# Automated Verification of an In-Production DNS Authoritative Engine

Naiqian Zheng* Peking University
Mengqi Liu* Alibaba Cloud
Yuxing Xiang Peking University
Linjian Song Alibaba Cloud
Dong Li Alibaba Cloud
Feng Han Alibaba Cloud
Nan Wang Alibaba Cloud
Yong Ma Alibaba Cloud
Zhuo Liang Alibaba Cloud
Dennis Cai Alibaba Cloud
Ennan Zhai Alibaba Cloud
Xuanzhe Liu Peking University
Xin Jin Peking University

## Abstract

This paper presents DNS-V, a verification framework for our in-production DNS authoritative engine, which is the core of our DNS service. The key idea for automated verification in general is based on the layered verification principle. However, we face the challenge that our in-production DNS authoritative engine lacks modularity, more specifically, as can be seen with unclean interfaces and poor data structure encapsulation. This makes the layered verification hard to apply. To address this challenge, we propose a summarization approach that performs full-path symbolic execution to accumulate all path conditions and computation effects, and then represents a module's behavior in an abstract form as a set of input-effect pairs. In addition, for portability to future iterated versions of our DNS authoritative engine, we identify common dependency library modules that remain stable across different versions, and carefully design their abstractions to make them amenable to automated reasoning. Our framework has been successful in identifying and preventing tens of critical bugs in different versions of our DNS authoritative engine from reaching production, with a porting effort of less than one person-week.

***CCS Concepts:*** **• Software and its engineering → Software verification and validation**; **• Networks → Naming and addressing**.

*The first two authors contributed equally to this work.

***Keywords:*** Verification; Domain Name System; Formal Methods

**ACM Reference Format:**
Naiqian Zheng, Mengqi Liu, Yuxing Xiang, Linjian Song, Dong Li, Feng Han, Nan Wang, Yong Ma, Zhuo Liang, Dennis Cai, Ennan Zhai, Xuanzhe Liu, and Xin Jin. 2023. Automated Verification of an In-Production DNS Authoritative Engine. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3600006.3613153

## 1 Introduction

Domain Name System (DNS) is one of the most critical infrastructure services on the Internet. It translates human-friendly domain names into IP addresses that computers and routers use to communicate with each other, thus making websites and services on the Internet easily accessible to users. As one of the largest cloud providers in the world, Alibaba Cloud operates a highly available and scalable DNS service that provides managed authoritative for both public and private DNS zones. Our DNS service is deployed globally, serving hundreds of millions of records and $O(10^{12})$ queries per day.

As the core of DNS service, the DNS authoritative engine plays a vital role: it is responsible for matching an incoming query with locally-held authoritative DNS records and computing the content of the DNS response. However, it is hard to implement the DNS authoritative engine correctly, since the DNS authoritative protocol is complex. In particular, many different types of DNS authoritative records are tangled with their diverse processing logics, resulting in complex function logical relations in the implementation of the DNS authoritative engine.

The implementation correctness of the DNS authoritative engine is crucial not only for the entire DNS service, but also for the proper functioning of the Internet. On the one hand, failures of the DNS authoritative engine could disconnect arrays of websites and services from the Internet, resulting in huge business losses [1–3, 5]. For example, on Oct. 7th, 2021, the `.club` and `.hsbc` authoritative servers responded with `SERVFAIL` messages. This failure caused a three-hour outage of resolving `.club` and `.hsbc` domains [4]. On the other hand, incorrect DNS authoritative results may direct users to fake websites and expose them to security risks.

Formal verification of the source code is a well-known methodology to guarantee the absence of bugs. In particular, verifying the *functional correctness* of a system involves defining a formal *specification* that describes the expected behavior of this system, and then using a verifier to *rigorously* check that every possible execution path in the source code meets this specification. A formally verified software is bug-free with respect to its specification.

Therefore, We decided to build DNS-V, a verification framework that not only checks our in-production DNS authoritative engine automatically during its development, but is also easily portable to the iterated versions of our DNS authoritative engine. However, building such a verification framework requires us to address an important challenge.

**Technical challenge: our DNS authoritative engine lacks modularity, more specifically, as can be seen with unclean interface and poor data structure encapsulation. This makes the layered verification hard to apply.** *Layered verification* [20, 22] is a key insight in recent advancements of formal verification for large systems. To tackle the complexity of verifying large systems, this approach decomposes a system into a set of layers. Each layer encapsulates all behaviors of its source code into an abstract specification, and proves that invoking this specification is equivalent to invoking the corresponding source code. In this way, the source code within each layer can be verified independently, since it only depends on abstract specifications of lower-layer functions.

A fundamental prerequisite of this approach is that the source code should follow a layered design pattern, thus allowing us to define a concise specification for each layer. Prior work [18, 20–22, 32, 34, 35, 39, 44, 51, 52, 57] typically builds software from scratch (or retrofits existing ones) to ensure that it is designed with modularity and consists of well-defined interfaces between modules, making the layered verification straightforward and effective.

The formal verification of our DNS authoritative engine, however, does not have this flexibility, making it a poor target for layered verification. First, even though the intended behaviors of our DNS authoritative engine are well documented in production, the functionalities of individual modules are typically ambiguous or unclear. Thus, the key step of defining each module's expected behavior is hard, if possible at all. This is partly due to the fact that the in-production system is the result of numerous rounds of iterations, containing both legacy code and small increments adding new features. The interface keeps getting increasingly more intricate unless a major refactoring happens. Second, our in-production DNS authoritative engine is not designed with modularity in mind. It could have tightly-coupled modules and data structures, making it difficult to be divided into layers. As a comparison, Gu *et al.* [20] report that a software module suitable for verification should encapsulate its internal data and only expose well-defined interface functions, thus ensuring that external code always has a consistent abstract view of this module. Our in-production DNS authoritative engine, however, violates this assumption, making the abstraction process challenging.

**Our verification framework.** In this paper, we still follow the layered verification principle to break down the complexity of the in-production system, building a practical framework to the verification of our DNS authoritative engine. We use symbolic execution [26] to achieve automated reasoning. We address the above-mentioned technical challenge in verifying in-production DNS authoritative engine with the key idea of *summarization.*

We extend the layered approach by automatically computing *summary specifications* for modules that lack well-defined functionality. More specifically, summarization works by performing full-path symbolic execution and accumulating all path conditions and computation effects (*e.g.*, writes to memory, return value, *etc.*). It then represents a module's behavior in an abstract form as the set of input-effect pairs collected above (see §6.4 for an example). In this way, we keep the verification modular and efficient, while avoiding the difficulty of manually designing abstractions for poorly-designed in-production modules (§5.3). Note that summarization is challenging in general since a module's computation effects can be rather complex. However, we observe that resolution modules in the DNS authoritative engine follow limited effect patterns that make them amenable to automated summarization (§4.2). Further, we design our verifier's memory model in a flexible way to accommodate poor data structure encapsulation in in-production DNS authoritative engine. In particular, a data structure can be partially abstracted and abstract specifications no longer have to distinguish between abstract and concrete data (§5.1).

By adopting this idea, we build DNS-V, a verification framework for our in-house developed DNS authoritative engine. In particular, we design a top-level specification for the core logic of our DNS authoritative engine, formally verify a base version of our DNS authoritative engine against this specification, and keep porting the verification to newer versions. To enable efficient summarization for resolution modules, we identify common dependency library modules that remain stable across different implementation versions, and carefully design their abstractions. In particular, we design verification-friendly encodings for DNS-related data structures and restrict branch conditions in the specification to simple linear integer arithmetic (§6.3). Our experience shows that DNS-V helps us keep the porting effort below *one person-week*, which is fast enough to keep up with the evolution of our DNS authoritative engine. As we apply DNS-V into the development workflow, we have found and prevented tens of critical bugs in different versions of our production DNS authoritative engine. We also adapt the top-level specification to accommodate new features. This process is still ongoing with the active development and maintenance of our DNS service.

## 2 Background

DNS translates human-readable domain names into machine-readable IP addresses. As the core of DNS, the DNS authoritative engine matches incoming queries with local authoritative DNS records and composes the DNS response. Since this paper focuses on the DNS authoritative engine, we detail important terminologies below.

