designs the refinement layers, dividing code modules, devising per-layer abstract domains, and deciding which layers are to be handled manually. Then, manual specifications must be developed for those layers, providing sufficient abstraction for further refinement. Finally, the refinement proof is executed bottom-up with symbolic execution, using automated summaries as refined representations in the summarized layers. Notably, an important decision is made for each layer by the user prior to its proof—either a manual specification is provided to help abstract the layer, or the verifier assumes nothing and generates a full summary.

While DNS-V partially automates the refinement proof in summarized layers, we find that its ahead-of-time approach remains time-consuming and expertise-demanding in practical application, especially in the following three aspects.

Designing layers. The mapping from code modules to refinement layers, as well as the assignment of manual layers and their abstract domains, requires expertise in formal methods. Consider the real-world example in Figure 3, which compares two domain names in the RFC-standard format [1] (`name_t`). For non-experts, it is unclear how the exposed complexity in this function will affect the efficiency of automated AOT

```
1  typedef struct {
2    unsigned char      *ndata;    // binary data
3    unsigned int        labels;   // number of labels
4    unsigned char      *offsets;  // offset of labels
5  } name_t; /* other fields omitted */
6
7  namereln_t name_fullcomp(name_t *n1, name_t *n2) {
8    /* definitions omitted */
9    nlabels = 0;
10   ofs1    = n1->offsets;  //|
11   ofs2    = n2->offsets;  //| get offset pointers
12   l       = min(n1->labels, n2->labels);
13   ldiff   = n1->labels - n2->labels;
14   ofs1   += n1->labels;   //|
15   ofs2   += n2->labels;   //| start from the right
16   while (l-- > 0) {
17     ofs1--;                    //|
18     ofs2--;                    //|
19     p1    = &n1->ndata[*ofs1]; //| go to the
20     p2    = &n2->ndata[*ofs2]; //| next label
21     c1    = *p1++;     //| get label size
22     c2    = *p2++;     //| and move pointer
23     cdiff = (int)c1 - (int)c2;          //|
24     c1    = cdiff < 0 ? c1: c2;         //|
25     diff  = ascii_lowercmp(p1, p2, c);  //| compare
26     if (diff != 0 || cdiff != 0) goto done;
27     nlabels++;
28   }
29   if (ldiff < 0)      return namereln_contains;
30   else if (ldiff > 0) return namereln_subdomain;
31   else                return namereln_equal;
32 done:
33   if (nlabels > 0)    return namereln_commonancestor;
34   return namereln_none;
35 }
```

Figure 3. The (simplified) domain name comparison function in Bind 9. `offsets` indicates label positions in `ndata` (*e.g.,* for `a.com.`, `ndata` is `[1,a,3,c,o,m,0]` and `offsets` is `[0,2,6]`).

summaries. Further, should manual effort be put into a specification for this module, it is also not straightforward to design its abstract domain. Mapping a `name_t` to a list of strings can be efficient for future refinement, but necessitates extra formulation on the well-formedness of data structures to successfully lift the low-level implementation. Meanwhile, using a less abstract mapping may result in easier lifting but hinder reasoning in upper layers.

Developing for manual layers. The heavy lifting of AOT summarization is also rooted in the potential scale of manual layers. Table 1 roughly estimates the percentage of functions that likely require manual handling in DNS-V across different DNS engine implementations. For Bind 9 (implemented in C), its verbose and performance-driven nature renders the portion of manual work as high as 64%, underscoring a potentially significant undertaking for specification development.

Executing layer proofs. During the bottom-up execution of AOT summarization, lifting can fail due to incorrect implementation, insufficient specification, or path explosion in summarization. For the last two cases (paramount in practice), intricate adjustments to the layer assignment, symbolic state design, and manual specifications are required, involving yet more manual effort and expertise.

3.3 Opportunities

To overcome the aforementioned challenges, we identify two opportunities regarding DNS authoritative engines and summarization. These insights make DNS engines a particularly good fit for the **just-in-time** (JIT) summarization approach, allowing Iceberg to achieve verification while reducing manual refinement effort.

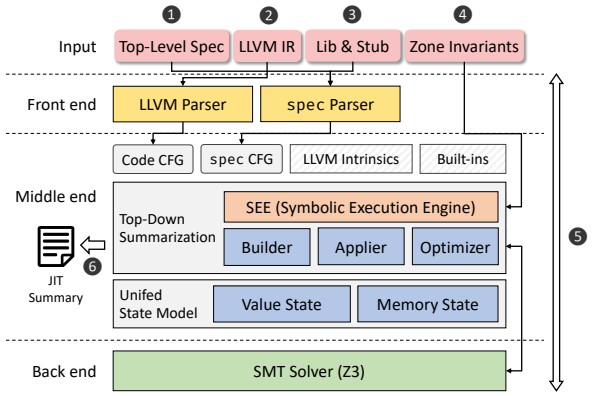


Figure 4. Iceberg system and workflow overview.

DNS zones provide invariants. DNS authoritative engines operate with pre-configured DNS zones. To serve a query, these zones are traversed and compared against the query, despite varying implementation details. The concrete and structured DNS zone data propagate through code modules, manifesting as fixed arguments in function calls, constant results at memory accesses, and well-formed data structures—all of which can be captured as *invariants* to help with verification. For instance, assume in Figure 3 there is an invariant that `n1` is an in-zone domain name. Values read on lines 10, 14, 19, and 21 would then be constant, and function calls on lines 12 and 25 would have prior knowledge of their arguments. Further, all accesses to `n1` are easily proved to be safe (not causing undefined behaviors, *etc.*) due to the readily set structure in `n1`, which otherwise requires more reasoning.

Invariants simplify summarization. The summary representations, especially for code modules regarding DNS semantics, can be expressed in simpler forms with the help of invariants. In the case of our running example (Figure 3), the complexity of the function behavior is reduced when at least one of the arguments (`n1` and `n2`) is a concrete domain name. This is apparent during summarization in that no out-of-bound access can occur for the concrete name on lines 19 or 20, and the maximum loop count on line 16 is bounded, *etc.* Combined with the fact that the DNS engine only invokes the function for comparing the query name with in-zone names, the summary of the function is *soundly* simplified to comparing a symbolic name to the concrete set of in-zone names. For this and other DNS operations alike, zone invariants lead to a smaller set of paths and a more efficient summary for refinement.

We note that the effectiveness of invariants also extends beyond modules directly related to DNS semantics. For example, a function that pushes a character to the back of a string would benefit from the knowledge that the string is concrete data, which, as a popular optimization [27], rules out special cases like moving the storage from the stack to the heap.

