

Provably Secure DNS: A Case Study in Reliable Software

Barry Fagin and Martin Carlisle

Department of Computer Science
US Air Force Academy
Colorado Springs, CO 80840 USA
{barry.fagin, martin.carlisle}@usafa.edu

Abstract. We describe the use of formal methods in the development of IRONSIDES, an implementation of DNS with superior performance to both BIND and Windows, the two most common DNS servers on the Internet. More importantly, unlike BIND and Windows, IRONSIDES is impervious to all single-packet denial of service attacks and all forms of remote code execution.

Keywords: domain name server, formal methods, software systems, DNS, Ada, internet security, computer security, network security, buffer overflows, , domain name system, denial of service.

(Distribution A, Approved for public release, distribution unlimited.)

1 Introduction

DNS is a protocol essential to the proper functioning of the Internet. The two most common implementations of DNS are the free software version BIND and the implementations that come bundled with various versions of Windows. Unfortunately, despite their ubiquity and importance, these implementations suffer from security vulnerabilities and require frequent patching. As of this writing, according to the Internet Systems Consortium's web site, there are 51 known vulnerabilities in various versions of BIND [1]. Over the past five years, Microsoft has released at least 8 security bulletins relating to vulnerabilities in Windows DNS. Since neither of these products have ever been, to our knowledge, formally validated, it is likely that further flaws remain for hackers to discover and exploit.

The existence of security flaws in such a vital component of the Internet software suite is troubling, to say the least. These vulnerabilities permit not only bad-packet denial of service attacks to crash a DNS server, but in the worst case can actually lead to remote code execution exploits, giving the adversary control over the host machine.

To address this problem, the authors have used formal methods and the SPARK tool set from Praxis Systems [2] to develop a high-performance version of DNS that is provably exception-free. We first give a brief overview of DNS, and our implementation of it using the SPARK tools. We then describe our experimental test bed and the results we obtained. We conclude with lessons learned and directions for future work.

2 Overview of DNS

DNS is an abbreviation for the Internet's Domain Name System. Theoretically it is a naming system for any resource connected to the Internet, but in practice it associates host names (www.cnn.com) with IP addresses (157.166.226.26). The DNS protocol was developed by Paul Mockapetris, first codified in IETF documents RFC 882 and RFC 883 and later superseded by RFC's 1034 and 1035. Clients of a DNS server interact with it supplying queries of various types, with the server providing the answers. Communication between a DNS client and server takes place at either the UDP or TCP layers of the Internet protocol stack.

The distinguishing feature of DNS is its hierarchical and distributed nature. Because it is hierarchical, a single DNS server may not and need not know the answer to a client query. If it does not, it can query another DNS server at a higher level in the Internet domain name space for further information. This process may be repeated up to the root server, with further information then propagating back down to the original querying server.

The system's distributed nature means that there is no central DNS server. Hundreds of thousands of implementations of DNS are all running at once, and because they all use the same protocols to communicate they all function correctly.

Simple implementations of DNS may perform solely as authoritative name servers, responsible only for managing the IP addresses associated with a particular zone. To reduce the load on the root zone servers and to improve performance of applications that rely on nearby DNS servers, more complex implementations of DNS may cache query answers as well as fully implement the recursive query protocol described previously.

The most popular implementation of DNS is the Berkeley Internet Name Domain server, or BIND. Originally written in 1984, it has been ported to a number of systems and compilers, and has been distributed as free software since its inception. According to the Wikipedia entry on DNS, it is the dominant name service software on the Internet. However, numerous alternatives remain available, including implementations bundled with Microsoft Windows.

3 SPARK: A Tool For Creating Provably Correct Programs

The SPARK language and toolset from Altran Praxis is used in the creation of software systems with provable correctness and security properties. SPARK is a subset of Ada, augmented with special annotations. These annotations appear as

ordinary comments to Ada compilers, but are parsed by SPARK's pre-processing tools used to validate the software. SPARK is a fairly mature technology and has been used on several projects [3-5]. Accordingly, given our prior institutional experience with Ada (see for example [6]), we chose SPARK and Ada as the platform for constructing DNS software that would not be subject to most of the vulnerabilities of BIND and Windows versions currently deployed around the globe.