**Domain name, zone, and resource records.** A *domain name* consists of a list of labels, separated by the dot symbol. This is similar to the structure of the path name in a file system but in reverse order. DNS utilizes this tree-like structure to partition the domain name space among various service providers. For example, a web service that owns the domain name "example.com" has all names within this sub-domain tree (*e.g.*, "foo.example.com"). All these names form a *zone*. The domain owner configures subdomains within a zone by registering this zone along with its resource records onto the authoritative engine. A *resource record* (RR) contains the following major components.

- *R*name: the domain name of this record.
- *T*ype: the type of the resource, e.g., A is for IPv4 address and AAAA is for IPv6 address.
- *R*data: the data content of this resource, e.g., the exact IPv4 address for an A-type RR.

**Authoritative name resolution: DNS query and response.** A DNS query contains the requested domain name called `qname`, and the requested RR type called `qtype`. Upon receiving a query, the authoritative engine fetches the relevant zone configurations and looks up RRs that match both the `qname` and the `qtype`. It then composes a response message, containing the query, and the following important fields.

- *R*esponse code (`rcode`): whether this query is successful, and possible reasons in case of failure.
- *A*uthoritative answer (`AA`): a flag indicating whether the queried nameserver holds authoritative RRs for this query.
- *A*nswer section: RRs that answer the query.
- *A*uthority section: RRs that provide information about the authority of nameservers.
- *A*dditional section: RRs that in most cases provide optional information to the query.

$$\begin{tikzcd}
s'_0 \arrow[rr, "\text{Execution on spec}"] \arrow[d, dash, "R"'] & & s'_f \arrow[d, dash, dashed, "R"] \\
s_0 \arrow[r] & \text{......} \arrow[r] & s_f \\
& \text{Execution on code} &
\end{tikzcd}$$

**Figure 1.** The refinement proof between a piece of concrete code and its abstract specification.

## 3 Challenges with Verifying In-Production DNS Authoritative Engine

Verifying our entire DNS software, which contains $O(10^5)$ lines of code in Go, is impractical and unnecessary. As abovementioned, the DNS authoritative engine is the core of the DNS service; thus, we construct a verifier for the DNS authoritative engine.[1]

### 3.1 Our Goals

**Goal 1: Formal verification.** Our first goal is to formally verify our in-production DNS authoritative engine. Our DNS authoritative engine consists of 2,000 LOC in Go and makes intensive use of complex control flows, such as nested unbounded loops, that make its automated reasoning challenging. Our verification specifically targets two properties:

- Safety: the DNS authoritative engine should not cause runtime error given any incoming `qname` and `qtype`.
- Functional correctness: the returned DNS response is correct for any query, with respect to a top-level specification that we develop.

**Goal 2: Great portability to iterated versions.** Our verification should not focus on only a certain version of our DNS authoritative engine. To accommodate the diverse needs of our customers, our developers iterate the DNS authoritative engine frequently. Our experience shows that every version of the engine will almost always contain bugs, either caused by code increments adding new features or by imperfect patches intending to fix previous bugs. Rather, we observe that the practical way of ensuring the safety and correctness of the DNS authoritative engine is to perform formal verification on it in a continuous manner with the iteration of its source code versions. In order to achieve this, we need to build our verification framework to be highly adaptive to accommodate source code updates with low effort.

[1]As a comparison, other components (*e.g.*, those for encoding and decoding packets, and logging) exhibit relatively more straightforward functionalities and have less influence on the correctness of DNS responses. In Alibaba Cloud, traditional testing techniques for these modules are enough.

## 3.2 Preliminary: Refinement & Symbolic Execution

**Refinement-based verification [8, 20, 33, 40, 56].** The formal verification of safety and functional correctness properties is usually achieved by building refinement proofs, whether manually or automatically. This involves a piece of concrete code code, its abstract specification spec, and a simulation relation R that associates concrete states with abstract ones. As shown in Figure 1, assume that the initial concrete state $s_0$ and abstract state $s'_0$ satisfy $R(s_0, s'_0)$. Also, assume that executing code results in the final state $s_f$. If there exists $s'_f$ such that it is the result of executing spec, and $R(s_f, s'_f)$ also holds, then spec is indeed a faithful abstraction of the code's behaviors.

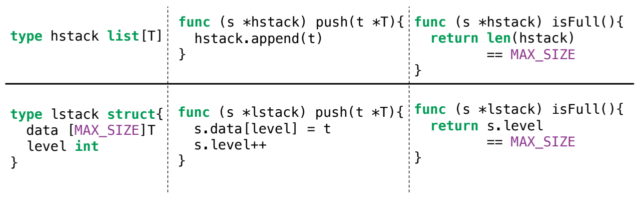
**Symbolic execution.** Symbolic execution [26] systematically explores possible execution paths of a program by replacing concrete inputs with symbolic variables. It generates symbolic expressions for each path, representing its constraints and effects. Full-path symbolic execution is often used by automated verifiers to compute the initial-state-to-final-state transition of a program, *i.e.*, the relation between $s_0$ and $s_f$ in Figure 1. Thus, it is an integral step in automating the refinement verification.

Two factors affect the practicality of symbolic execution techniques significantly: (1) the number of possible execution paths decides the required time for symbolic execution to fully explore a program's behaviors, and (2) the complexity of path constraints affects the overhead of invoking automated solvers, or whether they are automatically solvable at all.

## 3.3 Challenges with Adopting Layered Verification

Layered verification [20, 21, 33] is considered as a key technique to automate large-scale system verification [39]. However, our in-production DNS authoritative engine lacks modularity, as can be seen with the unclean interface and poor data structure encapsulation. This is the key difficulty of in-production system verification compared with state-of-the-art verification. The latter, *i.e.*, previous efforts [20–22, 39], built their systems from scratch to specifically follow a layered design, enabling these systems to fit the layered verification methodology.

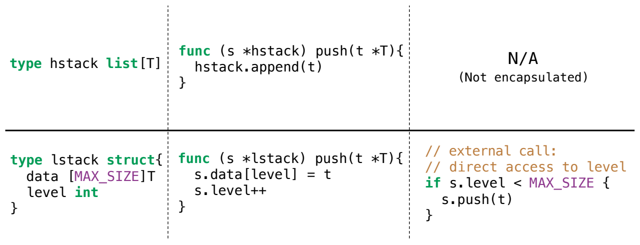
**Unclean interface & function division.** Applying layered verification relies on abstract specifications to encapsulate the behavior of each module. In other words, layered verification works well if the source code indeed follows a layered design pattern. Our in-production DNS authoritative engine, however, is not designed to prioritize modularity. Due to the fact that our in-production DNS authoritative engine evolves constantly, it contains both legacy code and small increments adding new features. The interface also keeps getting increasingly more intricate unless a major refactoring happens. For example, our DNS authoritative engine focuses on computing matching results for incoming queries, while



```
type hstack list[T]      func (s *hstack) push(t *T){   func (s *hstack) isFull(){
                           hstack.append(t)               return len(hstack)
                         }                                       == MAX_SIZE
                                                        }
```

```
type lstack struct{      func (s *lstack) push(t *T){   func (s *lstack) isFull(){
  data [MAX_SIZE]T         s.data[level] = t              return s.level
  level int                s.level++                             == MAX_SIZE
}                        }                              }
```

**Figure 2.** A properly encapsulated custom stack implementation (bottom half) and abstract specification (top half).



```
type hstack list[T]      func (s *hstack) push(t *T){          N/A
                           hstack.append(t)             (Not encapsulated)
                         }
```

```
type lstack struct{      func (s *lstack) push(t *T){   // external call:
  data [MAX_SIZE]T         s.data[level] = t            // direct access to level
  level int                s.level++                    if s.level < MAX_SIZE {
}                        }                                s.push(t)
                                                        }
```

**Figure 3.** A poorly encapsulated custom stack implementation (bottom half). Lifting it into an abstract specification is challenging because both the abstract function (push) and concrete code (bottom right) access its internal field 'level'.

different match types and resource record types require different processing logic. Its source code is full of control flags and a function may exhibit different behaviors when invoked from different execution paths. For such in-production software, it is unrealistic to write an abstract specification for every function (or module).

