Limiting Replay Vulnerabilities in DNSSEC

He Yan*, Eric Osterweil†, Jon Hajdu*, Jonas Acres* and Dan Massey*
* Colorado State University
Email: yanhe@cs.colostate.edu, hajdu3@hotmail.com, acresjonas@gmail.com, massey@cs.colostate.edu
† UCLA
Email: eoster@cs.ucla.edu

***Abstract*— The DNS Security Extensions (DNSSEC) added public key cryptography to the DNS, but problems remain in selecting signature lifetimes. A zone's master server distributes signatures to secondary servers. The signatures lifetimes should be long so that a secondary server can still operate if the master fails. However, DNSSEC lacks revocation. Signed data can be replayed until the signature expires and thus zones should select a short signature lifetime. Operators must choose between reduced robustness or long replay vulnerability windows.**

This paper introduces a revised DNSSEC signature that allows secondary servers to operate even if the master has failed while simultaneously limiting replay windows to twice the TTL. Each secondary server constructs a hash chain and relays the hash chain anchor to the master server. The signature produced by the master server ensures the authenticity of the hash anchor and the DNS data. A secondary server includes both the signature and a hash chain value used by resolvers to limit signature replay. Our implementation shows the added costs are minimal compared to DNSSEC and ensures robustness against long-term master server failures. At the same time, we limit replay to twice the record TTL value.

I. Introduction

The DNS Security Extensions (DNSSEC)[1], [3], [2] add public key cryptography to the DNS, but DNSSEC does not include a revocation mechanism. Instead, DNSSEC relies on explicitly specified signature lifetimes to limit the replay of old data. All signatures include a fixed expiration time and an attacker can continue to replay the old data until the expiration time. DNS administrators can limit replay vulnerabilities by selecting short signature lifetimes. Ideally, signatures lifetimes would be roughly equal to the TTL associated with a data record, in order to limit replay attacks.

Unfortunately, the signatures lifetimes must also meet other DNS operational requirements, such as allowing a secondary server to continue operating if the master fails. DNS master servers specify a "zone expiration" time during which slave servers must serve zone data without being refreshed by the master. For example, the master may fail but slaves should still ensure the zone data is available for up to the "zone expiration" time. DNS administrators must choose signature lifetimes that are greater than the zone expiration time. Limiting replays requires signature lifetimes be roughly equal to the TTL values, typically in minutes, but signature lifetimes must be larger than the zone expiration time, typically in weeks. Operators must choose between short signatures with reduced DNS robustness or long signatures with long replay vulnerability windows.

This paper introduces a revised DNSSEC signature that preserves robustness while simultaneously limiting replay windows to twice the TTL. Our approach enhances each existing public key based signature with a set of Hash Chains [10] that are each specific to a single authoritative server. The hash chain is used to determine if the data is still fresh. To evaluate this approach, we implemented it in BIND [8], one the of the most widely used DNS packages. Our evaluation shows (using actual DNSSEC data) that this approach reduces the average vulnerability period by a factor of over 28 while only incurring up to 4% CPU overhead.

The remainder of this paper is organized as follows: Section II gives a brief introduction of DNSSEC. Section III defines the requirements and shows why large signature lifetimes are needed. Section IV presents our solution and Section V evaluates the results. Section VI discusses the related work and concludes the paper.

II. Background

The Domain Name System (DNS) [11] maps hostnames, such as `www.colostate.edu`, to IP addresses and provides a wide range of other mapping services ranging from email to geographic location. Virtually every Internet application relies on some form of DNS data. DNS data is stored in Resource Records (RRs), and each RR has an associated name, class, and type. For example, an IPv4 address for $www.colostate.edu$ is stored in an RR with name $www.colostate.edu$, class IN (Internet), and type A (IPv4 address). The set of all resource records associated with the same name, class, and type is called a Resource Record Set (RRset) and resolvers issue queries for RRsets. A browser seeking the IPv4 address of $www.colostate.edu$ will query for the RRset $\langle$`www.colostate.edu, IN, A`$\rangle$. Note that the smallest unit that can be requested in a query is an RRset. If $www.colostate.edu$ has multiple IP addresses, the resulting RRset will have multiple RRs (one for each IP address) and all RRs in the set will be returned in a response. All DNS actions, including cryptographic signatures, apply to RRsets.