4 Overview of IRONSIDES

IRONSIDES is an Ada/SPARK implementation of the DNS protocols. The IRONSIDES authoritative DNS server was described previously in [7]. Since that publication, off-line signed DNS records have been added to IRONSIDES using DNSSEC, the protocol that adds encryption to DNS transactions to further reduce vulnerability to spoofing and other attacks [8]. Below we describe the architecture of the IRONSIDES recursive service. In actual operation, both versions would be running concurrently.

The high level structure of the IRONSIDES recursive service is shown in Figure 1:

RECURSIVE QUERIES

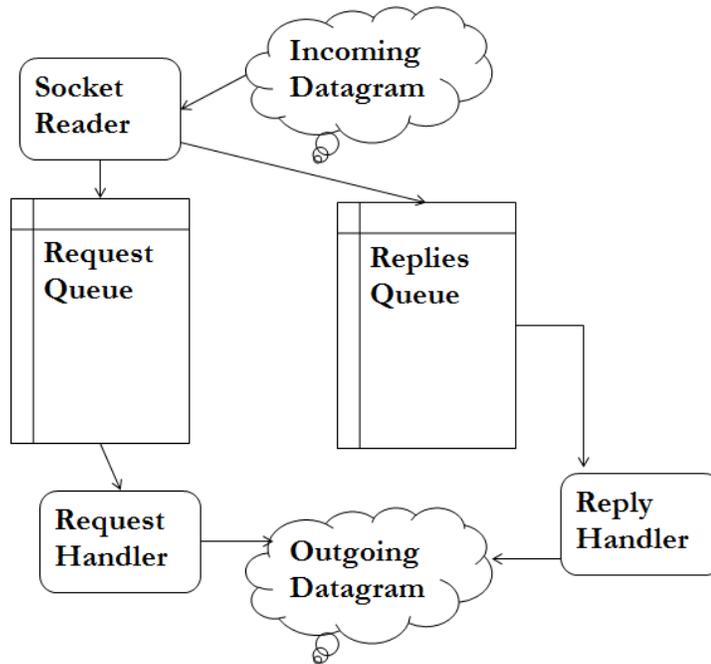


Fig. 1. High-level structure of IRONSIDES recursive service.

Incoming DNS messages are either queries from a “downstream” client, in which case they are placed in a request queue, or responses from an “upstream” server, which are placed in a reply queue. Queries are checked against a DNS record cache (not shown). If appropriate matching records are found, a response DNS message is constructed and sent out on the wire back to the requesting server. Otherwise, the query is forwarded to one or more upstream servers. Responses are sent back to the original requesting server and stored in the DNS record cache. The modules that implement the above structure and their data dependency relationships are shown below. Lines indicate a data dependency from the module above on the module below. Transitive dependencies are implied.

