Digital Object Identifier: 10.1109/NCA.2014.27

URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6924218

GPU-based NSEC3 Hash Breaking

Matthäus Wander, Lorenz Schwittmann, Christopher Boelmann, Torben Weis

University of Duisburg-Essen
Duisburg, Germany

***Abstract*—When a client queries for a non-existent name in the Domain Name System (DNS), the server responds with a negative answer. With the DNS Security Extensions (DNSSEC), the server can either use NSEC or NSEC3 for authenticated negative answers. NSEC3 claims to protect DNSSEC servers against domain enumeration, but incurs significant CPU and bandwidth overhead. Thus, DNSSEC server admins must choose between more efficiency (NSEC) or privacy (NSEC3). We present a GPU-based attack on NSEC3 that revealed 64% of all DNSSEC names in the `com` domain in 4.5 days. This attack shows that the NSEC3 privacy promises are weak and thus DNSSEC server admins must carefully decide whether the limited privacy is worth the overhead. Furthermore, we show that an increase of the cryptographic strength of NSEC3 puts attackers at an advantage, since the cost of an attack does not rise faster than the costs incurred on the DNSSEC server.**

I. Introduction

Authenticated denial of existence is a technique used by the Domain Name System Security Extensions (DNSSEC) to prove the non-existence of a queried domain name. Negative responses in the Domain Name System (DNS) are indicated by a non-existent domain error. DNSSEC enables cryptographic validation of signed DNS zone data. However, DNSSEC does not sign the response but instead the resource records (data sets) inside the response. This subtle difference has consequences for negative responses, as they cannot be signed directly by DNSSEC due to the absence of an existing resource record.

To cope with negative responses, DNSSEC introduces a new record type called NSEC. An NSEC record proves the existence of two names in a DNS zone, and indirectly the non-existence of any names in between them in a canonical order. An example would be "`abc NSEC foo`", proving the existence of the names `abc` and `foo`, and also proving the non-existence of names in between like `bar` or `def`. The chain of NSEC records enables the enumeration of zone data: clients can retrieve a list of all existing names of a DNS zone. Several major DNS operators argue that they are obligated to hide the zone data for legal or policy reasons.[1]

The later introduced NSEC3 record type attempts to obstruct zone enumeration by using hash values of names in the non-existence proof [5]. In addition to providing an authenticated denial of existence, NSEC3 adds the privacy goal of not disclosing existing names. The privacy benefit of NSEC3 comes at a cost: because of the hashing, negative responses are larger on average and cause additional CPU load for each negative response. This can lead to a considerable overhead to name servers which send negative responses with NSEC3.

In this paper, we analyze the efficiency of GPU-based attacks against the NSEC3 privacy goal. After describing the NSEC3 fundamentals (Section II), we survey the use of NSEC3 for top-level domains (Section III). We describe an algorithm for retrieving NSEC3 hashes (Section IV) and three methods for NSEC3 hash breaking: brute-force attack, dictionary attack and Markov attack (Section V). The methods are designed and implemented to run efficiently on Graphics Processing Units (GPUs) that are used in consumer-grade graphic cards. We evaluate the methods in a case study with the `com` top-level domain (Section VII). The goal of this paper is to provide DNS operators with methods and tools for assessing whether the privacy benefit of NSEC3 is worth the server-side costs it causes.

II. Authenticated Denial of Existence

Negative responses need to be authenticated to prevent denial of service attacks on actually existing domain names. The design rationale behind NSEC and NSEC3 was to support offline signing of negative responses. Offline signing is the process of creating DNSSEC signatures a priori, before queries to the DNS zone are handled. In order to protect the private key, offline signing can be performed in a secure, private network. The resulting signatures can then be copied to public name servers, which serve the DNS zones without access to the private key.

NSEC proves the non-existence of a name n by returning an enclosing NSEC range (r_1, r_2). It holds that $r_1 < n < r_2$ under a definite canonical ordering of names. The NSEC record is signed and its correctness can be validated by DNSSEC just as for existing domain names. A name server returning a negative response needs to look up the pre-generated NSEC record and its signature in the DNS zone and return it to the client. The NSEC method is for example used by the Regional Internet Registry APNIC, which replies 40% of all queries to their reverse DNS zones with negative responses [6].