Every RRset belongs to a specific zone and is stored at the nameservers of that zone. DNS zones are organized in a tree structure. At the top of the tree, the root zone delegates authority to *top level domains* like `com.`, `net.`,

This material is based upon work supported by by National Science Fundation(NSF) under Award 0524172. Any opinions, findings and conclusions or recommendations expressed in this material are those of theauthorsand do not necessarily reect the views of NSF.

`org.`, and `edu.`. The zone `com.` then delegates authority to create `google.com.`, `edu.` delegates authority to create `colostate.edu.`, and so forth. In the resulting DNS tree structure, each node corresponds to a *zone*. Each zone is owned by a single administrative authority and is served by multiple *authoritative nameservers* that provide name resolution services for all the RRsets in the zone. For example, the RRset ⟨`www.colostate.edu, IN, A`⟩ belongs to the colostate.edu zone and is stored in the colostate.edu authoritative nameservers. For robustness, it is recommended that zone's authoritative nameservers are widely distributed[6].

Security was not a primary objective when the DNS was designed in mid 80's and a number of well known vulnerabilities have been identified [5], [4]. DNSSEC [1], [3], [2] adds cryptographic authentication to the existing DNS. To prove that data in a DNS reply is authentic, each zone creates public/private key pairs and then uses the private portions to sign its RRset. The zone's public keys are stored in a new type of RR called `DNSKEY`, and all the signatures are stored in another new type of RR called `RRSIG`. In response to a query, an authoritative server returns both the requested RRset and its associated `RRSIGs`. A resolver that has learned the `DNSKEY(s)` of the requested zone can verify the *origin authenticity* and integrity of the reply data. To resist replay attacks, each signature carries a definitive expiration time.

In order to authenticate the `DNSKEY` for a given zone, say `www.colostate.edu`, the resolver needs to construct a *chain of trust* that follows the DNS hierarchy. In the ideal case, the public key of the DNS root zone would be obtained offline in a secure way and stored at the resolver, so that the resolver can use it to authenticate the public key of *edu.*; the public key of *edu.* would then be used to authenticate the public key of *colostate.edu.*. There is no revocation mechanism in DNSSEC, but each signature carries a definitive expiration time. An attacker can successfully replay all RRset and signatures in the chain of trust until one of the signatures has expired.

III. DNSSEC Requirements and Open Issues

For a typical DNS zone, there is a single master server M and multiple slave servers S_1, S_2, ... ,S_n. The zone's RRsets R_1, R_2, ... ,R_k and their corresponding signatures (RRSIGs) are generated by the zone administrator. Each signature has a lifetime that indicates how long the signature will be valid. The master M is responsible for managing all data and signatures. The slave servers provide replicas of the zone data, help distribute the query load, and ensure the zone data remains available even if the master server has failed. Slave servers retrieve the RRsets and their corresponding signatures using a combination of polling the master server to detect changes[11] and/or listening for notification messages from the master server[13]. In this paper, t denotes absolute times and T denotes relative times.

Threat Model: We suppose that in addition to the valid slave servers, there are some unknown number of attackers that attempt to imitate slave servers. Some slave servers may have also been compromised by an attacker. Let A_1, ... ,A_m denote the machines controlled by the attacker. As seen in Figure 1, when a client requests an RRset R_i, the request may be directed to either M, S_i, or A_i.

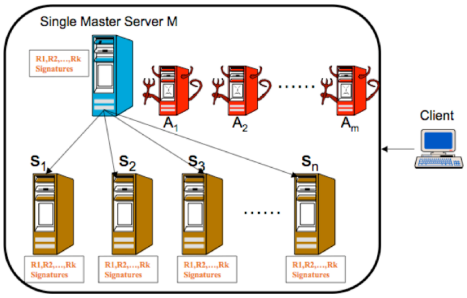

Fig. 1. DNSSEC Threat Model

Requirements:

1) RRset Authenticity Despite Untrusted Slaves: Only M can modify the zone's RRsets. Given a zone's public key, a resolver must be able authenticate any RRset associated with the zone. In other words, a resolver that has the zone's public key must be able to detect if any slave S_i or attacker A_i has modified the data.
2) Zone Expiration: If S_i downloads an RRset R_j from M at time t, then S_i *must not* be able to serve this RRset to clients after time $t+T_{expire}$. This the zone expiration time and is distinct from the signature expiration time.
3) Zone Survivability: If S_i downloads an RRset R_j from M at time t, then S_i *must* be able to serve this RRset to clients until at least time $t + T_{survive}$. In other words, if the master fails, data will be available from slaves for at least time $T_{survive}$ and at most time T_{expire}.
4) Limit Replay: If any RRset R_i is changed by M, and all S_i's have obtained the new R_i by time t, and none of the slave servers are compromised, then an attacker A_i cannot replay the old R_i after time $t + n \times TTL$, where n is a constant factor.
5) Slave Identification: If a resolver has obtained record R_i, some third party can provably identify the slave server that reported R_i. It is specially useful to identify compromised slaver servers.

Among these five requirements, the first three have been previously proposed in DNS and DNSSEC and the last two are new. Prior to DNSSEC, slave servers were implicitly trusted not to modify data. DNSSEC was designed primarily to handle requirement 1 (RRset Authenticity Despite Untrusted Slaves). The signature associated with an RRset can be used to authenticate the RRset and no one can modify it without invalidating the signature. Thus, any attempt by a slave server to modify the data would invalidate the signature. DNS best practices[6] encourage slave servers to be widely distributed and operated by varying administrative groups. Partly because of their distributed nature, slave servers do not hold the private key needed to generate new signatures.

Requirement 2 (Zone Expiration) limits how long S_i is able to serve RRsets to resolvers. For example, it may be the case

that the master is still working and is modifying the zone data, but the slave has lost connectivity to the master and thus cannot receive any updates. All RRsets held by the slave will be discarded if the slave has been unable to communicate with the master server for some zone expiration time, denoted T_{expire}. The zone expiration time is specified in the DNS SOA[11] record and intended to ensure a disconnected slave will eventually stop serving obsolete data. DNS implicitly trusts slaves to discard data after the zone expiration period. DNSSEC can be used to enforce this by selecting a signature lifetime $L \leq T_{expire}$. Even if the slave refuses to discard the zone data, the signatures will expire and resolvers will ignore the data .

Requirement 3 (Zone Survivability) indicates S_i must be able to serve RRsets for a relatively long time even if the master server M fails. Consider the DNS scenario (without DNSSEC) in which S_i obtains all the RRsets from M at time t and then M fails. S_i must be able to serve these RRsets to resolvers at least until time $t+T_{survive}$. Without DNSSEC, the RRsets held at slave S_i do not expire and slaves are implicitly trusted to serve data for exactly $T_{survive} = T_{expire} =$ zone expiration time.

In DNSSEC, the signature lifetimes add explicit T_{expire} to each RRset. And the zone administrator must select a signing interval and signature lifetimes such that the signatures present at slave are valid for at least time $T_{survive}$. For example, suppose RRsets are signed at the first of each month and have a lifetime 30 days. If the master fails on June 29, no slaves will receive updated signatures. All data stored at slaves will be invalid on July 1st and the zone will be unreachable. A longer signature lifetime (e.g. expires after two months) could extend the zone's survivability in this case. Similarly, more frequent signing (e.g. resigning every 15 days) could also extend the zone's survivability. Overall DNSSEC can meet the first three requirements, provided that the zone administrator picked an appropriate signature lifetime L and signing interval, denoted by Int in this paper.

For example, suppose a zone administrator wants to tolerate a master server failure of 20 days ($T_{survive} = 20$ days) and requires slave servers to stop serving data if the master has been unavailable for 30 days ($T_{expire} = 30$ days). In this case, the administrator can pick 28 days as the signature lifetime ($L = 28$ days) and generate new RRSIGs every 8 days ($Int =$ 8 day) as shown in Figure 2. In general, the zone administrator can meet requirement 3 by selecting $T_{survive} \leq L - Int$.