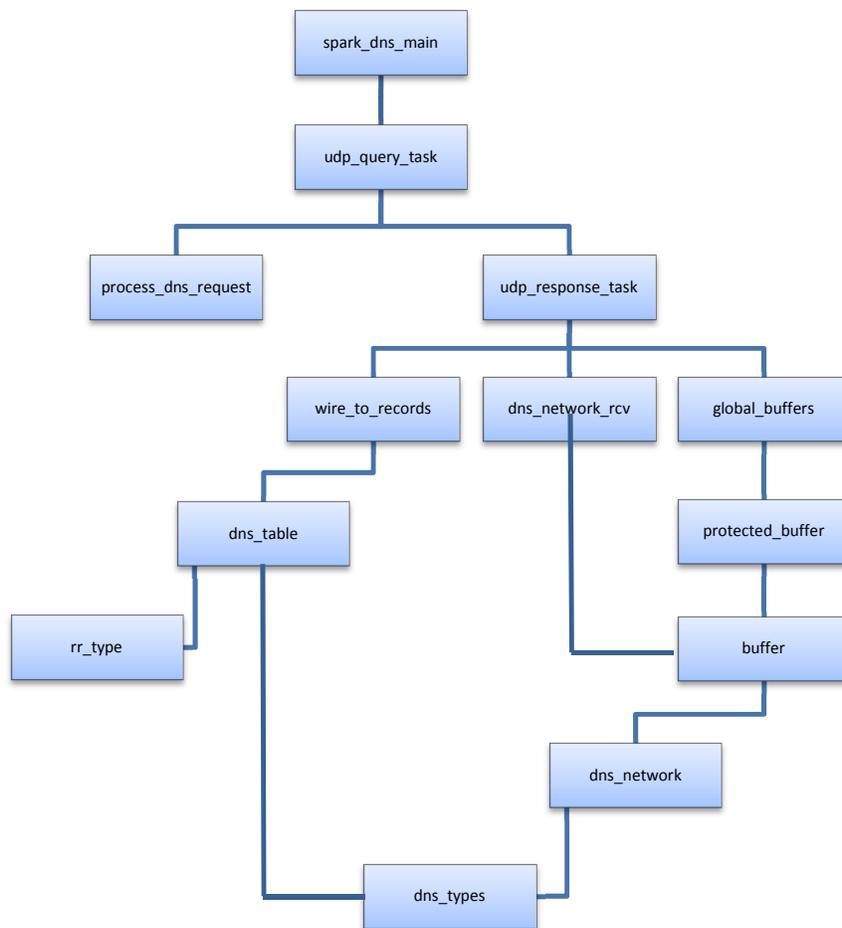


Fig. 2. Module and data dependency representation of IRONSIDES.

The functions of these modules are as follows:

- spark_dns_main: Top-level executable
- udp_query_task: Concurrently executing task responsible for all incoming DNS traffic
- udp_response_task: Concurrently executing task responsible for managing all responses from upstream servers
- process_dns_request: Interprets incoming packet, queries DNS table, queues query if answer not found
- wire_to_records: Builds DNS resource records from DNS packets on the wire
- dns_network_rcv: SPARK wrapper for network traffic to guarantee no overflows
- global_buffers: Query and response queues
- protected_buffer: ADT for the query and response queues
- buffer_pkg: ADT for a queue
- dns_table: Cache of DNS resource records
- rr_type: Top-level package for all DNS resource record types
- dns_network: Handles low-level network IO
- dns_types: Data types for working with DNS packets

As a result of the software validation process, IRONSIDES code is known to be free of uninitialized values, data flow errors (e.g. writes that are never read or values derived from incorrect sources), array bounds errors, and all runtime exceptions. This renders it invulnerable to single-packet denial of service attacks and all remote execution exploits. If IRONSIDES is properly compiled and configured, it cannot be taken over as a result of any external input, no matter when the input arrives and no matter how it is formatted. Also, it cannot be crashed and all its loops are guaranteed to terminate, which renders it invulnerable to denial of service attacks that rely on badly formatted packets.

Current statistics on the proof requirements and code size of IRONSIDES authoritative are shown in Tables 1 and 2.

Table 1. Proof requirements of IRONSIDES authoritative

	Total	Examiner	Simplifier	Victor
Assert/Post	3106	2209	884	13
Precondition	561	0	532	29
Check stmt.	12	0	12	0
Runtime check	3750	0	3704	46
Refinement. VCs	44	42	2	0
Inherit. VCs	0	0	0	0
=====				
Totals:	7473	2251	5134	88
%Totals:		30%	69%	1%

Table 2. IRONSIDES source lines

Total Lines:	11598
Blank Lines:	871
Non-blank non-comment lines:	7543
Lines of SPARK annotations:	1133
Semicolons:	5403

A “Verification Condition”, or VC, is a theorem that must be proved in order for SPARK to consider the program as validated. Typical VC’s include assertions that integers do not overflow or wraparound, that array bounds are not exceeded, and so forth. Simpler VC’s are proved by the Spark Examiner. More complicated ones are proved by the Verifier. According to AdaCore Technologies [9], over 95% of VCs are proven automatically by the SPARK toolset. In our case, this was 99%. We were unwilling to allow any VCs to remain unproven, lest they be false and lead to a security vulnerability. Consequently, we used Victor, a wrapper for the advanced Satisfiability Modulo Theories (SMT) solver Alt-Ergo, developed at the University of Paris-Sud [10], to prove the final 1%. Readers interested in learning more about the SPARK tool set are referred to [2].