**Poor data structure encapsulation.** To encapsulate source-code level details of a module into abstract specifications, existing work requires that there is no external access to internal data structures of a module. Rather, internal states are exclusively maintained by the module's own functions. In this way, the entire data structure can be lifted into a consistent abstract representation outside this module. Figure 2 uses a custom stack to illustrate proper data structure encapsulation. Here, the internal state of the stack is only accessible by its functions, push and isFull. In this way, it is easy to prove that its specification (shown on the top half) as an abstract list indeed captures the behavior of these two operations.

However, this property does not hold in our DNS authoritative engine. Production engineers may not always prioritize clean encapsulation and may implement the above stack in a slightly different way. As shown in Figure 3, while the push operation is well-defined and encapsulates the data field, external code still directly accesses the level field to check whether the stack is full or not. This is problematic in a layered verification framework, which does not allow a field to be maintained by both abstract functions and concrete instructions. Such code patterns make it challenging to fit in-production programs into existing layered verification frameworks directly.

### 3.4 Low-Level Implementation Makes Automated Reasoning Difficult

Besides the lack-of-modularity problem, our in-production DNS authoritative engine is filled with various low-level implementations, which pose difficulty to automated verification. Suppose a basic component in the DNS authoritative engine is the comparison between two domain names. Our developers intentionally choose to represent the domain name using raw bytes instead of using high-level language constructs, such as lists of strings. This avoids extra overhead, but results in much more complexity for its symbolic execution and automated reasoning.

Figure 4 shows the pseudo Go code for comparing two domain names (adapted from our production program) when the raw byte representation is used. Here, two names are compared byte to byte from the last position, until they eventually differ or when any one of them reaches the very beginning. Depending on whether they are of equal length, or whether they at least have one common label, this function returns one of three possible results. Our DNS authoritative engine contains a lot of similar low-level implementations, which are invoked over and over by higher-level modules. This significantly increases the difficulty of adopting automated verification on the DNS authoritative engine as a whole.

```go
type RawName struct {
  // e.g. byte array for "www.example.com."
  data        []byte
  // starting offset for each label.
  // e.g., [0, 4, 12]
  offsets     []int
}

func compareRaw(n1 *RawName, n2 *RawName) int {
  l1 := len(n1.offsets) - 1
  l2 := len(n2.offsets) - 1
  lcount := 0
  for l1 >= 0 && l2 >= 0 {
    p1 := n1.offsets[l1]
    p2 := n2.offsets[l2]
    for n1.data[p1] != '.' && n2.data[p2] != '.' {
      if n1.data[p1] == n2.data[p2] {
        p1++
        p2++
      } else {
        break
      }
    }
    if n1.data[p1] != '.' || n2.data[p2] != '.' {
      if lcount > 0 {
        return PARTIALMATCH
      } else {
        return NOMATCH
      }
    }
    l1--
    l2--
  }
  if l1 == l2 {
    return EXACTMATCH
  } else {
    return PARTIALMATCH
  }
}
```

**Figure 4.** Pseudo Go code for comparing two domain names with the low-level raw bytes representation.

## 4 DNS-V Overview

The in-production DNS authoritative engine we aim to verify consists of 2,000 LOC in Go. In Figure 5, yellow boxes represent modules that implement low-level library functions which tend to stay stable across different versions. As a comparison, blue boxes represent modules that carry out the DNS matching operations. They are subject to changes in software iterations. Among them, `Resolve` is the top-level entry point for the DNS authoritative engine.

Unlike other in-production software, the DNS authoritative engine has well-defined functionality: standard authoritative resolution is processed following RFC protocols. We follow these protocols to develop the top-level specification for this engine.

We aim to verify the DNS authoritative engine implementation against this top-level specification. This section gives a high-level overview of how we verify this engine in a modular way. In particular, how we extend layered verification to accommodate the lack of modularity in production code.

### 4.1 LLVM as Frontend

We use GoLLVM [7] to translate the Go source code into LLVM Intermediate Representation (IR) [29] for our verification. Note that the production binary code is not compiled by GoLLVM. The LLVM IR only serves as reference semantics for the DNS authoritative engine. We trust the generated IR to have the same semantics as the Go source code.

The low-level representation brings an immediate benefit—safety checks are also automatically embedded as explicit LLVM instructions. Unsafe behaviors (*e.g.*, array out-of-bound access) are encoded as panic blocks in the LLVM IR. In this way, verifying the safety property is reduced to verifying that these panic blocks are not reachable.

### 4.2 Automated Refinement with Code Summary

We follow existing work [39, 40] and base our verification tool on full-path symbolic execution and refinement proofs. As mentioned in §3.3, naively applying symbolic execution on the entire software would not work. Instead, we follow the refinement-based layered approach and divide the software into smaller modules. For each library module that exhibits well-defined functionality (yellow boxes in Figure 5), we write an abstract specification and use our automated verifier to prove that this specification indeed captures all possible behaviors of its underlying source code.

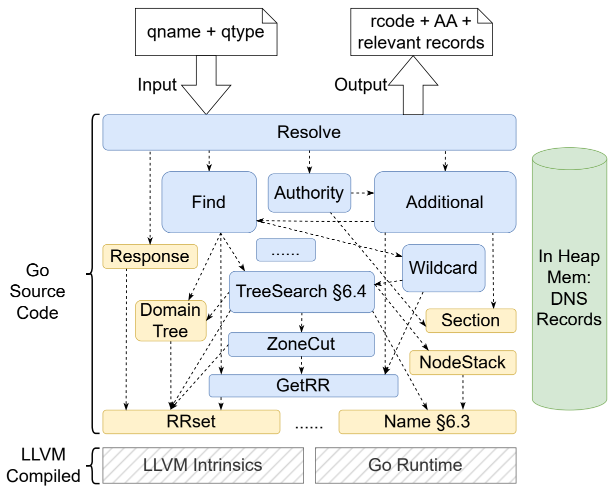


**Figure 5.** Overview of the DNS authoritative engine. Yellow boxes are low-level library functions. Blue boxes are DNS matching operations. Grey boxes are LLVM intrinsics and Go runtime. The green cylinder is the in-heap domain tree.

To address scenarios when in-production modules do not have clear functionality definitions and do not directly fit into a layered verification framework (§3.3), we make the following novel contribution.

**Key idea**: summarization. We perform full-path symbolic execution on a module and accumulate all path conditions and computation effects (*e.g.*, writes to memory, return value). We then represent this module's behavior in an abstract form as the set of input-effect pairs collected above, which we call its summary specification (see §6.4 for an example).

**We automatically generate summary specification for resolution logic that is too complicated for abstraction.** One of the most fundamental challenges in verifying in-production software is the lack of modularity, *e.g.*, unclean interfaces and function division. It is often unrealistic to write a concise and abstract specification for certain functions or modules. This prevents the adoption of layered verification, which requires an abstract specification in place to hide the implementation details of the source code.

However, we observe that such modules are often suitable for automated specification generation through summarization. First, the arguments for these modules have clear input-output patterns. A common pattern among these modules is that most of their input arguments, however complex in heap memory, are read-only by design. Other arguments are dedicated to holding the computation results, and the most common operations on them are setting a new value for a particular field, or appending an item to a slice field. This pattern allows us to associate symbolic input parameters with fields in the input arguments and then perform full-path symbolic execution. For each feasible path, we compare the contents of those arguments holding computation results, identify assignments and appends caused by the execution on this path, then group all paths' conditions and their effects to generate the abstract specification. §5.3 lists the common code patterns that enable straightforward encoding of computation effects.

Second, individual branch conditions in these modules are often simple and can be efficiently checked by SMT solvers. Although these modules consist of a major part of the entire codebase and contain complex computations, *e.g.*, nested loops containing function calls in the body, lots of branches, *etc.*, individual loop conditions or branch conditions are often simple comparisons between two integer variables or pointer variables. Automatically generated summaries on these modules are thus usable by other modules, simply because the conjunction or disjunction of many of these simple clauses is still easily solvable by SMT solvers.