JIT summarization. The key of our approach is to encode code behaviors only when necessary, *i.e.,* summarize just in time. A JIT summary only describes a module in the context where it is used, rather than accounting for all hypothetical contexts that may be impossible due to invariants. In practice, by applying JIT summarization using zone invariants, Iceberg produces sufficiently efficient summaries for *all* layers during the refinement proof, holistically reducing the effort involved in designing, developing, and executing layer proofs, thus achieving low-effort verification.

4 Iceberg Overview

Iceberg is a verification framework for DNS authoritative engines. It accepts program source code and outputs JIT summaries for functions in the program. This section describes the key components and overall workflow of Iceberg.

4.1 Architecture

As shown in Figure 4, Iceberg is organized with a three-layer architecture, consisting of the parsing front end, the summarization middle end, and the solver back end.

Front end. The front end parses input code files and generates control flow graphs (CFGs) of the functions. Iceberg supports two input formats: the LLVM [50] intermediate representation (IR) for the verified program, and `spec`, a simple language we develop for representing specifications in Iceberg.

- LLVM parser. We select LLVM IR as the universal format for programs to benefit from its wide support for programming languages, feature-complete instruction set, and active maintenance. The parser also acknowledges the data layout string in IR files, which enables Iceberg to accurately reason about the memory layout of all data structures and support the optimized and low-level access patterns prevalent in LLVM programs (*e.g.,* accessing members with untyped pointer arithmetic).
- `spec` parser. To achieve verification, Iceberg requires a top-level specification describing the ideal behavior of DNS engines, as well as emulation for library calls and interposing stubs (§6.3). For an integrated specification-writing experience, we provide `spec`, a simple language featuring a Rust-like functional programming paradigm, with higher-order functions and syntax highlighting. Figure 5 shows a piece of the top-level specification for Bind 9.

Middle end. The middle end takes the CFG produced by the front end and conducts symbolic execution to build JIT summaries. After setting up the unified state model (§5.1) with the zone invariants, the symbolic execution engine starts from an *entry* function and summarizes the function behavior, building more summaries as functions are called in new contexts (§5.2), or applying summaries in explored contexts (§5.3). A summary optimizer (§6.2) further optimizes summary representations. See the later sections for details.

```
fn exact_match(
  relevant_rrs: List<Record>, query: Query, zone: Zone,
) -> Response {
  let ns_records = filter::<Record>(
    relevant_rrs, |x| {x.rtype == 2w16}); // |..|{ } lambdas
  if all::<Record>(relevant_rrs, |x| {x.rtype != 6w16})
    && len(ns_records) != 0 {
    let glue_records = glue(zone, ns_records);
    Response {
      rcode: RCode.NOERROR, aa: false, answer: [],
      additional: glue_records, authority: ns_records,
    }
  } else { ... }
}
// Higher-order functions with generics and recursion
fn filter<E>(list: List<E>, f: (E)->Bool) -> List<E> {
  fold::<E, List<E>>(list, [], |filtered, elem| {
    filtered ++ if f(elem) { [elem] } else { [] }
  })
}
fn fold<E, I>(list: List<E>, init: I, f: (I, E)->I) -> I {
  if len(list) == 0 { init }
  else { fold::<E, I>(list[1..], f(init, list[0])), f }
}
```

Figure 5. Code snippets of the top-level specification for Bind 9.

Back end. The back end of Iceberg interfaces with an SMT solver to guide middle-end summarization. We currently use Z3 [37] as the solver implementation. The encoding strategy for SMT constraints is kept deliberately simple. Specifically, Iceberg only encodes scalar-typed expressions (*i.e.,* bitvectors and pointers): literals, variables, symbolics, arithmetic operations, and comparisons, all of which have straightforward equivalents in solver APIs. In contrast, intricate solver features such as uninterpreted functions and arrays are avoided due to their impact on performance.

4.2 Verification Workflow

Iceberg achieves refinement verification by building JIT summaries in a top-down manner. Figure 4 breaks Iceberg's workflow into the following steps:

Step 1. Supply a top-level specification in `spec`.

Step 2. Supply the target program as LLVM IR files.

Step 3. Supply library specifications for emulated library calls and stub functions (§6.3) in `spec`.

Step 4. Supply zone invariants to configure the symbolic execution engine. A memory dump consisting of DNS zones should be provided in JSON format. Iceberg uses this dump file to faithfully set up the initial memory state before summarization. We provide utilities to help generate these memory dump files from the target program (§7.1).

Step 5. Initiate push-button summarization of the entry function. No extra manual specification is required.

Step 6. Check the summary of the entry function for any specification violations. For simplicity, users can write a wrapper function that compares the results of the ideal model and the implementation, picking the wrapper function as the entry. Bugs then manifest as branches with unequal results.

5 Just-In-Time Summarization

To achieve JIT summarization, Iceberg applies symbolic execution to gather collections of constraint-effect paths, extended with just-in-time state splitting to capture invariants. This section demonstrates the details of this process. We begin by describing the unified state model (the domain of constraints and effects) in §5.1, followed by the building (§5.2) and applying (§5.3) of JIT summaries.

5.1 Unified State Model

The state model in Iceberg hosts the symbolic states for all layers throughout the refinement proof, serving as a common interface between JIT summaries. Unlike DNS-V, our holistic adoption of summarization enables Iceberg to operate with one unified domain instead of relying on ad hoc per-layer abstract domains, alleviating the effort of designing layers. Figure 6 summarizes our state model. For example, consider a function `f(void* arg)` and the function call `f(greeting)` where `greeting` is a global variable storing a string literal. The function summary will refer to the argument as `%arg`, which then gets reinterpreted at the call site as `@greeting` that points to the `Const::greeting` region that contains a constant sequence of bytes.

Iceberg memory model. Inspired by KLEE [32] and DNS-V, Iceberg represents memory as a series of disjoint *regions*. Each state is then identified with its region and an integer offset. Unlike these systems, Iceberg tracks the region layout down to the scalar level, computed with the LLVM data layout string as mentioned in §4.1. For instance, for a `Region::int` in Figure 6, the state refers to the scalar member of the region at the given offset. This approach induces less overhead than the byte-addressable memory in KLEE, while remaining more flexible than the typed memory in DNS-V, which is often confused by LLVM's aliased or untyped memory addressing.

Iceberg further distinguishes between the `Stack`, `Global`, `Const`, and `Heap` regions, each with proper lifetime tracking to model behaviors such as dangling pointers, memory leaks, and more complex memory management issues.