We see from Table 2 that the overhead of SPARK annotations in terms of code size and typing time is negligible, approximately ten percent of the total number of lines in the program.

5 Experimental Results

Having software that is crash-proof is valuable, but unless its performance is comparable to existing implementations it is not likely to be accepted by the user community. System administrators, if faced with the choice, might regard software vulnerabilities as acceptable risks if fixing them significantly impacts performance. Furthermore, from a computer security research perspective, it would be useful to understand the nature of the tradeoff between security and performance, or even better to discover that in at least some cases no such tradeoff is required. We present here the results of a case study performed to better understand these questions.

Previous work [7] compared the performance of the IRONSIDES authoritative server to BIND running on a Linux system (Ubuntu 11.0). We now present results comparing the performance of IRONSIDES authoritative with BIND and Windows DNS on Windows Server 2008. As in [7], we use the DNS stress testing tool ‘dnstest’ [11]. Because IRONSIDES is still in development, it does not yet have the feature range of BIND or Windows DNS (though we are continually adding more features and the gap is rapidly closing). Any comparison should take these differences into account. Following the style of [12], we show a comparison of these three DNS packages below. Footnotes and parenthetical comments for BIND and Windows are omitted to save space.

Tables 3a/3b. Comparison of BIND, Windows and IRONSIDES functionality

Server	Authoritative	Recursive	Recursion ACL	Slave mode	Caching
BIND	Y	Y	Y	Y	Y
Windows DNS	Y	Y	N	Y	Y
IRONSIDES	Y*	in progress	N	N	in progress

DNSSEC	Server	TSIG	IPv6	Wildcard	Free Software	split horizon
Y	BIND	Y	Y	Y	Y	Y
Y	Windows DNS	Y	Y	Y	N	N
offline- signed	IRONSIDES	N	Y	N	Y	N

*The following resource record types are currently supported: A, AAAA, CNAME, DNSKEY, MX, NS, NSEC, PTR, RRSIG, SOA.

Our experimental test bed is shown in Figure 3:

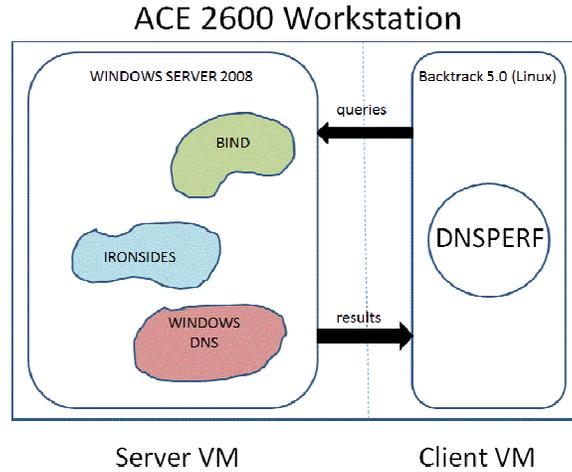


Fig. 3. Experimental test bed for performance comparisons of DNS software

'dnstperf' runs on a Backtrack 5.0 client virtual machine. A Windows Server 2008 virtual machine is loaded as a server. Testing is done by starting up the DNS server to be tested under the server virtual machine, and then running dnstperf. Only one DNS server is active at a time.

Since the purpose of the experiment is to measure the computational performance of the server, both VMs are loaded on the same computer, in this case an ACE 2600 Workstation with 8GB of RAM. Using the same computer for client and server eliminates the effect of network latency. 'dnstperf' issues queries over the standard DNS port to whichever server is listening. The server in turn responds as appropriate. At the end of a run, the tool generates a performance report.

We performed three test runs for three DNS implementations and then averaged the results, scaling them to queries per millisecond. The raw data are shown in Table 4. Averaged results are shown in Figure 4:

Table 4. Comparison of DNS software (queries per second for three test runs).