**We design a flexible memory model to support the partial abstraction of data structures.** As illustrated in Figure 2, existing verification techniques often make a clear distinction between the abstract state (operated by abstract specifications) and the concrete memory state (operated by concrete instructions). This makes it hard to reason about in-production software, which may exhibit imperfect data encapsulation (*i.e.*, both abstract specifications and concrete code access the same field), as illustrated in Figure 3.

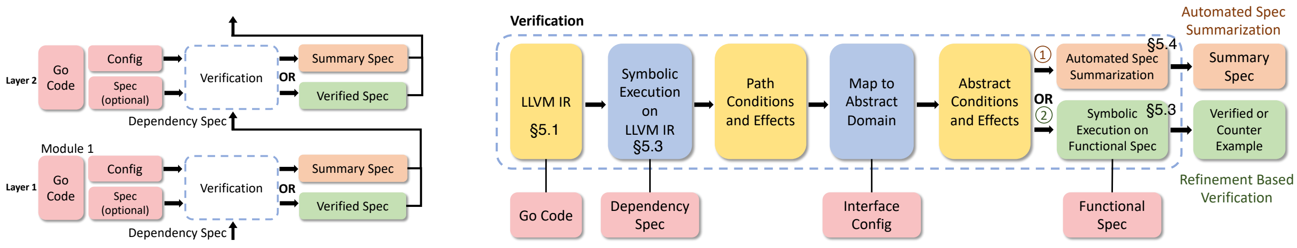
We address this issue by designing a flexible memory model such that abstract specifications become oblivious of whether memory states are concrete or not. In particular, we model the memory as a set of non-overlapping blocks, referenced by LLVM pointers. Each block contains an abstract array or struct, whose contents can be either concrete values or abstract values. In this way, we can partially abstract certain fields within a data structure. While abstracted fields may only be accessed by specifications, the remaining ones can be operated by both specifications and ordinary LLVM instructions. §5.1 gives more details about this design.

## 4.3 Overview of the DNS-V Workflow

Figure 6 gives an overview of the DNS-V workflow. The overall verification follows the layered approach. As a lower layer guarantees that its abstract specification faithfully captures all behaviors of the corresponding Go source code, the higher layer relies on this specification to reason about its own behavior, such that each layer's source code is verified independently. We verify each layer from the bottom to the top (Figure 5). The top-most layer, `Resolve`, is verified against the top-level specification we develop.

For each layer, we verify it by one of the two approaches.

- Refinement-based verification against a manually-provided functional specification. On the right side of Figure 6, red



Overall verification workflow    Verification workflow of a single layer

**Figure 6.** Our verification workflow. On the left, the overall verification follows the layered approach, where a lower layer exposes its specification, instead of source code, to the higher layer. On the right, an individual layer is either verified by refinement proofs or is automatically summarized into an abstract specification.

blocks represent inputs for the verification, including the Go source code, its functional specification, specifications for dependency layers, and an interface config associating concrete variables with abstract ones (refinement relation). We then use GoLLVM [7] to translate the Go source code into LLVM IR. It is then fed into the verifier to perform full-path symbolic execution. The result is a set of path conditions and computation effects. The verifier then maps them to the abstract domain and checks whether all paths are consistent with the specification. Finally, the verifier either decides that the source code and its specification are consistent, or outputs counter-examples.

- Automated specification summarization. Alternatively, we may choose not to supply a functional specification to this verifier. Instead, the verifier aggregates all path conditions and computation effects, then generates an abstract summary specification for this layer.

Choosing which approach to take depends on the particular layer's functionality and implementation. Manual specifications are concise and highly abstracted. Summarized specifications use simple formulas and relatively large-size encodings. As a result, summarized specifications may contain more explicit branches but simpler branch conditions. This could reduce the overall symbolic execution overhead in certain circumstances. For instance, DNS matching operations exhibit complex functionalities due to lots of flags and input arguments. We apply summarization to such layers to avoid having to manually develop complete specifications. On the other hand, domain name comparison (Figure 4) is suitable for manual specification because its logic and functionality are clear. We rely on human insights to abstract the sequence of letter-level comparisons into concise word-level comparisons.

## 5 DNS-V Design and Implementation

This section explains the design and implementation of DNS-V we develop for verifying the in-production DNS authoritative engine. As shown on the right side of Figure 6, we

| | | | |
|---|---|---|---|
| T | ::= | Void | |
| | \| | Bool | |
| | \| | BitVector w | w-bit Integer |
| | \| | Array[n * T] | n-element Array |
| | \| | Struct (field_id → T) | Struct |
| | \| | Pointer T | |
| | \| | List[T] | Abstract List |

**Figure 7.** Types of AbsLLVM values.

explain how we encode the semantics of LLVM IR and abstract specifications (§5.1), how we handle opaque pointers and bitcasts in the IR (§5.5), and how the two verification approaches work—refinement-based verification (§5.2) and summarization (§5.3).

### 5.1 AbsLLVM Language and Memory Model

The verifier has two frontends, one for GoLLVM instructions and another for abstract specifications. Both are translated into the unified AbsLLVM language, which extends the LLVM language with an abstract domain. This allows us to mix a piece of GoLLVM source code with abstract specifications of its dependency functions.

**Types and expressions.** Figure 7 shows types used in the AbsLLVM language. While most types have their counterparts in the LLVM language, List[T] denotes an abstract list that does not correspond to a concrete LLVM type. Our verifier supports circular types, such as a TreeNode that contains pointers to children TreeNodes, which are common patterns in the DNS authoritative engine.

Figure 8 shows expressions in the AbsLLVM language. Most constructs have their roots in the LLVM language. However, *havoc* means any value is possible. Struct field access and Array/List element access are mostly used by abstract specifications, since high-level structs and arrays in the Go source code are usually compiled to pointer and primitive data operations on the LLVM level.

**Flexible memory model.** Our memory model follows CompCert [30], where memory consists of non-overlapping blocks, referenced to by `block_ids`. However, we differ from a usual C memory model in two ways. First, a memory block can

```
aop   ::=  ’+’ | ’-’ | ’&’ | ’|’ | ’ˆ’ | ’«’ | ’»’
bop   ::=  ’&&’ | ’||’ | ’~’
cop   ::=  ’==’ | ’!=’ | ’>’ | ’<’ | ’<=’ | ’>=’
val   ::=  int
       |   ’true’ | ’false’
       |   sym_id                        Symbolic Constant
       |   T ’{’ exp* ’}’                Struct of Type T
       |   T ’[’ exp* ’]’                List/Array of Type T
       |   ’havoc’                       Any Value
       |   block_id | ’null’             Raw Pointers
exp   ::=  var_id                        Variable
       |   val
       |   ’getelementptr’ exp (exp)+    Pointer with Indices
       |   exp aop exp                   Arithmetic Operations
       |   exp cop exp                   Comparison
       |   exp bop exp                   Boolean Composition
       |   exp ’[’ exp ’]’               Array/List Access
       |   (’load’ | ’store’) exp        Load/Store
       |   ’alloca’ T                    Alloca Mem for Type T
       |   func_id ’(’ exp* ’)’          Function Call
       |   ...
```

**Figure 8.** Syntax of AbsLLVM expressions.

hold any `val`, whether it is concrete or abstract. Second, instead of relying on concrete offsets, we follow the LLVM convention and use a list of indices to specify fields within a memory block. In this way, we support the partial abstraction of data structures—abstracting one field in a struct does not interfere with other fields.

We define the memory state as a mapping from `block_id` to `val`. We then implement memory operations (*e.g.*, load, store, and alloca) by properly setting and restoring the corresponding `val`. Unlike existing work that models array accesses using the uninterpreted functions theory, we choose this modeling to accommodate the intensive use of complex data structures, *i.e.*, nested structs and arrays. This design is also based on the observation that memory writes in the DNS authoritative engine mostly happen to concrete fields within structs. For a few library functions that indeed write to variable indexes in an array, we carefully isolate them from other modules and use concretization techniques to facilitate their verification.

Another issue is how to distinguish stack memory between heap memory. This is straightforward in LLVM, for the alloca instruction only creates stack memory, which is freed upon function exit. Heap memory is concrete and is derived from domain tree configurations. Notice that we do not reason about stack or heap resource usage. Instead, we keep the memory model oblivious of concrete data layouts in order to allow partial abstraction of data structures.