The use of signatures clearly enforces requirement 1. The RRsets held at any slave server expire after 28 days and thus a slave server can serve RRsets for at most $L = 28 < T_{expire} =$ 30 days, satisfying requirement 2. Requirement 3 is more complex and depends on both L and Int. Let t denote the time slave S_i last retrieved signatures from the master. In the worst case, slave S_i retrieves signatures immediately before the next $Int = 8$ signing update and then all communication with the master fails. The earliest expiration date for the signatures at S_i is $t - Int + L = t + 20$ days. In other words, slave S_i can continue to serve these records for $20 = T_{survive}$ days, meeting requirement 3. Note that requirement 2 depends only on the signature lifetime L while 3 depends on both the signature lifetime L and signing interval Int.

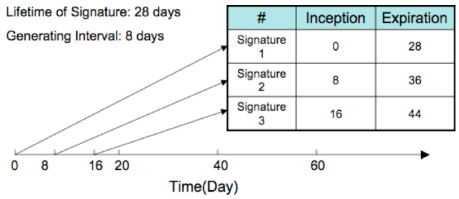

Fig. 2. Example DNSSEC Signature Lifetimes

A. Open Challenges

Requirement 4 (Limited Replay) remains an open challenge. DNS makes extensive use of caching and thus old copies of RRsets may remain in caches for up to some Time to Live (TTL) period. However, an attacker A_i can request an RRset R_i from M or any of the S_i and does not need to respect the TTL value. Once A_i has an RRset and its signature, she can simply replay them until the signature expires. Even if the RRset and its signature have been withdrawn from M and all S_i, an A_i can imitate an authoritative server and continue to replay the old RRset until the signature has expired. One would like to limit replays to a value on the order of the TTL. To meet requirement 4, one could choose a signature lifetime $L \leq t+n \times TTL$ where n is a constant factor. Unfortunately, TTLs are often on the order of seconds or minutes while $T_{survive}$ is often measured in weeks. One cannot select a signature lifetime L such that $T_{survive} \leq L \leq T_{expire}$ and $L \leq TTL$ when $TTL \ll T_{survive}$.

Our approach will also meet a final new requirement (Slave Identification) not previously available in DNS or DNSSEC. One would like to prove the identity of the slave server that originally reported any RRset R_i. This new feature would add a much finer degree of origin authenticity and can help in debugging or identifying compromised slave servers.

IV. Hash Chain Based Expiration

Our approach uses the standard DNSSEC signature lifetimes to meet the first three requirements of Section III. Zone administrators determine the desired values for $T_{survive}$ and T_{expire} and then select a signature lifetime L and signing interval Int such that $L \leq T_{expire}$ and $L - Int \geq T_{survive}$.

In our approach, the signature expiration is only an *upper bound* on the RRset lifetime. To enforce a tighter bound, each authoritative server associates a hash chain with each RRset and releases a new value every TTL. Once the resolver has verified the signature, it checks that the record has an appropriate hash chain value. RRsets that lack an appropriate hash chain value are discarded as invalid. A hash chain value is valid for at most $2 * TTL$, meeting requirement 4 (Limited Replay) and also identifies which server announced the signature, meeting requirement 5 (Slave Identification).

Adding Hash Chains to DNSSEC: Hash chains (also called one-way chains) were first proposed by Lamport [10].

A hash chain of length n is constructed by applying a one-way hash function H recursively with an initial seed h_0. Once values h_0 to h_n have been constructed, the values are released in the opposite order starting from h_n. Given h_i, it is easy to verify whether h_{i-1} is the next value by checking if H(h_{i-1}) equals h_i, but it is computationally infeasible to generate h_{i-1}.

In our scenario, a slave server S_i generates one hash chain $(h_0, ..., hn)$ for each RRset. Server S_i distributes the *hash anchor* h_n to the zone administrator. The zone administrator adds the hash anchor h_n to the signed data. If the zone has n slave servers and one master sever, $n+1$ hash anchors are used, one for each authoritative server.