BIND		
16478.3	16667.9	17020.0
IRONSIDES		
37329.1	37814.6	37024.4
Win DNS		
34188.0	35676.1	35089.3

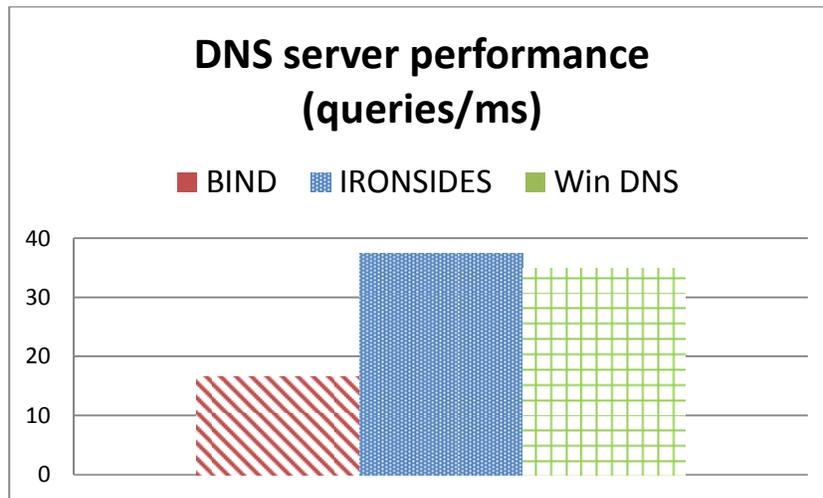


Fig. 4. Comparison of DNS software (queries per millisecond, averaged)

The most important result of our experiment is that IRONSIDES not only has better security properties than the two most popular DNS servers, but outperforms them as well. On a Windows machine, IRONSIDES is 7% faster than Windows DNS and more than twice as fast as BIND. Given IRONSIDES superior security posture, we find these results significant. They show that one need not sacrifice security for performance in software design.

In fact, it should not be that surprising that there are at least some instances in which the use of formal methods can improve performance. Data flow analysis, for example, can identify redundant or ineffective statements that generate unnecessary

code. Code that has been proven exception-free no longer needs run-time bounds checking, so that code can be eliminated as well.

On the other hand, there are also cases where total reliance on formal methods negatively impacts performance. Allowing users to override formal proof requirements when appropriate is an important feature that we believe formal methods tools should continue to support. In one case, performing this type of optimization in IRONSIDES led to a 14% improvement in performance on a Windows VM. Since such overriding is optional, users in environments where manual verification of source code is deemed too risky can revert to the original, formally verified source code at some cost in performance.

IRONSIDES is invulnerable to denial of service attacks caused by badly formatted packets that raise exceptions. But terminating a server is not the only way to deny service. If the server can be thrown into an infinite loop, service is just as effectively denied. IRONSIDES is invulnerable to this form of service denial as well, because the tools employed help prove that all of its 85 loops terminate. This is accomplished by using loop invariant assertions to show that loop variables monotonically increase and have an upper bound. This is not accomplished automatically by SPARK, but with appropriate loop assertion annotations added by the programmer SPARK can assist in showing these properties to be true.

For example, consider the code below:

```

-- Amount_Trimmed prevents infinite loop
while Answer_Count=0 and Amount_Trimmed < RR_Type.WireStringType'Last
and Natural(Character'Pos(Current_Name (Current_Name'First)))/=0 and
Current_Qname_Location <= DNS_Types.QNAME_PTR_RANGE(Output_Bytes)loop
--# assert Answer_Count=0 and Amount_Trimmed>=0 and
--# Amount_Trimmed<RR_Type.WireStringType'Last
--# and Output_Bytes <= DNS_Types.Packet_Length_Range'Last and
--# Current_Qname_Location <=DNS_Types.QNAME_PTR_RANGE(Output_Bytes);
  Trim_Name(Domainname => Current_Name,
  Trimmed_Name => Trimmed_Name,
  Qname_Location => Current_Qname_Location,
  New_Qname_Location => New_Qname_Location);
  Create_Response_SOA(Start_Byte => Start_Byte,
  Domainname => Trimmed_name,
  Qname_Location => New_Qname_Location,
  Output_Packet => Output_Packet,
  Answer_Count => Answer_Count,
  Output_Bytes => Output_Bytes);
  Current_Name := Trimmed_Name;
  Current_Qname_Location := New_Qname_Location;
  Amount_Trimmed := Amount_Trimmed +
    Natural(Character'Pos(Domainname(Domainname'First))+1);
end loop;