## 5.2 Symbolic Execution on AbsLLVM

Given a module’s source code and an abstract specification, we follow prior work [39, 40] to verify their equivalence through refinement proofs (Figure 1). The overall workflow is explained in §4.3. At the heart of it is the symbolic execution on AbsLLVM. We implement the verifier’s symbolic execution engine with about 10,000 LOC of Java. Upon each branch instruction, we translate its condition into SMT expressions and invoke Z3 [17] to check its satisfiability. These conditions may include simple arithmetic comparisons, as well as more sophisticated built-in predicates (*e.g.*, listEq) that we use in developing specifications. We explain more about built-in predicates in §6.1.

## 5.3 Automated Summarization of Specifications

As mentioned in §4.2, summarization is a key technique that we employ to address the challenges with verifying in-production software—the lack of modularity, as can be seen with unclean interfaces and poor data structure encapsulation. Summarization achieves modular verification without having to rely on manually-developed specifications. To put it simple, summarization computes the set of input-effect pairs of a module. Inputs for invoking a module include immediate symbolic values for parameters, and symbolic values that are pointed to by parameter pointers. We rely on a consistent naming convention to associate symbolic values with parameters.

**Representing computation effects.** Representing computation effects can be rather complicated in general, *e.g.*, sequences of stores to variable indexes. We observe that the in-production DNS authoritative engine’s resolution logic modules, which we try to automatically summarize, are computation-heavy and are relatively straightforward in updating the heap memory. In particular, we need to support the following code patterns.

- Allocating a new struct and populating each field. For example, this is useful for wildcard matches, where the DNS authoritative engine makes a copy of the wildcard RR and replaces its `rname` with the actual query name, then stores this altered RR in the DNS response.
- Appending to an array. Although representing generic array stores is hard, the in-production DNS authoritative engine exhibits a simpler pattern: store to a particular index then increment that index by 1. For example, this is used in the custom-defined stack structure, which tracks how a query goes down the domain tree.
- Updating specific fields in a struct. This is the most common way for modules to pass around complex values.

Representing updates to struct fields is straightforward, where we only need to follow the naming convention to associate abstract variables with parameters. We define built-in constructs, `newobject` and `append`, to represent the allocation and appending operations in the summary specification.

**Summary specification.** During the symbolic execution of the LLVM IR, we collect the input-effect pair of each execution path. For the $k$-th path, we collect its path condition $\theta_k$ and effects $f_k$, then express them in an abstract form, *i.e.*,

$\theta'_k$ and $f'_k$. We aggregate all these pairs to form the final summary specification as below.

$$summary\_spec(s'_0) = \begin{cases} f'_0(s'_0) & \theta'_0(s'_0) \\ \ldots\ldots & \\ f'_{n-1}(s'_0) & \theta'_{n-1}(s'_0) \end{cases}$$

## 5.4 Encoding Methodology

We carefully design the encoding of data structures to facilitate the automated reasoning of our DNS authoritative engine. For example, the variable-length list (e.g., for representing queried domain names) is encoded as a series of individual variables for each active list element and a symbolic length variable. The verifier keeps a mapping between elements in the list and symbolic variables defined in the SMT solver context. This encoding is feasible because our codebase never accesses a list element with a random symbolic index. We use the above primitive encoding instead of the built-in sequence type provided by SMT solvers because it provides sufficient expressiveness for our codebase and enjoys efficient automated reasoning.

## 5.5 Resolving Opaque Pointers and Bitcasts

The above verifier design relies on LLVM pointers to be typed, that is, they are constructed from a pointer memory block type with a list of indices to reference fields within the block. However, as the in-production DNS authoritative engine uses complex data structures, the verifier-friendly typed pointers are not always available. In many places, pointers are cast back and forth between the typed ones and opaque ones, where concrete byte offsets are used instead of a list of indices. In fact, LLVM is intentionally evolving toward opaque pointers to facilitate optimizations, so there is no easy way around this challenge when verifying in-production software that uses complex data structures.

We extend LLVM to resolve such opaque pointers and bitcasts. In particular, we track each chain of pointers from the first time it is introduced and utilize the data layout to translate the load/store of opaque pointers to typed ones.

## 6 Verification of an In-Production DNS Authoritative Engine

This section explains how we use the verifier in §5 to formally verify the in-production DNS authoritative engine, which consists of about 2,000 LOC in Go. Developers choose the Go language to enjoy its memory safety features, concurrency support, and easy compilation and deployment.

Notice that the DNS authoritative engine we verify corresponds to the data plane of the DNS authoritative resolution. We rely on the control plane, which is outside this engine, to supply concrete in-heap domain trees as the runtime environment. §9 contains more discussions on how we generate these domain trees.

```
DNSResponse rrlookup(zone: Zone, query: Query){
  let relevant_mask := get_relevant_mask(query,
      zone.zone_name, zone.rrs) in //filter
      irrelevant
  if (relevant_mask == 0){
    let soa_records_mask := filter_mask_rtype_SOA(
      zone.rrs) in
          return DNSResponse(RCode.NXDOMAIN,
               soa_records_mask) //not found
  }else{
        let equal_mask := filter_mask_name_eq(zone.
             rrs, query.qname) in
        if (relevant_mask & equal_mask) != 0{
            exact_match(relevant_mask, query, zone.
                 rrs, false) //exact match
        }
        else{
            // wildcard match
            // ...
        }
  }
}
```

**Figure 9.** Code snippets of the top-layer specification.

## 6.1 Specifications

We design specifications in an executable style (see the AbsLLVM language in §5.1), such that developers without verification background can also understand them. This is important in practice because specifications for dependency libraries and the top-level specification are manually developed, which requires the collaboration between developers from both the verification background and the DNS background. We also define easy-to-use built-in predicates to facilitate the development of specifications. In particular, the top-level specification makes extensive use of list operators, such as conditioned filtering, equality checking, computing the max element, *etc.* We leverage the fact that the in-heap domain tree is concrete to specifically simplify the encoding of the above predicates to make their automated reasoning efficient.

**Top-level specification and guarantees.** The top-level specification describes the whole-program behavior of the DNS authoritative engine. Adhering to RFC standards [19, 31, 36, 37, 55] and custom features, we manually develop the top-level specification (about 200 LOC) to describe the overall authoritative resolution process. Our verification ensures the following two properties for our DNS authoritative engine.

- Functional correctness: given any possible inputs, the verified code always returns the same response as computed by the top-level specification.
- Safety: the verified code does not incur any runtime error. This is entailed by the functional correctness property, since a low-level runtime error (*e.g.*, nil pointers, index out of bounds, *etc.*) would prevent the code from returning expected responses. We verify the absence of such errors by proving for each module that GoLLVM-emitted panics in the IR are never reachable.

We follow SCALE [25] and formalize the behavior of the DNS authoritative engine as a function `rrlookup`. It takes a zone configuration, *Zone*, and a DNS request, *query*, as input, and returns a DNS response, *response*. *Zone* is a list of resource records. A resource record is a set of resource name, resource type and resource data. A DNS query *query* is a pair of query type and query name. A DNS response *resp* is a set of response status, response flag and response data.

Figure 9 shows the skeleton code of the top-level specification. Unlike the production code that traverses a domain tree, the specification groups all zone resource records in a list and carries out the resolution logic by iterative filtering on this list.

## 6.2 Defining Layers

We first follow the layered approach to divide the DNS authoritative engine into layers, such that each layer can be verified independently. Unlike prior work [20, 39], we also decide for each layer whether its specification should be manually developed or automatically summarized.

**Layers whose specifications are manually developed.** These include layers that stay stable across different versions of the DNS authoritative engine, and also layers that require significant data abstraction. We manually develop specifications for the yellow layers in Figure 5.

- `Name`: It defines the encoding of domain names and implements operations such as comparison and subtraction.
- `DomainTree`: A prefix-tree-like structure that organizes all resource records.
- `Response`, `Section`: They define the encoding of DNS responses and three response sections.
- `RRset`, `NodeStack`: They define resource record sets and traversed nodes for `TreeSearch`.