Adding this data requires a change to the RRSIG record format. Our revised RRSIG-H record format is shown in Figure 3) and contains all fields from the previous RRSIG format. Four additional fields (shaded in the Figure) are also included. The *Hash Algorithm* identifies the hash algorithm (e.g. SHA-1, MD5) used in creating the chain. The *Number of Hash Anchors* indicates how many hash anchors are included in the record and the *Hash Anchor List* lists the hash anchors. For example, a zone with one master and four slave servers may have five hash anchors. If SHA-1 is used as the hash algorithm, the total length of this field would be $20 \times 5 = 100$ bytes. Finally, the *Current Hash Value* lists the hash value h_i inserted by a particular server. Since this field will vary, it is not covered by the signature.

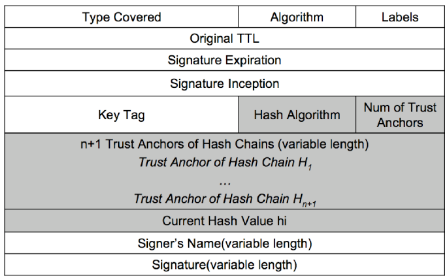

Fig. 3. RRSIG-H Format

A server S_i obtains the RRset data and RRSIG-H from the master server M and then fills in the *Current Hash Value*. If less than one TTL period has elapsed since the signature inception date, h_n is included as the *Current Hash Value*. If more than one, but less than two TTL periods have elapsed since the signature inception date, h_{n-1} is included as the *Current Hash Value*. In general, the slave starts with value h_n and then releases the next hash value every TTL seconds. Although the hash anchor h_n is public, only S_i knows the hash chain $(h_1, .., h_n)$ and thus only S_i can determine which hash value will be released during the next TTL period.

Verifying Data Using Hash Chains: A resolver receives the RRset and the RRSIG-H record and discards the data if the signature fails or has expired. The resolver next checks whether the *Current Hash Value* h_i is correct. The resolver does not know the full hash chain, but does know the signature inception date and the TTL from the RRSIG-H record[1]. A valid server should have released k hash values where:

$$k = (\lfloor \frac{t_{current} - t_{inception}}{TTL} \rfloor) \quad (1)$$

Applying the hash function k times to the *Current Hash Value* should result in a hash value that matches one of the hash anchors listed in the *Hash Anchor List*. If there is a match, the *Current Hash Value* is valid and was produced within the last TTL seconds by one of the authoritative servers.

One final issue is that DNS makes extensive use of caching and the response may come from a cache rather than directly from the authoritative server. If the cache is acting correctly (e.g. not falsely replaying data), the RRset may be at most one TTL value old. In other words, the *Current Hash Value* may be either $h_{n-(k-1)}$ or h_{n-k}. By starting with the *Current Hash Value* and applying the hash function either k or $k-1$ times, the resulting value should match one of the hash anchors listed in the *Hash Anchor List*. Any other value is discarded.

An attacker that attempts to replay a response after more than $2 \times TTL$ periods will fail the above check. If the data changes, the authoritative servers will not issue responses using the next hash chain values in the sequence and the attacker cannot guess these values. Our approach thus limits the replay of response to $2 \times TTL$, meeting Requirement 4.

Furthermore, the server that announced the RRSIG-H has been uniquely identified. The *Current Hash Value* matched one of the hash anchors and only the server using that hash chain could have produced that RRSIG-H. In the event a slave server is malfunctioning, a zone administrator can uniquely identify the server that sent this message, meeting Requirement 5.

Example: Suppose a resolver receives the RRSIG-H record shown in Figure 4 at time 1212339740 (Jun-1st-2008 17:02:20). The signature inception date $t_{inception}$ = 1212332400 (Jun-1st-2008 15:00:00) and it expires 1214838000 (Jun-30th-2008 15:00:00). The original TTL is only 3600 seconds and thus should be discarded by Jun-1st-2008 18:02:20. If the signature is valid, an attacker could replay this RRSIG and corresponding RRset for up to 29 days. This replay is possible under standard DNSSEC even if the RRset changed and all slave servers learned of this change.

To check the correctness of the *Current Hash Value* (052d86bbeff9478edf34166901081b94), the resolver needs to calculate $k = \lfloor \frac{1212339740-1212332400}{3600} \rfloor = 2$. That means the resolver should hash the *Current Hash Value* either $k = 2$ or $k - 1 = 1$ times. Applying the hash function two times results in a match with the third trust anchor (9bdf40befda56aded47d3d00bffed831). By doing this 2-steps check, the resolver can make sure the data in the reply message was originated in M and sent from one of five authoritative servers in past $2 \times TTL = 7200$ seconds.