NSEC3 works in a similar way and proves the non-existence of a name n by returning an enclosing NSEC3 range $(h(r_1), h(r_2))$. h is a hash function and it applies $h(r_1) < h(n) < h(r_2)$. As shown in Fig. 1, an NSEC3

[1] Nominet explained in 2004 that denying public access to the `uk` zone file is in the best interest of their local community, and that some kind of technical enforcement is needed to protect their database right granted under European Union law [1] [2]. DENIC explained in 2004 that public access to the `de` zone file would be in conflict with Germany's Federal Data Protection Act [3] [4].

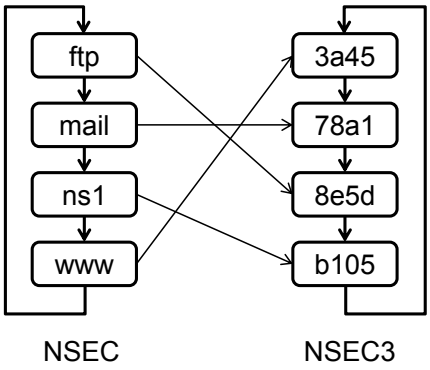

Fig. 1. NSEC composes an ordered chain of names, whereas NSEC3 composes an ordered chain of hash values of names.

chain is ordered by the resulting hash values which does not necessarily equal the order of domain names. NSEC3 requires the name server to hash the non-existent query name to find the matching NSEC3 record. Although hashing is in general very fast compared to asymmetric signing operations, it is a significant overhead for name servers with pre-generated NSEC3 signatures [7]. Furthermore, NSEC3 responses are larger than regular NSEC responses on average due to the size of hashed names being larger than most existing domain names.

A. NSEC3

The NSEC3 hash function h is defined [5] as

$$h(n, s, 0) = f(n \| s) \tag{1}$$

$$h(n, s, i) = f(h(n, s, i-1) \| s), \text{for } i > 0 \tag{2}$$

where n is the input domain name, s an arbitrary salt value, i the number of hash iterations, f an underlying hash function and $\|$ the concatenation operator. The purpose of the salt is to prevent recurring attacks with pre-computed dictionaries, e.g. rainbow tables. As noted by Bau and Mitchell [8], the salt does not slow down a one-time hash computation attack after acquisition of all NSEC3 records because the same salt value is used for all NSEC3 records of a chain. This is a different use of salt than for example in password databases, in which each password is hashed with a different salt to increase the necessary computing time for attackers.

The domain name is given as fully qualified domain name in wire format (binary representation) and is thus unique. Therefore the name `www` will result in a different hash value below `example.com` than below `example.org`, even when both zones use the same salt value.

Note that i is defined as *additional* iterations, i.e. $i = 0$ implies one hash operation and $i = 10$ implies 11 hash operations.

B. Underlying Hash Function

The security of NSEC3 is based on the preimage resistance of the underlying hash function. NSEC3 uses an algorithm field to allow for future use of other hash functions, but currently the only specified function is SHA-1. So far, there are no known preimage attacks on SHA-1. Collision attacks do not compromise the security goals of NSEC3. Collisions during DNSSEC signing will be detected by the signer, which can restart the signing process with a different salt. Collisions during runtime will be detected by the server, which will return a server error to the client. Runtime collisions can be triggered only with specially-crafted query names. Resolving such a name will lead to a validation failure at the client, which is harmless because the name is most likely not in use anyway.

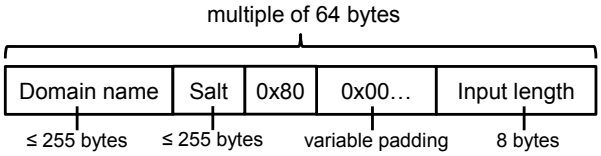

Fig. 2. SHA-1 input for the initial NSEC3 iteration.