**Layers whose specifications are automatically summarized.** These include layers that are computation heavy and do not involve sophisticated data abstraction. They are evolving across versions as the DNS authoritative engine incorporates new features.

- `TreeSearch`: It walks down the domain tree to match an incoming `qname`.
- `Find`, `Additional`, `Wildcard`, *etc.*: They implement resolution logic such as regular matches, glue record lookups, wildcard matches, *etc.*

## 6.3 Abstraction of Lower-Layer Libraries

As explained in §3.3, one of the major challenges of verifying in-production software is that its source code is often overly complex in order to achieve better performance. For example, Figure 4 illustrates how the `Name` module represents domain names and implements the comparison between

```
// e.g., [int("com"), int("example"), int("www")]
type Name List[Int]

int compareAbs(Name n1, Name n2){
  if (n1[0] != n2[0]){
    return NOMATCH
  }else{
    if (listEq(n1, n2)){
      return EXACTMATCH
    }else{
      return PARTIALMATCH
    }
  }
}
```

**Figure 10.** Abstract specification for `compareRaw`.

them. This byte-by-byte comparison exposes overly complex low-level details, making it unrealistic to naively apply automated verification techniques (*i.e.*, symbolic execution) on the whole-program scale.

We adopt refinement-based approaches and try to lift the `Name` module into a more abstract form. In particular, as one of the most basic and frequently invoked modules, we try to design its specification to be amenable to automated verification, that is, it should consist of primitive types and basic arithmetic operations only. To simplify the specification further, we observe that whenever `compareRaw` is called, the argument `n2` must be concrete, *i.e.*, it is a concrete name from the domain tree node.

Without loss of generality, assume that `n2` represents "www.example.com" and `n1` is a symbolic `qname` that can take any valid value. On an abstract level, the comparison carries out by computing the lexicographical order label-by-label, which are bound to contain no more than 64 characters. This allows us to map labels to integers, drastically simplifying the `compareRaw` operation.

Figure 10 shows the abstract specification for `compareRaw`. Here, `compareAbs` represents a domain name as a list of integers, corresponding to the list of labels, but in the reversed order. Assume that `n2` is represented as [100, 200, 300], corresponding to "com", "example", and "www", respectively. Also assume `n1` is represented as $[n1_0, n1_1, ...]$, while the total length is another symbolic variable, $nameLen_1$. In this case, calling `compareAbs` (`n1`, `n2`) will reduce the first branch condition to $n1_0 != 100$, and the second branch condition to $n1_0 == 100 \land n1_1 == 200 \land n1_2 == 300 \land nameLen_1 == 3$. In this way, we reduce the name comparison into simple comparisons between integer values, making it amenable to automated verification.

We follow §5.2 to prove that `compareAbs` indeed captures all behaviors of `compareRaw`. In particular, we rely on the fact that the total length of `qname` is also bounded to restrict the set of execution paths to be finite.

Other library modules are relatively straightforward and we omit details about their verification.

| Path ID | Example qname satisfying the path condition |
|---|---|
| P0 | example.com |
| P1 | cs.example.com |
| P2 | c.example.com |
| P3 | www.example.com |
| P4 | w.example.com |
| P5 | wwww.example.com |
| P6 | a.www.example.com |
| P7 | web.cs.example.com |
| P8 | w.cs.example.com |
| P9 | zoo.cs.example.com |
| P10 | a.web.cs.example.com |
| P11 | z.cs.example.com |
| P12 | zooo.cs.example.com |
| P13 | a.zoo.cs.example.com |

**Table 1.** All possible execution paths walking down the domain tree.

## 6.4 Summarization of Resolution Logic

Layers implementing resolution logic evolve constantly as the DNS authoritative engine incorporates more features. We automatically generate specifications for them.

For example, `TreeSearch` takes as input a `qname`, then walks down the domain tree to find the deepest relevant matching node for this query name. Meanwhile, the entire path from the root to the final node is stored in a custom stack object. Further, input flags control corner case behaviors, *e.g.*, whether this walk terminates immediately if it encounters an NS-type node (whether further resolutions should be delegated to other servers or not). Manually writing an abstract specification for the above operations is non-trivial.

Figure 11 illustrates an example domain tree. In addition to the usual left and right nodes, a node also has a down node, denoting the beginning of its subdomains. Given a `qname`, `TreeSearch` invokes `compareRaw` (which we have abstracted to `compareAbs`) to compare it with the root node.

- PARTIALMATCH: in this case, `qname` must be a subdomain and `TreeSearch` walks to the down node.
- EXACTMATCH: in this case, `TreeSearch` has found the exact node.
- NOMATCH: in this case, `TreeSearch` looks for the left or right node, depending on the actual ordering.

We perform full-path symbolic execution on `TreeSearch` with this example domain tree to collect all input-effect pairs. In particular, an execution path may terminate at finding an actual tree node, or at finding a `nil` node. Table 1 shows example `qname` that leads to different matching paths on this domain tree. For example, path $P2$ corresponds to the following abstract input-effect pair.

```
    if (nameLen >= 3 && n0 == int("com") && n1 ==
        int("example") && n2 < int("cs")){
      match_result := SUBDOMAIN;
      match_node := NODE("example.com");
    }
```

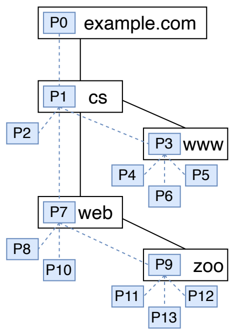


**Figure 11.** An example domain tree. P*s (in blue) indicate execution paths walking down the domain tree. White boxes denote actual (non-`nil`) tree nodes.

Such an abstract summarization captures all possible behaviors of executing `TreeSearch`. Its branch conditions consist of only simple integer arithmetic and comparisons, making the reasoning of higher layers easy.

## 6.5 Concrete Domain Tree

The codebase we verify corresponds to the data plane of the DNS authoritative resolution. Its control plane, which reads a zone file and constructs a domain tree in heap memory, is outside the scope of this work. We rely on the control plane implementation to supply concrete in-heap domain trees as the runtime environment. In particular, we develop scripts to randomly generate thousands of zone configurations, such that each run of the overall verification proves the correctness and safety of the DNS authoritative engine deployed on a concrete zone configuration snapshot.

This is a desirable feature of data plane verification. The concretization of the in-heap domain tree makes full-path symbolic execution feasible. On the one hand, it effectively removes unbounded loops in the recursive lookup code, making the program's behavior finite. On the other hand, it avoids the reasoning of symbolic tree data structures, greatly simplifying the verification.

## 7 Evaluation

This section illustrates the bug-finding ability and porting cost of applying DNS-V. As one of the largest cloud providers, Alibaba Cloud operates a global DNS service. To further improve the reliability of our service, we adopt a dual stack apporach to actively develop an alternative DNS software stack and gradually deploy in parallel with our time-tested solutions. We perform verification on this alternative implementation to ensure its correctness. In particular, DNS-V has been used during the development of our production DNS authoritative engine for two months. We evaluate DNS-V on verifying four different versions of our codebase.

**Bug finding ability.** By performing rigorous verification on our DNS authoritative engine, DNS-V found and prevented

| Index | Codebase Version | Classification | Description |
|---|---|---|---|
| 1 | 1.0 | Wrong Flag | AA flag missing for certain authoritative answers |
| 2 | 1.0 | Wrong Authority | Extraneous NS/SOA authority |
| 3 | 1.0 | Wrong Answer | Incorrect resource record matching on MX |
| 4 | 2.0 | Wrong Additional | Incomplete glue for certain queries |
| 5 | 2.0 | Wrong Additional | Incomplete glue when handling wildcard |
| 6 | 2.0 | Wrong Answer/rcode | Incorrect domain tree search for certain wildcard domains |
| 7 | 2.0 | Wrong Additional | Extraneous records in the additional section |
| 8 | 3.0/dev | Wrong Answer/rcode | Incorrect judgments on certain wildcard domains |
| 9 | dev | Runtime Error | Incomplete bug fix may cause invalid memory access |

**Table 2.** Issues prevented from reaching production by applying formal verification. V1.0 is a base version. V2.0 and v3.0 are iterations containing new features and performance improvements. Dev is the immediate iteration after v3.0.