5.2 Building Summaries

Iceberg follows Algorithm 1 to build JIT summaries, where JIT-related handling is marked in blue. We elaborate on the details in the rest of this section.

Collecting constraints and effects. Iceberg organizes summaries as trees of `Effects`. Each path from the root to a leaf represents a possible execution path for the corresponding module. Table 2 lists all `Effects` and their definitions.

Initially, an empty summary begins with a `Root` node (line 3 in Algorithm 1). During the depth-first search of the control flow graphs (CFGs), various other `Effect` nodes are grown on the summary tree: upon entering a feasible branch, an `Assume` node is grown (line 9); upon function return, a `Return` node, along with other nodes capturing newly (de)allocated heap

```
State       ::=  ValueState          Intermediate & named values
             |   MemoryState         Values in the memory model

ValueState  ::=  %ident              Local values (e.g., arguments)
             |   @ident              Global values (e.g., functions)

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

Figure 6. Iceberg state model.

Table 2. The catalog of `Effects` in Iceberg.

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]`

regions, memory writes, and the return value, is added (line 12); a `Panic` node is grown whenever erroneous operations (*e.g.,* out-of-bound memory access) occur (line 6).

Handling function calls. Our summarization approach is top-down, meaning that more summaries get built as symbolic execution proceeds. At function calls (line 16), Iceberg checks whether a summary of the called function already exists. If none is available, one is built (line 17) using the up-to-date state model $\mathcal{S}$ carrying invariants. Then, the summary is applied (line 18), enacting updates to the state model $\mathcal{S}$, constraint set $\mathcal{C}$, and the building summary f^{SUM}.

This scheme also handles recursive calls with ease. In cases of recursion (line 14), Iceberg treats recursive calls as *inlined*, merging their CFGs. The resulting constraints and effects are seamlessly integrated into the caller's summary.

Just-in-time state splitting. To capture zone invariants in the state model, Iceberg relies on state splitting [40], a well-established optimization technique for tuning the performance of symbolic execution. Essentially, it splits a single execution path into multiple disjoint paths with added constraints, simplifying state representation in each path. For example, in the program *if (x < 0) {x = 0;} else {x = 5;}*, the joined state $(\text{true}, x = \text{ite}(x < 0, 0, 5))$ can be split into two states: $(x < 0, x = 0)$ and $(x \geq 0, x = 5)$. As discussed in §3.3, Iceberg is particularly suited to this technique due to the zone invariants and their impact on summarization.

Iceberg adopts a specific form of state splitting: limiting a state s in the state model to individual concrete values. Iceberg performs state splitting **just in time** (lines 6 and 18 in Algorithm 1) to grow `Specialize` nodes, capturing the actual values of states in $\mathcal{S}$ and enforcing extra constraints

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
5: ▷ **Entering an instruction node:**
6: Apply instruction on $\mathcal{S}$ *
7: ▷ **Entering a branch node with constraint** c**:**
8: **if** c is satisfiable given $\mathcal{C}$ **then**
9: Grow `Assume` on f^{SUM}
10: $\mathcal{C} \leftarrow \mathcal{C} \bigcup \{c\}$
11: ▷ **Entering a return node:**
12: Grow `Effects` on f^{SUM} accordingly
13: ▷ **Entering a function call node for** g**:**
14: **if** g is recursed **then**
15: Inline the CFG of g; continue
16: **else if** summary of g is not built **then**
17: BUILDSUMMARY(g, $\mathcal{S}$)
18: APPLYSUMMARY(g, f, $\mathcal{S}$, $\mathcal{C}$) * # See Algorithm 2
19: ▷ **Exiting a node:**
20: Revert updates to $\mathcal{S}$ and $\mathcal{C}$, backtrack on f^{SUM}
21: ▷ **JIT state splitting*:**
22: Grow `Specialize` on f^{SUM}, and add constraint to $\mathcal{C}$

Algorithm 2 APPLYSUMMARY

Input: Callee g, caller f, state model $\mathcal{S}$, constraint set $\mathcal{C}$

1: $g^{\text{SUM}} \leftarrow$ summary of g
2: $f^{\text{SUM}} \leftarrow$ summary of f (currently being built)
3: Traverse g^{SUM} in depth-first search (DFS) order
4: ▷ **Entering an `Assume` node with constraint** c**:**
5: **if** c is satisfiable given $\mathcal{C}$ **then**
6: Grow `Assume` on f^{SUM}
7: $\mathcal{C} \leftarrow \mathcal{C} \bigcup \{c\}$
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: $\mathcal{C} \leftarrow \mathcal{C} \bigcup \{s\}$
12: ▷ **Entering a `Return` node:**
13: Bind return value in $\mathcal{S}$, yield to f
14: ▷ **Entering other `Effect` nodes:**
15: Apply `Effect` on $\mathcal{S}$ and f^{SUM} accordingly
16: ▷ **Exiting a node:**
17: Revert updates to $\mathcal{S}$ and $\mathcal{C}$, backtrack on f^{SUM}
18: **if** all sibling `Specialize` failed **then**
19: $g_0^{\text{SUM}} \leftarrow$ BUILDSUMMARY(g, $\mathcal{S}$) # Re-summarization
20: Merge g_0^{SUM} into g^{SUM}
21: Continue through newly merged `Specialize` node

(lines 21 and 22) when any of the following rules applies:

(a) s is used as a pointer offset or an array index.

(b) s is dereferenced as a pointer.

(c) s is evaluated against a `Specialize` during application.

(d) s is marked with the `specialize` primitive in stubs.