C. SHA-1

The SHA-1 hash function compresses input data of any length $< 2^{61}$ bytes to a 20 bytes hash value [9]. Before hashing, the input data is appended with an 8 bytes data length field and 1 or more padding bytes to align the input data to a multiple of 64 bytes (Fig. 2). The data is then processed in blocks of 64 bytes, each additional block increasing the hashing workload linearly. The number of SHA-1 blocks to be computed is given as

$$b = \left\lceil \frac{len(n) + len(s) + 9}{64} \right\rceil + i \cdot \left\lceil \frac{20 + len(s) + 9}{64} \right\rceil \tag{3}$$

where the function len returns the length of a variable in bytes.

If the length of the domain name and salt are ≤ 55 bytes in wire format, they will fit into one SHA-1 block during the initial iteration. If the salt is ≤ 35 bytes, each additional iteration will also fit into one block. This applies to the majority of domain names in common use, hence $b = 1 + i$. An attacker attempting to exhaust CPU time on the server can send queries for non-existent names with the maximum length of 255 bytes [10] but cannot control the salt, hence $b = 5 + i$.

D. Minimally Covering Records

Instead of returning pre-generated NSEC records, a name server can generate minimally covering NSEC records specific to the non-existent query name, i.e. $n - 1 < n < n + 1$. The principle is compatible with both, NSEC and NSEC3, and prevents zone enumeration entirely.[2] On-the-fly generation of NSEC or NSEC3 records requires online signing with public-key cryptography, which is several orders of magnitude slower than SHA-1 hashing. It opens an attack vector for CPU exhaustion attacks and furthermore requires access to the private key on all authoritative name servers [11]. Minimally

[2]Except for the naive brute-force method of probing for existing names by sending queries to the server, which is usually considered infeasible.

TABLE I
NSEC3 PARAMETERS FOR MAJOR TOP-LEVEL DOMAINS IN MAY 2014.

TLD	i	Salt	TLD	i	Salt
at.	5	b4c3ee8e123154f8	info.	1	d399eaab
ch.	2	e7bd8151	jp.	8	f5435b71d7
cn.	10	aef123ab	net.	0	
com.	0		nl.	5	c9545f07a61d0d33
de.	15	ba5eba11	org.	1	d399eaab
eu.	1	5ca1ab1e	ru.	3	00ff
fr.	1	41b69d30	uk.	0	

covering records are thus only suitable for non-critical servers with low traffic volumes.

III. TOP-LEVEL DOMAIN SURVEY

In this section, we survey the use of NSEC3 for top-level domains. We determined the data in this section by probing the authoritative name servers of all TLDs that are listed in the IANA root zone file.[3] The probing ran four times a day from May 2013 to May 2014. Table I shows the NSEC3 parameters in use by a few major TLDs. As of May 2014, 53 TLDs use NSEC and 335 TLDs use hashed NSEC3 as authenticated denial of existence. The numbers of signed TLDs have vastly increased with the deployment of new generic TLDs, which began in October 2013. Registry operators are bound by contract with ICANN to deploy DNSSEC for new generic TLDs [12]. The choice between regular and hashed authenticated denial of existence is not mandated, but one of them is necessary for operational DNSSEC.

Eight TLDs use zero iterations and also an empty salt value, the most notable being `com`. Note that using zero iterations implies one hashing operation, as explained in Section II. Using zero iterations with NSEC3 obstructs zone enumeration to a basic degree while keeping the server-side CPU cost to the essential minimum. It furthermore allows the operator to use the opt-out feature which is the more deciding factor for large zones. With opt-out, NSEC3 records need to be created only for DNSSEC-signed domain names. Domain names without DNSSEC signatures do not need NSEC3 records. In May 2014, `com` served 113 million second-level domains but just 345,000 of them were signed.[4] The opt-out feature is only specified for NSEC3 and thus can not be used with regular NSEC. With NSEC3 and opt-out 345,000 NSEC3 records were required, but with NSEC the operator would have needed to sign and serve 113 million NSEC records, which is a substantial resource overhead. Of the 335 TLDs with NSEC3, 45% use opt-out, in particular large TLDs with many registered second-level domains.

The most frequent iteration count is $i = 1$, in use by 60% of all TLDs with NSEC3. From a conservative security perspective, choosing one iteration over zero is a reasonable choice: when a preimage attack against SHA-1 is discovered, hashing twice might provide a safety margin to make the attack ineffective. However, another possible explanation for the popularity of one iteration is that not all operators are aware of zero iterations being a valid parameter.