```

Fig. 5. Using loop invariants to prove termination

SPARK annotations begin with "--#". Here the annotations are loop invariants that serve as both a postcondition for one part of the loop and as preconditions for the next. In this case the tools prove that Amount_Trimmed is at all times both non-negative and below a constant upper bound. Data flow analysis shows that Amount_Trimmed is not modified elsewhere in the loop. Given these properties and

the last line of the loop, we can conclude that Amount_Trimmed is monotonically increasing, therefore the loop terminates.

Note that without the use of this variable and the proof annotations, we could not prove loop termination. This would leave open the possibility for the other termination conditions to never be reached, something that could be exploited under the right circumstances to deny service through an infinite loop.

6 Lessons in Humility

The use of formal methods and the SPARK tools in particular produced results that were both impressive and humbling. Both the authors are experienced software engineers, having written compilers, introductory programming environments, circuit emulators, and other non-trivial software systems. In addition to over 40 years combined computer science teaching experience, we have consulted for both industry and government. Nonetheless, the formal methods tools we employed caught boundary conditions and potential problems. Some examples are shown below:

- 1) The use in a zone file of a domain name consisting of a single character:

```
--SPARK caught possible exception if length=1, modified
--by adding "length > 1 and then"
if Name(1) = '.' or Name(1) = '-' or (length > 1 and then
(Name(Length-1) = '.' or Name(Length-1) = '-')) then
  RetVal := False;
```

- 2) A resource record of length equal to the maximum line length allowed:

```
--endIdx might be the maximum value possible, so must
catch last character here. Caught by SPARK.
if Ctr = EndIdx and numSeparators <= REQ_NUM_SEPARATORS
then
```

- 3) Failure to account for erroneous input:

```
if Query_Class /= IN_CLASS then ...
elsif Query_Type = A then ...
end if;
--Forgot else to handle erroneous input! Caught by SPARK.
```

- 4) Failure to check for subscript overflow:

```
--copy name from packet to Domainname (null terminated)
while Integer(Byte) < Integer(Input_Bytes) and then
Input_Packet.Bytes(
  Byte)/=0 loop
```

```

--this could overflow Domainname array! Caught by SPARK.
  Domainname(I) := Input_Packet.Bytes(Byte);
  I := I + 1;
  Byte := Byte + 1;
end loop;
Domainname(I) := ASCII.NUL;

```

These are all problems we should and could have detected on our own, but did not. Had they gone undetected, they could have led to security holes exploitable by hackers, particularly if they had access to source code. Our experience suggests the use of formal methods and tools is an essential part of improving the security properties of software. Using experienced, security-conscious programmers is not enough.

7 Hitting the Sweet Spot

Much of the emphasis on applying formal reasoning to computer programs has focused on proofs program correctness. This has proven to be quite difficult. Correctness properties for all but the most trivial programs are extremely complex, requiring elaborate formal models and axiomatic formulations that may be more difficult to construct than the original program.

While we anticipate continued progress in the use of formal methods to prove program correctness, our results suggest that an exclusive focus on proofs of correctness may be causing researchers to miss a “sweet spot” of opportunity: Proofs of security.

On one end of the spectrum, correctness properties are useful to prove but very hard for most interesting programs: Existing tools and technology are not yet sufficiently sophisticated to complete them. On the other hand, there are properties of programs that are easy to prove (correctness of mathematical functions, small subroutines, and so forth), but are not particularly interesting or important. Security properties fall into that middle ground of things that are both important to prove and provable with existing technology.