This *Current Hash Value* is only valid during the $2 \times TTL = 7200$ seconds from Jun-1st-2008 17:00:00 to Jun-1st-2008 19:00:00. An attacker cannot replay this record after that

[1]These values are covered by the signature and thus could not have been modified.

Type Covered: A	RSA/SHA1	Labels: 3
Original TTL: 3600s		
Signature Inception: 1212332400 (Jun-1st-2008 15:00:00)		
Signature Expiration: 1214838000 (Jun-30th-2008 15:00:00)		
Key Tag: 28113	SHA1	5
5 Trust Anchors Trust Anchor 1: 2e93a252a954f23912547d1e8a3b5ed Trust Anchor 2: 8e547d9586f6a73f73fbac0435ed7695 **Trust Anchor 3: 9bdf40befda56aded47d3d00bffed8312** Trust Anchor 4: 2547d1e8a3b5ed6e1bfd709782d345f5 Trust Anchor 5: 6951218fb7d0c8d788a309d75d4a345f		
Current Hash Value: 052d86bbeff9478edf34166901081b94		
Signer's Name: netsec.colostate.edu		
Signature: 07e547d9586f6a73f73fbac0435ed76951218fb7d0c8d788a309d785436bbb642 e93a252a954f23912547d1e8a3b5ed6e1bfd7097821233fa0538f3db854fee6		

Fig. 4. Example RRSIG-H From A Response

time and cannot generate *Current Hash Values* for future times. Using the hash chain, the resolver limit replays to only $2 \times TTL = 7200$ seconds rather than 29 days.

V. Evaluation

Based on our data from SecSpider [7], there are currently 10,428 zones using DNSSEC. From this set we observe that the RRSIG lifetimes spans from 60 seconds to 5477.2 days. After discarding some zones that are clearly test deployments, we found the median signature lifetime was 30 days (720 hours). The TTLs associated with these zones span from 0 seconds to 168 hours, with an average of 12.75 hours. Thus, we can compute the reduction in the vulnerability by comparing the average value of DNSSEC's current vulnerability period against our approach: $\frac{R^{lt}}{T} = \frac{720}{2 \times 12.75} = 28$.

Our approach assumed a server S_i (or M) had a unique hash chain to associate with each RRset. Hash chains can be computed in advance. Since a new hash value is used once every TTL, the length of the hash chain should be $n = \frac{L}{TTL}$ where L is the signature lifetime. The hash anchor h_n must be securely distributed to the zone signer before the signature is created (typically once per month based on current deployment trends). Only the authoritative server has access to the full hash chain $(h_1, ..., h_n)$. An authoritative server (or its proxy) can generate multiple hash chains and pre-distribute the hash anchors to the zone signer. Once the hash values have been generated and the RRSIG-H records created, authoritative servers must increment the Current Hash Value every TTL seconds. Since the hash chains were pre-generated, this amounts a single copy operation at the slave server.

We define the cost of our system with respect to computation, communication, and storage. To evaluate the costs, we modified the `named` server, `dnssec-signzone` signer, and `dig` resolver from BIND 9.4.1. Our modified `dnssec-signzone` is used to sign a DNS zone and produce signatures in our new RRSIG-H format. Our modified `named` serves the zone data and RRSIG-H records and our modified `dig` is able to verify the new RRSIG-H.

Computation Cost: Our measurements of the computation costs were done on a uniprocessor 1 GHz AMD64 running Ubuntu Linux. We used SHA-1 as our hash function and 30 days as the signature lifetime. The additional computation cost at the *slave servers* is related to generating the hash chains. Thus, we focus our analysis on i) how many hash chains must be created and ii) how long each hash chain takes to generate. The number of hash chains needed is derived from both the number of RRsets in a zone, and how often they change. If a zone has s RRsets, and the average number of changes during the lifetime of signature is x, then a slave server needs to generate $x+s$ hash chains every signature lifetime. The length of hash chain must be $\frac{lifetime}{TTL}$. We call s the *zone size* and x the *zone dynamics*. The amount of time needed to generate a hash chain is the runtime of the actual hash function times the length of the chain.