[3]http://www.iana.org/domains/root/files

[4]According to numbers on verisigninc.com and verisignlabs.com.

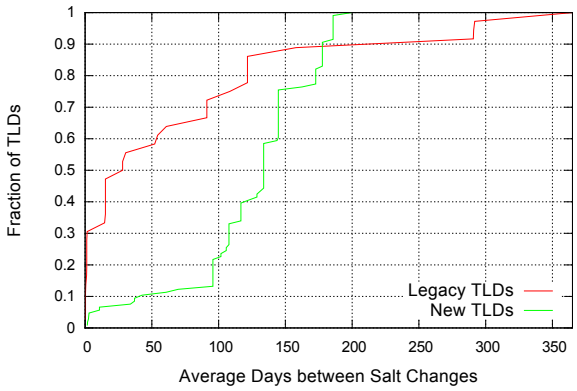

Fig. 3. CDF of average time between two salt changes.

The median salt length is 4 bytes, and there are three common usage patterns for the salt value. First, 18 TLDs (5%) do not use a salt value, i.e. the salt length is 0 bytes. Second, 175 TLDs (52%) use a salt value but did not change it during our 1 year observation period. Using a constant salt value is essentially like using no salt. The NSEC3 salt provides only protection against pre-computed rainbow table attacks if changed regularly. Lastly, the remaining 142 TLDs (42%) use the salt for its intended purpose and changed it at least once during our observation. The average interval between two salt changes is shown in Fig. 3 as a cumulative distribution function (CDF). The new generic TLDs are shown separately because they have been introduced gradually during our observation and have existed for less than 200 days. Five TLDs changed the salt more than once per day, which requires re-signing of all NSEC3 records each time. Changing the salt interrupts the NSEC3 hash crawling at a given time (cf. Section IV); an attacker subsequently has to restart from scratch.

IV. HASH CRAWLING

The first part of an NSEC3 zone enumeration attack is to retrieve all NSEC3 hash values from the server. DNS clients cannot query directly for NSEC3 records and instead have to query for random non-existing names until all NSEC3 records have been retrieved. Bernstein proposed an algorithm [13] which sends queries to the server only for those names which yield a new NSEC3 record. The approach is sketched as Algorithm 1 and comprises choosing random names and calculating their hash values. If a name is found whose hash value is not covered yet by any of the retrieved NSEC3 ranges, a DNS query will be sent to the server. The server will respond with a non-existent name error and return an NSEC3 record which can be added to the set of known NSEC3 ranges. The algorithm terminates as soon as all retrieved NSEC3 ranges form a closed circular chain.

Algorithm 1 NSEC3 hash crawling

```
1: function CRAWL(z, s, i)        ▷ zone name, salt, iterations
2:     var
3:         R ⊆ ℕ × ℕ
4:         n ∈ {'-', '.', '0'...'9', 'a'...'z'}^m
5:         x ∈ {0 ... 2^160 − 1}
6:     end var
7:     R ← {}          ▷ Crawled NSEC3 ranges, set of tuples
8:     repeat              ▷ Until all NSEC3 ranges are known
9:         repeat          ▷ Until a new range has been found
10:            n ← genRandomName() || z
11:            x ← h(n, s, i)
12:        until ∀(r_0, r_1) ∈ R : x < r_0 ∨ x > r_1
13:        R ← R ∪ queryNSEC3(n)
14:    until ∀(r_0, r_1) ∈ R : ∃a, b ∈ R : r_0 = a_1 ∧ r_1 = b_0
15:    return R
16: end function
```

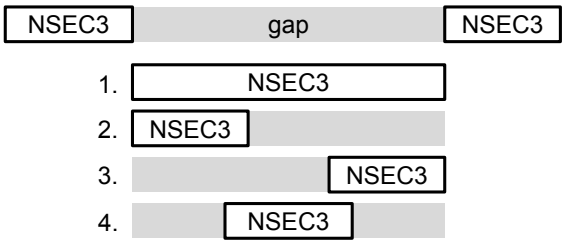

Fig. 4. Four cases for a gap to be cut by a newly retrieved NSEC3 range.