With the help of SPARK and the use of Ada, for example, we can formally prove the following security properties of the IRONSIDES DNS server:

- 1) No classic buffer overflow
- 2) No incorrect calculation of buffer size
- 3) No improper initialization
- 4) No ineffective statements
- 5) No integer overflow/wraparound
- 6) No information leakage
- 7) All input validated

- 8) No allocation w/o limits (no resource exhaustion)
- 9) No improper array indexing
- 10) No null pointer dereferencing
- 11) No expired pointer dereferencing (use after free)
- 12) No type confusion
- 13) No race conditions
- 14) No incorrect conversions
- 15) No uncontrolled format strings
- 16) All loops guaranteed to terminate

Problems with all of the above have so vexed BIND that the US Defense Advanced Research Projects Agency is funding a program to crowd source it and other important software to achieve formal verification of security properties [13]. By contrast, because IRONSIDES is written in Ada, a language designed from the beginning with software engineering principles in mind, and because a commercially backed tool is available for formal analysis of Ada programs, we are able to achieve provably exception-free code despite being only two academic researchers employed at an undergraduate university.

8 Conclusions and Future Work

Our work indicates that the theory and practice of formal methods has progressed considerably in the past few years, to the point where formal verification of certain desirable properties of software is now achievable at relatively little additional cost. Within less than a year, two academics whose primary duties are teaching were nonetheless able to produce a verifiably exception-free version of DNS. We did this despite having no prior familiarity with SPARK or indeed any formal language tools from industry.

While overriding the requirements for explicit storage initialization does indeed permit software engineers to trade security for performance, our results show that in general no such tradeoff is required. The IRONSIDES authoritative server runs significantly faster than either BIND or Windows DNS, and does so on a Windows “home court” VM running Windows Server 2008.

IRONSIDES is in the public domain, and is distributed free of charge at <http://ironsides.martincarlisle.com>. Currently development focuses on the IRONSIDES recursive service. Future work could include testing under other operating systems, testing under actual network loading, online zone signing, GUI and web interfaces, and other more advanced features. Other implementations of Internet protocols that suffer from security flaws could also benefit from the approach described here.

This work was funded by the US Defense Advanced Research Projects Agency, whose support is gratefully acknowledged. We thank AdaCore Technologies and

Altran Praxis for providing technical support on using their tools. We also wish to thank the USAFA Department of Computer Science, the Academy's Director of Research, and the Academy Center for Cyberspace Research.

References

1. Internet Systems Consortium, <http://www.isc.org>
2. Barnes, J.: High Integrity Software: The SPARK Approach to Safety and Security. Addison-Wesley Publishing, 0-321-13616-0, © 2003.
3. <http://www.adacore.com/2010/08/16/spark-skein/>
4. Barnes, J. et al: Engineering the Tokeneer Enclave Protection Software. In: 1st IEEE Symposium on Secure Software Engineering (2006).
5. Woodcock, J. et al.: Formal methods: Practice and experience. ACM Comput. Surv. 41, 4, Article 19 (October 2009), 36 pages.
6. Ricky E. Sward, Martin C. Carlisle, Barry S. Fagin, David S. Gibson: The case for Ada at the USAF Academy. In: ACM SIGAda International Conference on Ada pp 68-70 (2003).
7. Martin Carlisle and Barry Fagin: IRONSIDES: DNS With No Single-Packet Denial of Service or Remote Code Execution Vulnerabilities, Proceedings of IEEE GLOBECOM 2012, Anaheim CA.
8. DNSSEC – The DNS Security Extensions, <http://www.dnssec.net/>
9. <http://www.adacore.com/sparkpro/language-toolsuite/>
10. <http://alt-ergo.lri.fr>
11. Nominum, Inc. DNS measurement tools. Available online at <http://www.nominum.com/support/measurement-tools/>
12. Comparison of DNS Server Software, http://en.wikipedia.org/wiki/Comparison_of_DNS_server_software
13. [http://www.darpa.mil/Our_Work/I2O/Programs/Crowd_Sourced_Formal_Verification_\(CSFV\).aspx](http://www.darpa.mil/Our_Work/I2O/Programs/Crowd_Sourced_Formal_Verification_(CSFV).aspx)