We varied the TTL, zone size, and zone dynamics in order to quantify their effects on computation cost. Our results confirmed that the computation cost increases linearly with zone size and zone dynamics. For example, the computation cost was roughly 6,500 milliseconds for a zone size and dynamics of $s + x = 4,000$, signature lifetime $L = 30$ days, and TTL of one hour. This computation must take place once every signature lifetime. But with a typical signature lifetime of 30 days, this requires only 0.000251% percent of server CPU time. Figure 5 shows the percentage of server CPU time over varying TTL and zone size values. Note even a large zone with 60,000,000 RRsets (e.g. the "com" zone) requires less than 4% of the server CPU time on our generic unix box.

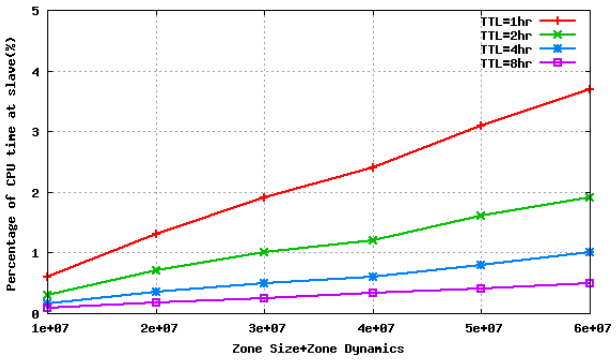

Fig. 5. Percentage of CPU Time at Slave Server

In our approach, several hash anchors are added to RRSIG-Hs (one for each authoritative server). To evaluate the impact on the signer, we created 20 unsigned DNS zone files with varying numbers of RRsets (500 to 10000). For each zone file we: i) ran the original `dnssec-signzone` to sign the zone file with RSA/SHA-1 20 times and calculated the average run time (computation cost), and ii) then ran the modified `dnssec-signzone` to sign the zone file with RSA/SHA-1 20 times and collected the average run time. Figure 6 shows the difference between original `dnssec-signzone` and modified `dnssec-signzone` in terms of computation cost.

In our approach, the resolver verifies the signature as in standard DNSSEC and then must do an additional check of the

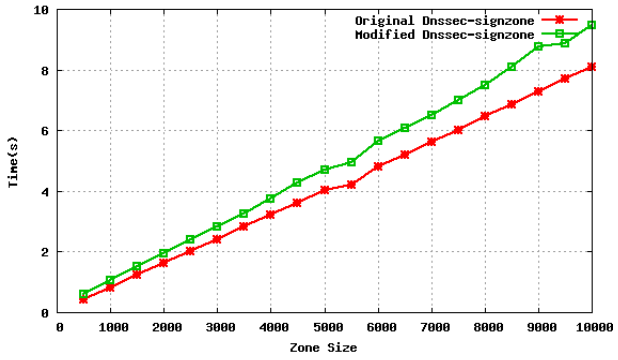

Fig. 6. Computation Cost Comparison Between Original dnssec-signzone and Modified dnssec-signzone

Current Hash Value in the RRSIG-H[2]. Using the signed zones generated in the step above, we randomly selected 100 RRsets. For each RRset, we calculated the average computation cost of standard DNSSEC by querying and validating the RRset and its RRSIG 10,000 times using dig. Similarly, we calculated the average computation cost of our approach by querying and validating the RRset and its RRSIG-H 10,000 times using a modified dig. Our measurements show the difference between original and modified `dig` is constantly about 10%.

Communication Cost: Our approach requires slightly more data to be transmitted in its RRSIG-H records compared to today's RRSIG. The additional communication cost comes from two new fields: i) a list of trust anchors of hash chains (the number depends on the number of slave servers and the hash function), and ii) the current hash value. Thus the communication cost can be expressed by the simple formula $c = (n+1) \times u$ Where c denotes the total additional communication cost, n is the number of servers including slave and master, and u is the size of a single hash value. If there is only one master and one slave server, the communication cost is $c = (2+1) * 20bytes = 60bytes$. For 13 servers, the cost would be $c = (13+1) * 20bytes = 480bytes$ per RRSIG-H record. DNS limits the number of servers to 13[3].