Rules (a) and (b) serve as simple heuristics to avoid the most common path explosion scenario: accessing memory with symbolic addresses. Rule (c) ensures coherence between JIT summaries (see §5.3). Rule (d) provides users with fine-grained control over the state-splitting behavior (§6.3).

5.3 Applying Summaries

Algorithm 2 describes the process of applying a JIT summary in Iceberg. Again, JIT-related handling is marked in blue. Due to JIT state splitting, the application of a JIT summary may either succeed (passing `Specialize`) or result in re-summarization (failing `Specialize`).

Passing `Specialize`. When applying a `Specialize` node in a summary, Iceberg forces the corresponding state to be split in the caller's summary as well (line 9). Essentially, Iceberg needs to check whether the rest of the summary, guarded by the `Specialize` node, was conducted in the same context as the application site (*i.e.,* the value of the state matches then and now). If the algorithm passes all `Specialize` nodes (as well as other constraints like `Assumes`) and reaches a `Return` node, symbolic execution simply yields to the caller's CFG (line 13) and continues.

Failing `Specialize`. If, however, the algorithm detects a situation where all sibling `Specialize` nodes fail to match the current state model (line 18), then a new context of the function is encountered that past summarization did not explore. In this case, Iceberg resorts to re-summarization of the function and merges the resulting summary (containing the current context) into the existing summary (containing all past contexts). After merging, the summary application resumes (lines 19-21) through the new `Specialize` node. Summary merging (line 20) is implemented as a straightforward DFS pass comparing the summary trees. §6.2 describes further optimizations performed for summaries in Iceberg.

6 Scaling Summaries

While JIT summarization allows Iceberg to efficiently summarize individual functions, full verification requires combining these summaries effectively. Iceberg employs several techniques to achieve this at scale.

6.1 Symbolic Region

The JIT state splitting mechanism described in §5.2 has a major limitation when handling pointers. Specifically, for pointers to stack or heap regions, splitting states based on exact pointer values leads to overfitting summaries. For example, consider invoking the domain name comparison function in Figure 3 with a stack-allocated name (`%name` in function `@f`) for `n1`. Memory states of this region would be split like:

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

This behavior is problematic because the summary would fail to apply for any `n1` other than `&@f::%name`, despite the function's behavior being independent of where the name is allocated. To resolve this, Iceberg introduces *symbolic regions*: anonymous memory regions that replace concrete stack and heap regions in the summary representation. During building, Iceberg hides the actual stack and heap regions in symbolic states by assigning incremental integer IDs to each unique region encountered. When applying, Iceberg binds these IDs to concrete memory regions when evaluating `Specialize` nodes, allowing the summary to be reinterpreted across different memory regions. Below is the summary snippet with symbolic regions, where `[1]` represents a symbolic region:

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

Iceberg ensures that the mapping between symbolic and concrete regions is one-to-one to avoid issues with pointer comparisons or memory aliasing across regions. The same re-summarization process described in §5.3 applies if symbolic region binding fails. Nonetheless, symbolic regions greatly enhance the reusability of summaries with memory states.

6.2 Summary Optimization

Iceberg represents summaries as trees due to their compatibility with the summary-building process. However, a summary tree generated on the fly may not be optimal when considering back-end invocation overhead. To address this, Iceberg provides an extensible interface for implementing summary optimization passes. Similar to compiler passes, they transform summary representations into more verification-friendly forms. Iceberg natively implements two passes:

- `MinimizeAssume`. This pass removes redundant `Assume` nodes from implied branching conditions during symbolic execution (*e.g.,* after assuming $x < 0$ and $x + y = 0, y > 0$ is implied). Iceberg identifies `Assume` nodes without feasible siblings and removes them, reducing subsequent calls to the solver during application.
- `MergeAssume`. This pass recursively compares and merges subtrees with identical `Effects`, eliminating the preceding `Assume` conditions entirely.

These optimizations significantly accelerate overall verification. For reference, none of the verification processes in §7 terminates without these optimizations.

6.3 Interposing Verification

Iceberg offers interposition capabilities for flexible control over the top-down verification process by accepting *stub* functions written in the `spec` language. A stub function shares the same parameter definitions as the interposed function but always returns a boolean. Right before summarizing the original function, Iceberg evaluates the stub function and interprets its return value as an extra path constraint. Stub functions can be used for:

- Hinting extra invariants during summarization.
- Enforcing state splitting with the `specialize` primitive.
- Printing debug information to the console.