$O(10)$ extra bugs from reaching production, including both runtime errors and functional correctness violations (*e.g.*, wrong header flags, wrong rcodes, and wrong answers). These bugs were not caught by existing test suites used by our DNS developers. Table 2 shows a subset of bugs found.

We first showcase the following three versions of the codebase—v1.0 (base), v2.0, and v3.0, which represent iterations that contain new features and performance improvements. We observe that every version contains bugs that the existing test suite does not find. On average, less than one percent of execution paths behave incorrectly, such that only specifically-tailored test cases can trigger these bugs. We also observe that not all functional correctness violations are bugs. We iterate our top-level specification when the behavior deviation is intentionally designed for satisfying custom requirements. Moreover, we observe that changes to the source code are prone to producing new elusive bugs (*e.g.*, Issue #6, Issue #7, Issue #8). This demonstrates the difficulty for humans to consider all corner cases comprehensively.

Second, we showcase the dev version, which contains bug-fix patches for v3.0. These patches have passed all test suites before going through DNS-V's rigorous verification. However, we found that (1) some bugs were not completely fixed in one go (*e.g.*, Issue #8), and (2) some new bugs were introduced in the bug-fixing process (*e.g.*, Issue #9). Our verification framework is able to continuously check each version of the DNS authoritative engine, comprehensively explore all its behaviors, and indeed uncovers deep and subtle bugs that have evaded existing test suites.

**Porting efforts.** Porting the verification across different versions of the DNS authoritative engine implementation takes little manual effort even for developers with limited knowledge of formal methods. Developers need to provide specifications of dependencies, interface configurations, and the top-level specification. Specifications of dependencies are carefully designed, but they remain stable across different versions. Interface configurations depend on the concrete code of the software. They are updated when the corresponding code modules' input-effect interfaces are modified.

The top-level specification is developed based on RFCs [19, 31, 36, 37, 55] and documents of custom features. RFC behaviors remain stable across different DNS software versions. Specifications of custom features are relatively short and simple. Table 3 shows porting efforts for the DNS authoritative engine's verification. Porting and verifying each version of software took roughly one person-week each. For each layer, DNS-V takes less than one minute to finish the symbolic execution and automatic summarization (shown in Figure 12).

| | v2.0 | changes v2.0⟶ v3.0 |
|---|---|---|
| **lines of code:** | | |
| implementation | O(2000) | O(200) |
| dependency specification | O(100) | O(10) |
| interface configuration | O(50) | O(20) |
| top-level specification | O(200) | O(10) |
| safety property | O(1) | 0 |

**Table 3.** Cost of verifying one version of the DNS authoritative engine and porting to a newer version.

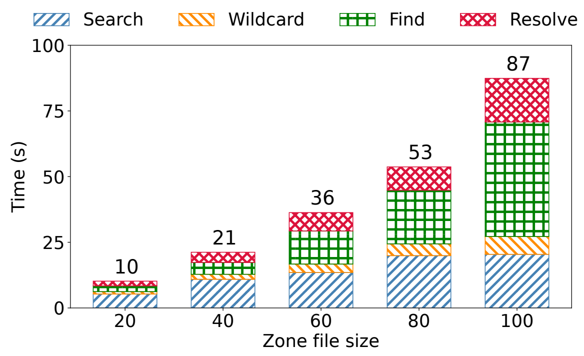


**Figure 12.** Verification time of major layers, when given different zone configuration sizes (number of entries).

Figure 12 shows the verification time for major layers. As the given zone configuration grows in size (number of entries), the verification time increases at a reasonable pace, demonstrating the effectiveness of our verification approach.

## 8 Deployment Experience

This section shares our experience in verifying our DNS authoritative engine. We verified four different versions of DNS authoritative engine, of which v1.0 is a base version, and v2.0 and v3.0 contain new features and performance improvements on top of prior versions. The dev version contains immediate bug-fix patches for v3.0. We explain in detail two representative issues uncovered by DNS-V.

**Issue #4 in Table 2: incomplete glue records.** When the computed DNS response contains `NS` records (*i.e.*, delegation of subdomains to another name server), `Resolve` invokes the name resolution process again to search for IP addresses of this name server. However, in some cases, the incomplete setting of internal control flags changes how a domain tree is traversed and may cause this search to return an empty list of addresses. Thus, no glue record is found, despite that there are resource records containing IP addresses for this name server.

This incomplete computation of glue records is inconsistent with the DNS protocol [36] and is indeed a functional correctness violation. It will cause the user to send additional queries for the address of this name server.

**Issue #9 in Table 2: a bug fix introduces potential invalid memory access** As our verification tool uncovered bugs in the DNS authoritative engine, developers examined these bugs, incorporated patches to fix them, and then went through the same testing and verification processes again. In one iteration, our developers failed to consider comprehensively how the proposed patch would interact with the vast amount of existing code. As a result, our verification tool found that the patch may potentially trigger invalid memory access and cause runtime error. It demonstrated how in-production software can be bug-prone: any code change without thorough consideration could introduce new bugs.

None of the above bugs was caught by existing test suites used by our developers because only carefully crafted DNS queries could trigger such bugs. It is beyond manual efforts to design a set of comprehensive test queries. As a comparison, our verification framework proves to be effective at exploring all possible behaviors of the DNS authoritative engine and uncovering deep and subtle bugs in it.

## 9 Discussion

**Generality.** The approach of DNS-V is not tied to the specific codebase we have verified. It generalizes to other in-production software systems with the poor-interface challenge. Specifically, it is effective for stateless modules, i.e., modules that incur no persistent modifications to heap memory, so that symbolic execution can always start from a clean and concrete initial heap state. This makes it feasible to achieve full-path symbolic execution, and to represent a module's behavior in input-effect pairs, which are essential for summarization. For example, the DNS engine we verify is stateless and does not modify the in-heap domain tree containing DNS resource records across different runs. The stateless property also holds in general for systems that have fixed control plane configurations, including other DNS data plane implementations (e.g., Bind, NSD, CoreDNS), database query engines, software-defined networking systems, *etc.*

**Trusted computing base.** This work verifies our DNS authoritative engine on the source code level. It assumes the correctness of the underlying toolchain—GoLLVM, the Z3 SMT solver, and our verification framework (*e.g.*, the correctness of our LLVM semantics modeling, our summarization implementation, *etc.*). In particular, we trust GoLLVM to generate IR that includes comprehensive safety checks, and we assume that the generated IR exhibits the same behavior as its binary built by Go compiler. Besides, we trust manual specifications of low-level Go library functions (*e.g.*, `Memcpy`, `NewObject`) and the top-level specification. DNS-V focuses on functional correctness and cannot rule out bugs related to resource usage, *e.g.*, stack overflow or out of memory errors.

**Assumptions on code patterns.** DNS-V leverages the fact that the overall behavior of the DNS authoritative engine is stateless, *i.e.*, the in-heap domain tree remains unchanged across different runs. The stateless property eliminates the need for inductive reasoning and for crafting suitable invariants, greatly simplifying the automated verification. This work also assumes effect patterns are limited (*e.g.*, return value, writes to memory, create and modify memory). It enables automated summarization to infer all effects of each layer. Similar to existing automated verifiers, to achieve full-path symbolic execution, this work relies on the assumption that the code does not incur unbounded loops and recursions (given any concrete in-heap domain tree).

**Reliance on concrete in-heap domain trees.** The DNS authoritative engine implements the matching logic given a domain tree that holds DNS resource records. This corresponds to the data plane of the DNS authoritative resolution. Its control plane, *i.e.*, the maintenance of the domain tree, is outside the scope of the DNS authoritative engine. We focus our verification effort on the data plane modules and rely on the control plane to supply concrete in-heap domain trees as the runtime environment. We develop scripts to randomly generate tens of thousands of zone configurations. For each zone, we favor the generation of complex domain names (*e.g.*, containing '*' at various positions) and the intertwining of resource records (*e.g.*, having sub-domains, referring to each other via NS records, *etc.*), such that the domain tree covers diverse matching scenarios.