Storage Cost: The additional storage cost of our approach at the resolver is $r \times c$, where r is the number RRsets that are cached and c is the number of bytes added in RRSIG-H (taken from above). In addition to the increased RRSIG-H size, the additional storage cost at an authoritative server is the space needed to store the hash chain for each RRset. Each server needs to generate $x + s$ hash chains every lifetime, and this is also the number of hash chains to be stored in server. As a result the max total additional storage cost is $s \times c + (x+s) \times j$, where j is the size of a single hash chain. In this case, $j = lifetime \times \frac{k}{TTL}$, where k is the size of a single hash value. For example, there are 3 slave servers, 2000 RRsets in the zone, 100 changes every lifetime. Lifetime is 30 days, TTL is 6 hours and SHA-1(20bytes) is used. In this example the total additional storage cost at slave server would be 2000*100+(100+2000)*720*20/6 = 5240kbytes. This storage cost should be acceptable for most of zones.

[2]The overhead imposed by this varies with the hash function used. For example, SHA-1 incurs a higher cost than MD5.

[3]The server limit is 13 IPv4 addresses. Anycast can be used to add more servers

VI. Conclusions and Related Work

This paper presented an approach for dramatically reducing DNSSEC replay vulnerabilities while preserving DNS robustness. Our approach enhanced each existing DNSSEC RRSIG records with a set of Hash Chains [10] that were each specific to a single authoritative server. The hash chain is used to determine if the data is still fresh and limits replay to $2 \times TTL$ while still allowing slave servers to serve the data for an extended period, T_{serve}, even if the master server fails. We implemented this approach in BIND [8] and our evaluation shows this approach reduced the average vulnerability period by a factor of over 28 while only incurring up to only 4% CPU overhead.

The operational community has developed best practices for generating signatures[9], but the focus has been on regenerating signatures so that they don't expire unexpectedly at cache. Toolsets have also been developed to help manage the signing process[8]. Monitoring projects are tracking operational use [12] and show the need to reduce signature lifetimes. Our work identifies the fundamental trade-off between robustness and replays in current DNSSEC and is the first to simultaneously address both the zone robustness and replay attacks.

Hash chains (also called one-way chains) were first proposed by Lamport [10] for one-time password, but have since been applied a number of settings. While our application of hash chains is specific to DNSSEC, our approach for combining hash chains with signatures could be applied to other protocols and settings that share a concept of trusted signer and multiple untrusted data sources.

References

[1] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirement. RFC 4033, Mar 2005.
[2] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Protocol Modifications for the DNS Security Extensions. RFC 4035, Mar 2005.
[3] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Resource Records for the DNS Security Extensions. RFC 4034, Mar 2005.
[4] D. Atkins and R. Austein. Threat Analysis of the DNS. RFC 3833, 2004.
[5] S. Bellovin. Using the DNS for System Break-Ins. In *Usenix Security Symposium*, 1995.
[6] R. Elz, R. Bush, S. Bradner, and M. Patton. Selection and Operation of Secondary DNS Servers. RFC 2182 (Best Current Practice), July 1997.
[7] Internet Research Lab, UCLA CS Department. The SecSpider DNSSEC Monitoring Project. http://secspider.cs.ucla.edu/.
[8] Internet Systems Consortium. Berkeley Internet Name Domain (BIND). http://www.isc.org/index.pl?/sw/bind/.
[9] O. Kolkman and R. Gieben. DNSSEC Operational Practices. RFC 4641 (Informational), Sept. 2006.
[10] L. Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11):770–772, 1981.
[11] P. Mockapetris. Domain Names: Concepts and Facilities. RFC 1034, 1987.
[12] E. Osterweil, D. Massey, and L. Zhang. Observations from the DNSSEC Deployment. *Secure Network Protocols, 2007. NPSec 2007. 3rd IEEE Workshop on*, pages 1–6, 2007.
[13] P. Vixie. A Mechanism for Prompt Notification of Zone Changes (DNS NOTIFY). RFC 1996 (Proposed Standard), Aug. 1996.