```
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 }
```

Figure 7. The stub for a function when verifying PowerDNS.

Figure 7 shows an example of a stub developed during the verification of PowerDNS. The stub logs the query's `qtype` on line 4, enforces state splitting for a field of `handler` on line 5, and restricts `qtype` on line 7, as those cases are out of the scope of verification.

7 Verification Experience

This section details our experience applying Iceberg to verify four open-source DNS authoritative engine implementations: CoreDNS [24], Bind 9 [10], PowerDNS [26], and HickoryDNS [25]. We focus our verification effort on the core lookup logic, excluding file I/O, networking, concurrency, and other peripheral features such as rate limiting. Figure 8 summarizes our effort. Below, we first describe the common workflow of Iceberg (§4.2) in action, then document how we handle the unique properties of each implementation.

7.1 Common Workflow

Top-level specifications. We derive our base DNS reference model from a list of RFCs covering the core algorithm [1], DNS records [2], error clarification [3], wildcards [7], `rcode` clarification [9], `DNAME` redirection [8], and glue clarification [23]. Notably, RFC rulings are not exhaustive, and engine behaviors may differ in cases not explicitly covered by the RFCs. Thus, we develop separate top-level specifications based on the reference model for each implementation.

Compilation to LLVM IR We use the respective LLVM compilers to obtain the LLVM IR files for each implementation. One technical challenge in this process is LLVM's shift to opaque pointers [21] in LLVM-15 and later versions, which alters the IR representation. Iceberg's LLVM parser and memory model align with the opaque pointer model, allowing Iceberg to parse both flavors of IR with ease.

Library emulation and stubs. Full-path symbolic execution of the four implementations requires emulating a small subset of the Go runtime, the C and C++ standard libraries, and Rust intrinsics. We provide code for emulating these functionalities in `spec`. Figure 9 shows a real emulated library call for PowerDNS. As for the interposing stubs, they are applied accordingly for certain modules (those with a yellow corner

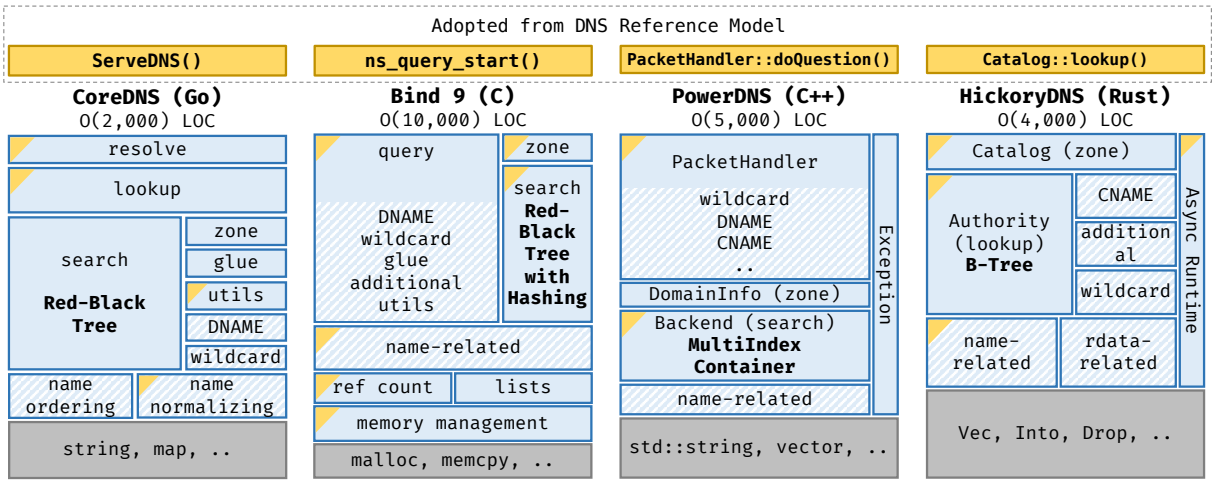


Figure 8. Summary of the verification effort for four implementations. Blue boxes represent code modules (isolated for demonstration) refined with automatic JIT summarization, yellow boxes (top) represent top-level specifications, and gray boxes (bottom) represent emulated libraries. Yellow corners indicate usage of stub functions, while slash patterns denote low-level DNS operations.

Table 3. Verified zone categories and sizes.

Name	Sizes	Count	Source
`simple`	$2 \sim 4$	1000	SCALE [48]
`complex`	$10 \sim 12$	100	Randomly generated
`real`	137 & 393	2	Public (CZDS [11])

in Figure 8) to limit the verification scope or hint JIT state splitting. We rely on timeouts and inspection of summaries to identify the need for adding stub functions.

Zone invariants. We conduct verification of engine correctness on a diverse set of DNS zones, summarized in Table 3. To obtain the zone data as invariants and initialize the state model, we provide implementation-specific utilities. For CoreDNS, we use Golang's runtime reflection to reliably dump the memory related to the configured DNS zones. For other implementations, we derive per-type dumper functions using type definitions in the IR, and then compile with the dumpers to achieve layout-accurate memory dumping.

7.2 Per-Implementation Considerations

CoreDNS. This is a highly modular engine designed around plugins, allowing customization by chaining plugins together. Our target plugin `file` serves queries based on zones parsed from files, where zone records with the same domain name are organized with Golang's native `map`.

Opaque `map`. The implementation of `map` depends on the Golang runtime and is thus opaque in the compiled LLVM IR. While we could emulate most runtime functions faithfully, verbosely modeling the `map` type proved overly complex due to its version-dependent low-level details. Given that the hashmap API is well encapsulated in Go, we opted for a purely abstract hashmap lookup implementation in `spec` that compares keys directly.

Bind 9. As the most widely deployed and time-tested DNS authoritative engine known for its robustness, Bind 9 is the most complex of the four engines we verified. Its verbose C libraries introduce some technicalities during verification:

```
use %"class.std::__cxx11::basic_string" as struct String
    { data: Ptr, length: Int<64>, sso: [Int<8>; 16] }
// std::__cxx11::basic_string<char, ...>::basic_string()
fn string_new(self: Ptr) {
    let _ = store(self |> String.data, self |> String.sso);
    let _ = store(self |> String.length, 0w64);
    store(self |> String.sso, 0w8)
}
```

Figure 9. Emulation for the string constructor in PowerDNS.

Self-managed heap. Unlike other engines with language-level memory management such as garbage collection, RAII [12], or ownership, Bind 9 implements a custom memory allocator on top of `malloc`. This allocator is used extensively during lookups to hold temporary objects, buffers, and response data, rendering Bind 9 prone to resource leaks or use-after-free bugs. Iceberg's explicit modeling of heap regions and heap (de)allocation tracking in JIT summaries (Table 2) helps rigorously verify the soundness of such custom memory management. Nevertheless, annotations of layouts for each `malloc` call are required, as the compiled IR excludes layout information. Throughout our verification, we added 7 type annotations to the IR by matching the source code files.

Hashing. Bind 9 employs hashing during lookups. Unlike CoreDNS, the hashing implementation is not native, and the hash nodes are intrusive. This makes direct emulation with `spec` difficult. Since hashing is notoriously difficult for SMT solvers [69], we replace the hash function in Bind 9 with an identity function before compilation to simplify verification.

PowerDNS. This is a performance-focused authoritative engine with heavy reliance on C++ features. Specifically, its use of C++ exceptions as an integral part of its control flow brings challenges to summarization.

Exceptions. To this end, Iceberg provides an extension that thoroughly emulates C++ exceptions. The emulation consists

Table 4. Bugs found by Iceberg across implementations. The status column indicates whether the bug has been reported (✓, *e.g.,* submitted as GitHub issues) or also acknowledged (☑) by the developers (*e.g.,* classified with the "bug" tag). The SCALE [48] column reports whether the generated tests using the same zones can reveal the bugs. The † symbol denotes issues originating from incomplete fixes of past bug reports.

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.	☑	✗

of three parts. First, it natively supports LLVM's exception instructions, including `invoke` (a special call instruction for functions that may throw), `landingpad` (which implements a catch block), and `resume` (rethrowing). Second, it includes built-ins to emulate the exception system described in the Itanium C++ ABI [14] (which our IR conforms to), where a set of `__cxa_*` functions work together to implement an exception stack with custom data payload and destructors. Finally, we extended JIT summaries (Table 2) to include `Throw`, an alternative to `Return` representing a thrown exception. In combination, these extensions allow Iceberg to summarize PowerDNS functions as effectively as normal functions, without requiring extra manual specifications.

HickoryDNS. Built around robustness and security using only safe Rust, this implementation is the only one of the four that is based on an asynchronous runtime, tokio [17], allowing it to serve multiple requests simultaneously.

Asynchronous runtime. The asynchronous code adds an extra level of complexity to verification, as it transforms the compiled IR and necessitates accurate modeling of the asynchronous control flow and state management. In Rust, asynchronous functions compile into two parts. The first acts as a constructor, copying function arguments into one buffer for centralized access, and the second implements a state machine where different logic is executed based on the current state. Manually designing specifications in this case would have been exceedingly laborious, as developers need to understand tokio's intricate details to produce accurate abstractions. In Iceberg, however, JIT summarization helps automate most of this process through state splitting, exploring only reachable states within the state machine. We rely on stub functions (especially the `specialize` hints) to handle the remaining state transitions and data accesses.

8 Evaluation

We implement Iceberg with over 16,000 lines of Rust code. Given our two goals of achieving correctness guarantees and low manual effort, we evaluate Iceberg based on its overall bug-finding ability, the amount of manual effort required, and its scalability with respect to zone sizes.

8.1 Bug-Finding Ability

By rigorously verifying engine correctness on various zones, Iceberg discovers 12 new bugs across implementations, ranging from minor ordering issues to major RFC misinterpretations and incorrect responses. Table 4 summarizes the bugs we found.[1] In total, we report 8 bugs in CoreDNS, 2 in PowerDNS, and 2 in HickoryDNS. We also confirm previously known issues in Bind 9, albeit no new bugs are identified using our test zones. Nevertheless, we are the first to validate the memory safety of Bind 9's memory management API, as mentioned in §7.2.

Notably, Iceberg's verification approach achieves superior coverage compared to testing methods, allowing us to uncover many more bugs. The rightmost column in Table 4 evaluates the same zones with SCALE [48], a state-of-the-art test case generator for DNS engines. SCALE works by performing symbolic execution on a logical model of DNS engines and materializing the paths as test cases, each corresponding to one pair of zone and query. We add extra constraints to its symbolic engine to fix the zone and force SCALE to output only queries, then inspect whether the queries trigger any bugs. As a result, only 2 out of the 12 bugs are also found by SCALE, demonstrating the effectiveness of our approach.

Moreover, Iceberg has discovered bugs that stem from issues previously found and patched through testing. For instance, bug #2 in Table 4 was thought to be fixed after SCALE's first report. However, we find that the fix was incomplete, revealing a new trigger for the bug that testing cases fail to cover. This discovery highlights Iceberg's ability to identify hidden bugs and ensure correctness guarantees.

[1] Not all reported bugs are acknowledged because developers may decide that they are tolerable (*e.g.,* missing glue is not a critical error). Note, however, that Bind 9 gives the correct response in all these cases.

8.2 Manual Effort

Table 5 quantifies the manual effort required to verify each implementation, measured by the lines of `spec` specification required and the specification-to-code ratios. We use accurate line counts for specifications and estimate code line counts using the LOC-counting tool `cloc`, over a selection of source code files directly related to the core DNS engine (*e.g.,* those implementing the lookup). Since top-level and library specifications are largely shared either by all implementations or by those with the same programming language, they are considered part of Iceberg itself. The remaining stub functions make up less than 10% of the lines of verified code, which is also an overestimation in every case since we only counted directly related code. Below, we present several brief case studies to better put our JIT summarization approach in context:

- Handling recursion. Many implementations make use of recursion to handle zone traversal or DNS aliasing semantics. For example, CoreDNS recurses in its search function along a red-black tree. Iceberg's JIT summarization handles this pattern out of the box. As the recursive calls get automatically inlined (§5.2), JIT state splitting efficiently adapts the summary to the tree structure. The final summary enumerates possible outcomes of the recursive search as if it were written iteratively. In contrast, manual specifications of recursive functions often involve crafting difficult invariants, which can be prohibitive for non-experts.
- Handling data structures. DNS engines utilize various complex data structures (bold text in Figure 8) for organizing DNS zone data. Since DNS zones are considered invariants during our verification, Iceberg can generate effective JIT summaries that state-split based on the existing data structure setup without needing to understand the detailed nuances of each implementation—an otherwise mandatory undertaking with manual specifications. Among the four engines, we cover a variety of data structures, from red-black trees and B-trees to trees extended with hashing, all while demanding little manual effort.

Consequently, we report an initial porting time of one person-week upfront for all verified engines, with subsequent retrofitting across versions taking as little as one person-hour. This low manual effort makes verification with Iceberg affordable for many other engines as well.

8.3 Scalability

Figure 10 reports the verification time for all zone categories mentioned in Table 3 measured on a standard CPU server (`ecs.c9i.4xlarge` on Alibaba Cloud), where all verification runs terminate within a reasonable time: for the `simple` and `complex` categories, all verification finishes in less than half an hour; for the larger in-production zones (`real`), a few hours suffice. Compared across implementations, CoreDNS and HickoryDNS exhibit quicker verification in general, due to their smaller scale of codebases and better encapsulated

Table 5. Manual effort for implementations, measured in lines of `spec` code and spec-to-code ratios (computed with stubs).

	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%

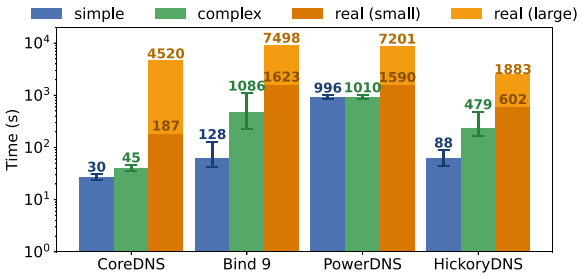


Figure 10. Verification time on different zones. Filled bars denote medians, and error bars indicate the 25th and 75th percentiles.

libraries (represented in Figure 8 with the larger gray boxes for library emulation). On the other hand, Bind 9 verification is relatively expensive, which is anticipated given the verbosity of its C implementation. PowerDNS's verification time is rather long and stable due to the intertwining of object-oriented programming features in C++ (resulting in many more functions in IR) and the exception emulation overhead.

9 Discussion

Trusted computing base. This work assumes the correctness of the LLVM toolchain, the Z3 SMT solver, and our verification framework (*e.g.,* JIT summarization implementation, library emulations, and top-level specifications). Additionally, Iceberg trusts the underlying memory management intrinsics and language-level APIs, such as the low-level `malloc` in C or the garbage-collected `newobject` in Go.

Generality. To achieve automated refinement without extra manual specifications, Iceberg's JIT summarization approach leverages the rich invariants from DNS zones, as well as the overall statelessness of DNS engines and simplicity of constraint-effect patterns in verified modules [72]. These properties allow JIT summaries to remain straightforward and efficient. While Iceberg cannot possibly be a panacea for general verification, the approach does generalize beyond DNS engines. To understand the boundary of Iceberg's capability, it is helpful to think about what the JIT summary would look like for the core of the target software. Take the following (simplified) code from Open vSwitch [13] as an example, which implements some routing logic:

```
1 bool ovs_router_lookup(
2     uint32_t mark, const in6_addr *ip6_dst, in6_addr *src)
3 {
4     struct flow flow = {.dst = *ip6_dst, .mark = mark};
5     struct router_rule *rule;
6     if (src && ipv6_addr_is_set(src)) {
```

```
 7         struct flow flow_src = {.dst = *src, .mark = mark};
 8         struct classifier *cls_local = cls_find(CLS_LOCAL);
 9         cr_src = classifier_lookup(cls_local, &flow_src); ...
10     }
11     PVECTOR_FOR_EACH (rule, &rules) {
12         struct classifier *cls = cls_find(rule->lookup_table);
13         cr = classifier_lookup(cls, &flow); ...
14     } ...
15 }
```

Its JIT summary, given that every `cls` and the global `rules` are considered invariants, would be essentially a collection of "iterate and compare" paths, where the specific algorithm for the lookup does not really matter, and Iceberg should be able to generalize without any fundamental obstacles. In fact, we envision Iceberg to be generally suitable for verifying most network function virtualization systems (typically with fixed control planes), or even database query engines, *etc.*

Soundness. Iceberg achieves sound verification up to the top-level specification, which defines the DNS engine's functional correctness. Indeed, compared to DNS-V and refinement proofs of other systems, our use of JIT summarization in place of manual specifications in lower layers means that Iceberg does not reason about functional correctness at those layers (safety is still ensured, as it is well defined in all layers and violations manifest as failed paths in the top layer). Due to the diversity of engine implementations, Iceberg leaves the choice to users: those with specific needs can opt in for additional specifications in lower layers and further verify individual modules, albeit at the cost of more manual effort.

Limitations. The current modeled list of RFCs in Iceberg (§7.1) does not cover dynamism such as DNS updates [4], caching [3, 5], and security aspects [6]. This is mostly due to the extra effort it would take to develop proper specifications, not a hard restriction on Iceberg's capabilities. However, modeling further temporal behaviors such as rate limiting or cache invalidation over time are indeed too complicated for Iceberg. Besides, the stub function development loop currently relies on manual inspection, which could be improved with recent AI tools (*e.g.,* a small LLM for diagnosis).

10 Related Work

Interactive and auto-active verification. Interactive or auto-active theorem provers are time-tested tools for proving arbitrary system properties. They allow users to specify properties and proof steps, which the verifier checks. This approach has been successful in verifying operating systems [33, 36, 43–45, 54], compilers [51], file systems [34, 35, 73], *etc.*, thanks to its versatility. However, the expertise required makes it impractical for widespread application.

Automated verification. In contrast, automated verification handles the proof steps from implementation to specifications, freeing users from the proof burden. KLEE [32] is a performant symbolic execution engine that forms the basis of a plethora of verifiers. Jitterbug [59], ucheck [61], Timepiece [66], Ellsberg [38], *etc.* [30, 64] extend the approach to microservices, network control planes, network protocols, and more via refinement proofs. Serval [57] constructs reusable verifiers by encoding instruction set semantics in Rosette [67, 68] and provides symbolic profiling [31] to help users iterate the specification design. DNS-V [72] verifies the DNS authoritative engine at Alibaba by performing ahead-of-time summarization for refinement proof, partially removing the need for extra manual specifications. None of these solutions fully addresses the challenge of specification design when verifying large projects like DNS authoritative engines.

Dynamic symbolic execution. For software testing, dynamic symbolic execution [42] is a well-studied approach that leverages concrete values to simplify path constraints, scaled by improvements like compositional refinement [41] and state joining [62]. Nevertheless, it remains fundamentally a test generation approach, improving test coverage by intentionally guessing or solving for concrete input values that trigger *unexplored* paths. Meanwhile, Iceberg's JIT summarization uses concrete zone invariants only as safe fallbacks to prune *impossible* paths, thus achieving verification at scale.

DNS correctness. GRoot [47] is a zone-file verifier for checking DNS configuration validity. It does not verify DNS software. SCALE [48] builds upon GRoot to generate high-coverage test cases for authoritative engines via symbolic execution, augmenting engine reliability in general. Nevertheless, it is agnostic of the implementation and cannot guarantee correctness. DNS-V [72] verifies a DNS authoritative engine against a reference model at the source code level, identifying bugs that are not detected by testing. However, it is limited to one implementation and involves significant manual effort. DNS formal models [55] and analyses [28] do not guarantee correctness of implementations.

11 Conclusion

We present Iceberg, an automated verification framework for DNS authoritative engines that leverages JIT summarization to deliver both correctness guarantees and low manual effort. Iceberg has been applied to four open-source engine implementations, uncovering 12 new bugs, with a low porting cost of one person-week and a specification-to-code ratio under 10%. *This work does not raise any ethical issues.*

Acknowledgments. We thank our shepherd, Mina Arashloo, and the anonymous reviewers for their valuable feedback. This work was supported by the Scientific Research Innovation Capability Support Project for Young Faculty under Grant ZYGXQNJSKYCXNLZCXM-I1, and also the National Natural Science Foundation of China under Grant 624B2007. Xin Jin is the corresponding author. All authors are also affiliated with Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

References

[1] RFC: Domain names - concepts and facilities. https://www.rfc-editor.org/info/rfc1034, 1987.

[2] RFC: Domain names - implementation and specification. https://www.rfc-editor.org/info/rfc1035, 1987.

[3] RFC: Clarifications to the DNS Specification. https://www.rfc-editor.org/info/rfc2181, 1997.

[4] RFC: Dynamic Updates in the Domain Name System (DNS UPDATE). https://www.rfc-editor.org/info/rfc2136, 1997.

[5] RFC: Negative Caching of DNS Queries (DNS NCACHE). https://www.rfc-editor.org/info/rfc2308, 1998.

[6] RFC: Resource Records for the DNS Security Extensions. https://www.rfc-editor.org/info/rfc4034, 2005.

[7] RFC: The Role of Wildcards in the Domain Name System. https://www.rfc-editor.org/info/rfc4592, 2006.

[8] RFC: DNAME Redirection in the DNS. https://www.rfc-editor.org/info/rfc6672, 2012.

[9] RFC: xNAME RCODE and Status Bits Clarification. https://www.rfc-editor.org/info/rfc6604, 2012.

[10] Bind9. https://www.isc.org/bind, 2014.

[11] Centralized zone data service (CZDS). https://www.icann.org/resources/pages/czds-2014-03-03-en, 2014.

[12] Resource acquisition is initialization (RAII). https://en.cppreference.com/w/cpp/language/raii, 2014.

[13] Open vSwitch. https://www.openvswitch.org/, 2016.

[14] Itanium C++ ABI. https://itanium-cxx-abi.github.io/cxx-abi, 2017.