**Properties verified.** We verify the safety (runtime errors captured by GoLLVM) and functional correctness (with respect to the top-level specification) of the DNS authoritative engine. We do not verify concurrency behaviors (*e.g.*, runtime updates to the in-heap domain tree). Furthermore, we restrict our verification to one-shot DNS queries. We do not reason about properties involving a sequence of queries (*e.g.*, load balancing, memory leak). We do not reason about DNSSEC related behaviors.

## 10 Related Work

**Correctness of DNS.** GRoot zone-file verifier [24] presents the first formalized DNS authoritative resolution semantics. It does not verify DNS software. SCALE [25] performs symbolic execution on the above semantics to systematically generate test cases for DNS authoritative servers. It successfully uncovers bugs related to the general behaviors of DNS resolution. However, it does not analyze the implementation source code and cannot find implementation-specific bugs. Ironsides [11] is a DNS authoritative nameserver proved to be absent of dataflow errors such as buffer overflows. However, it does not provide functional correctness guarantees. DNS fuzzers [43, 50] cannot guarantee the absence of bugs.

**Interactive verification of software.** There is a rich literature on the formal verification of complex software with the aid of interactive theorem provers [6, 12, 33, 42, 45]. Successes in verifying operating system kernels [20] and various extensions [14, 16, 21–23, 34, 35], compilers [30], file systems [13, 15, 58], *etc.*, demonstrate the expressiveness of this approach. Among them, seL4 [27, 28, 48] is the first formally verified operating system kernel. It also enjoys non-interference proof [38] and high-assurance WCET analysis [47]. However, interactive verification often requires years of manual efforts that are prohibitive for verifying in-production software.

**Automated verification of software.** Automated verification often relies on SMT solvers and symbolic execution to prove the equivalence between specifications and implementations. As a consequence, they often need to carefully design (or retrofit) the implementation source code to be amenable to SMT solvers and they restrict the properties to verify. Hyperkernel [40] builds an OS kernel and verifies its functional correctness. It designs a finite interface (*e.g.*, all loop bounds are made constant) to make SMT solving feasible. Jitterbug [41], ucheck [46], *etc.* [9, 49] demonstrate that this approach also applies to verifying compilation, microservices, and other systems, after employing manual proof efforts to break down the top-level theorem into smaller and local properties. Serval [39] builds reusable automated verifiers by encoding instruction set semantics in Rosette [53, 54]. It also utilizes symbolic profiling [10] to help developers identify bottlenecks in automated reasoning. Serval successfully retrofits two prior security monitors to automated verification. None of the above techniques addresses the poor-interface challenge of verifying in-production software.

## 11 Conclusion

We present a verification framework for our in-production DNS authoritative engine. DNS-V is the first tool that not only automatically verifies an in-production DNS authoritative engine but also makes the verification easy to port to newly iterated versions of our DNS authoritative engine. Our framework has been successful in identifying tens of critical bugs in different versions of our DNS authoritative engine, preventing them from reaching production. The porting cost is below one person-week.

**Acknowledgments.** We thank our shepherd, Aurojit Panda, and the anonymous reviewers for their valuable feedback. This work was supported in part by the National Key Research and Development Program of China under the grant number 2022YFB4500700, the National Natural Science Foundation of China under the grant numbers 62325201 and 62172008, the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas), the Beijing Outstanding Young Scientist Program under the grant number BJJWZYJH01201910001004, and Alibaba Group through Alibaba Research Intern Program. Xin Jin is the corresponding author. Naiqian Zheng, Yuxing Xiang, Xuanzhe Liu, and Xin Jin are affiliated with School of Computer Science at Peking University, Center for Data Space Technology and System at Peking University, and Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

## References

[1] 2017. How and why the leap second affected Cloudflare DNS. https://blog.cloudflare.com/how-and-why-the-leap-second-affected-cloudflare-dns/.
[2] 2018. Analyzing the Impact of a Public DNS Resolver Outage. https://www.catchpoint.com/blog/google-dns-outage.
[3] 2021. A DNS outage just took down a large chunk of the internet. https://techcrunch.com/2021/07/22/a-dns-outage-just-took-down-a-good-chunk-of-the-internet/.
[4] 2021. Observations on resolver behavior during DNS outages. https://blog.verisign.com/security/facebook-dns-outage/.
[5] 2021. Understanding how Facebook disappeared from the Internet. https://blog.cloudflare.com/october-2021-facebook-outage/.
[6] 2022. The Coq Proof Assistant. https://coq.inria.fr/.
[7] 2022. Gollvm: LLVM-based Go compiler. https://go.googlesource.com/gollvm.
[8] Anish Athalye, M Frans Kaashoek, and Nickolai Zeldovich. 2022. Verifying Hardware Security Modules with {Information-Preserving} Refinement. In *USENIX OSDI*.
[9] 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. 2021. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *ACM SOSP*.

[10] James Bornholt and Emina Torlak. 2018. Finding code that explodes under symbolic evaluation. *Proceedings of the ACM on Programming Languages* OOPSLA (2018).

[11] Martin Carlisle and Barry Fagin. 2012. IRONSIDES: DNS with no single-packet denial of service or remote code execution vulnerabilities. In *IEEE GLOBECOM*.

[12] Tej Chajed, Joseph Tassarotti, Mark Theng, M Frans Kaashoek, and Nickolai Zeldovich. 2022. Verifying the {DaisyNFS} concurrent and crash-safe file system with sequential reasoning. In *USENIX OSDI*.

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

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

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

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

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

[18] Mihai Dobrescu and Katerina Argyraki. 2014. Software Dataplane Verification. In *USENIX NSDI*.

[19] R. Elz. 1997. *Clarifications to the DNS Specification*. RFC 2181. https://www.rfc-editor.org/info/rfc2181

[20] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *ACM POPL*.

[21] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels.. In *USENIX OSDI*.

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

[23] Xiaojie Guo, Maxime Lesourd, Mengqi Liu, Lionel Rieg, and Zhong Shao. 2019. Integrating formal schedulability analysis into a verified OS kernel. In *International Conference on Computer Aided Verification*. Springer, 496–514.

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

[25] Siva Kesava Reddy Kakarla, Ryan Beckett, Todd Millstein, and George Varghese. 2022. {SCALE}: Automatically Finding {RFC} Compliance Bugs in {DNS} Nameservers. In *USENIX NSDI*.

[26] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* (1976).

[27] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems* (2014).

[28] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *ACM SOSP*.

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

[30] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* (2009).

[31] E. Lewis. 1997. *The Role of Wildcards in the Domain Name System*. RFC 4592. https://www.rfc-editor.org/info/rfc4592

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

[33] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. 2022. Design and Verification of the Arm Confidential Compute Architecture. In *USENIX OSDI*.

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

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

[36] Paul Mockapetris. 1987. *Domain names-concepts and facilities*. RFC 1034. https://www.rfc-editor.org/rfc/rfc1034

[37] Paul V Mockapetris. 1987. Domain names-implementation and specification. https://www.rfc-editor.org/rfc/rfc1035

[38] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. seL4: from general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy*.

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

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

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

[42] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*.

[43] NMap. 2023. NMap DNS Fuzzing. https://nmap.org/nsedoc/scripts/dns-fuzz.html

[44] 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. 2021. VSync: push-button verification and optimization for synchronization primitives on weak memory models. In *ACM ASPLOS*.

[45] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *ACM PLDI*.

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

[47] Thomas Sewell, Felix Kam, and Gernot Heiser. 2016. Complete, high-assurance determination of loop bounds and infeasible paths for WCET analysis. In *IEEE Real-Time and Embedded Technology and Applications Symposium*.

[48] Thomas Arthur Leck Sewell, Magnus O Myreen, and Gerwin Klein. 2013. Translation validation for a verified OS kernel. In *ACM PLDI*.

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

[50] Robert Swiecki and Anestis Bechtsoudis. 2020. Google Honggfuzz DNS Fuzzing. https://github.com/google/honggfuzz/tree/master/examples/bind

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

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

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

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

[55] P. Vixie. 1997. *Dynamic Updates in the Domain Name System (DNS UPDATE)*. RFC 2136. https://www.rfc-editor.org/info/rfc2136

[56] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. 2022. {DuoAI}: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols. In *USENIX OSDI*.

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

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