[15] Analyzing the impact of a public DNS resolver outage. https://www.catchpoint.com/blog/google-dns-outage, 2018.

[16] Google Honggfuzz DNS fuzzing. https://github.com/google/honggfuzz/tree/master/examples/bind, 2020.

[17] A runtime for writing reliable, asynchronous, and slim applications with the Rust programming language. https://github.com/tokio-rs/tokio, 2020.

[18] The case of the recursive resolvers. https://slack.engineering/what-happened-during-slacks-dnssec-rollout, 2021.

[19] Observations on resolver behavior during DNS outages. https://blog.verisign.com/security/facebook-dns-outage, 2021.

[20] Understanding how facebook disappeared from the Internet. https://blog.cloudflare.com/october-2021-facebook-outage, 2021.

[21] Opaque pointers: Moving towards a singular pointer type. https://llvm.org/docs/OpaquePointers, 2022.

[22] NMap DNS fuzzing. https://nmap.org/nsedoc/scripts/dns-fuzz.html, 2023.

[23] RFC: DNS Glue Requirements in Referral Responses. https://www.rfc-editor.org/info/rfc9471, 2023.

[24] CoreDNS. https://github.com/coredns/coredns, 2024.

[25] Hickory-dns. https://hickory-dns.org, 2024.

[26] PowerDNS. https://github.com/PowerDNS/pdns, 2024.

[27] Short string optimization. https://en.cppreference.com/w/cpp/language/acronyms, 2024.

[28] Alex Anderson, Aadi Swadipto Mondal, Paul Barford, Mark Crovella, and Joel Sommers. An elemental decomposition of DNS name-to-IP graphs. In *IEEE INFOCOM*, 2024.

[29] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. RWset: Attacking path explosion in constraint-based test generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[30] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *ACM SOSP*, 2021.

[31] James Bornholt and Emina Torlak. Finding code that explodes under symbolic evaluation. In *Proceedings of the ACM on Programming Languages*, 2018.

[32] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX OSDI*, 2008.

[33] Hao Chen, Xiongnan Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward compositional verification of interruptible OS kernels and device drivers. In *ACM PLDI*, 2016.

[34] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *ACM SOSP*, 2017.

[35] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare logic for certifying the FSCQ file system. In *ACM SOSP*, 2015.

[36] David Costanzo, Zhong Shao, and Ronghui Gu. End-to-end verification of information-flow security for C and assembly programs. *ACM SIGPLAN Notices*, 2016.

[37] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[38] Ding Ding, Zhanghan Wang, Aurojit Jinyang Li, and Panda Nyu. Runtime protocol refinement checking for distributed protocol implementations. In *USENIX NSDI*, 2025.

[39] Mihai Dobrescu and Katerina Argyraki. Software dataplane verification. In *USENIX NSDI*, 2014.

[40] Malay Ganai and Aarti Gupta. Tunneling and slicing: towards scalable BMC. In *Annual Design Automation Conference*, 2008.

[41] Patrice Godefroid. Compositional dynamic test generation. 2007.

[42] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *ACM PLDI*, 2005.

[43] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *ACM POPL*, 2015.

[44] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *USENIX OSDI*, 2016.

[45] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. *ACM SIGPLAN Notices*, 2018.

[46] Trevor Hansen, Peter Schachte, and Harald Søndergaard. *State Joining and Splitting for the Symbolic Execution of Binaries*. 2009.

[47] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. GRoot: Proactive verification of dns configurations. In *ACM SIGCOMM*, 2020.

[48] Siva Kesava Reddy Kakarla, Ryan Beckett, Todd Millstein, and George Varghese. {SCALE}: Automatically finding {RFC} compliance bugs in {DNS} nameservers. In *USENIX NSDI*, 2022.

[49] James C King. Symbolic execution and program testing. In *Communications of the ACM*, 1976.

[50] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization*, 2004.

[51] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 2009.

[52] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. A secure and formally verified Linux KVM hypervisor. In *IEEE Symposium on Security and Privacy*, 2021.

[53] Mengqi Liu, Lionel Rieg, Zhong Shao, Ronghui Gu, David Costanzo, Jung-Eun Kim, and Man-Ki Yoon. Virtual timeline: a formal abstraction for verifying preemptive schedulers with temporal isolation. *ACM POPL*, 2019.

[54] Mengqi Liu, Zhong Shao, Hao Chen, Man-Ki Yoon, and Jung-Eun Kim. Compositional virtual timelines: verifying dynamic-priority partitions with algorithmic temporal isolation. *Proceedings of the ACM on Programming Languages*, 2022.

[55] Si Liu, Huayi Duan, Lukas Heimes, Marco Bearzi, Jodok Vieli, David Basin, and Adrian Perrig. A formal framework for end-to-end DNS resolution. In *ACM SIGCOMM*, 2023.

[56] Carroll Morgan. Programming from specifications. In *Prentice Hall International Series in computer science*, 1990.

[57] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with Serval. In *ACM SOSP*, 2019.

[58] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an OS kernel. In *ACM SOSP*, 2017.

[59] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In *USENIX OSDI*, 2020.

[60] Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, et al. VSync: push-button verification and optimization for synchronization primitives on weak memory models. In *ACM ASPLOS*, 2021.

[61] Aurojit Panda, Mooly Sagiv, and Scott Shenker. Verification in the age of microservices. In *ACM SIGOPS HotOS Workshop*, 2017.

[62] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. MultiSE: multi-path symbolic execution using value summaries. In *Joint Meeting on Foundations of Software Engineering*, 2015.

[63] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *USENIX OSDI*, 2016.

[64] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. Nickel: A framework for design and verification of information flow control systems. In *USENIX OSDI*, 2018.

[65] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. Formal verification of a multiprocessor hypervisor on ARM relaxed memory hardware. In *ACM SOSP*, 2021.

[66] Timothy Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. Modular control plane verification via temporal invariants. In *ACM PLDI*, 2023.

[67] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with Rosette. In *ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, 2013.

[68] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Notices*, 2014.

[69] Hui Xu, Yangfan Zhou, Yu Kang, and Michael R. Lyu. Concolic execution on small-size binaries: Challenges and empirical study. In *International Conference on Dependable Systems and Networks (DSN)*, 2017.

[70] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. {DuoAI}: Fast, automated inference of inductive invariants for verifying distributed protocols. In *USENIX OSDI*, 2022.

[71] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. A formally verified NAT. In *ACM SIGCOMM*, 2017.

[72] Naiqian Zheng, Mengqi Liu, Yuxing Xiang, Linjian Song, Dong Li, Feng Han, Nan Wang, Yong Ma, Zhuo Liang, Dennis Cai, et al. Automated verification of an in-production DNS authoritative engine. In *ACM SOSP*, 2023.

[73] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. Using concurrent relational logic with helpers for verifying the AtomFS file system. In *ACM SOSP*, 2019.