The loop in lines 9 to 12 is the computationally expensive part, which we offload to the GPU. For performance reasons, we are keeping track of aggregated gaps between NSEC3 records instead of actual NSEC3 records, i.e. the inverse of the hash space covered by NSEC3 ranges found so far. The check in line 12 is a binary search over the sorted list of gaps in logarithmic time. Another performance optimization is in the `genRandomName` function, which iterates linearly through names starting from a randomly chosen name. Random input names are not necessary because small changes in the input name will result in seemingly random output hashes due to the avalanche effect of the underlying SHA-1 hash function.

The query for a non-existent name results in a new NSEC3 record, which is written to a database and is used to update the sorted list of gaps in memory. The new NSEC3 record closes a gap or cuts a part of it (Fig. 4). However, we also have to consider that names on the server may have been added, removed or renamed in the meantime. This happens several times a day for large TLDs and obsoletes part of the NSEC3 records crawled so far. If a retrieved NSEC3 record overlaps outside of the boundaries of an expected gap, we will remove the affected, obsolete NSEC3 record from our database and update the gap data structure accordingly. Furthermore, different authoritative name servers may serve different versions of zone data because zone updates are not synchronized precisely. This can lead to loops with continous DNS queries and data updates. In case of inconsistent NSEC3 records, we use the SOA serial number from the negative DNS response to identify fresh NSEC3 records and drop old ones.

The worst case complexity of our algorithm is unbound due to unpredictable output hash values. However, we provide averages for a case study in Section VII. The network traffic caused by the algorithm is linear to the number of names in a zone. For every NSEC3 record one query is sent, plus an insignificant amount of extra queries for handling inconsistent NSEC3 records after zone updates. An extra query is necessary if `genRandomName` yields an actually existing name. During testing and evaluation, we never encountered this case and consider this merely a theoretical possibility.

V. Hash Breaking

After having crawled the NSEC3 records of a DNS zone, the next part is to find the corresponding cleartext names. We discuss three GPU-based hash breaking methods in this section. All methods can be executed offline, i.e. communication with the name server is not required. Each of the attack methods consists of a performance-critical loop that generates a candidate name, computes the NSEC3 hash and checks whether the resulting hash value matches one of the crawled NSEC3 hashes. The attack methods differ on how they generate a candidate name but the remaining steps are the same.

The crawled NSEC3 hash values are stored as sorted array in GPU memory. The candidate name is hashed with the given salt and number of iterations. A binary search is used to check whether the resulting hash value is an element of the sorted array. Since random global memory access limits the performance of GPU applications, we also use a bloom filter to detect misses more efficiently. A bloom filter is a probabilistic data structure to efficiently check if an element is part of a set. It is created by using a bit array of length N and setting k bits in it for every element of the set. Since the positions of these k bits are specific to each element, a query has to check k bits in the worst case. If only one of the checked bits is zero, the element is definitely not in the set. On the other hand, if all k bits are set, the element might be in the set. In this case, binary search will be used to get a definite answer. Using this hybrid scheme one memory access is sufficient in most cases to recognize hash misses. Since the vast majority of all checked candidate names are misses, this optimization provides a significant speedup.

Compared with NSEC3 hash crawling, the hash breaking requires significantly more computation time. Distributed computing can be used to spread the workload among several hosts. As each candidate name in the search space is identified by an index number, slices of the search space can be easily distributed to different hosts.

A. Brute-Force Attack

A brute-force attack enumerates all names of a specific length (aaa, aab, aac, ...) and checks whether their hash value is an element of the array of crawled NSEC3 hashes. While

this attack is exhaustive and thereby guaranteed to find every present name of the specified length, it is computationally feasible only for names up to a certain length. The character set to be used in the brute-force attack must be specified beforehand. Though the DNS protocols supports arbitrary octets in domain names, typically a limited set of 37 or 38 characters is used. Unicode characters used in Internationalized Domain Names are encoded to this limited ASCII character set.

An optimization for the generation of candidate names is to use two different functions. First, a function to map a numerical index to a word by using a number system conversion algorithm. This allows GPU workers to efficiently determine their starting candidate name. Second, a dedicated incrementation function to map a candidate name to the next candidate name. This avoids unnecessary index conversions because in most cases only the rightmost character changes.

B. Dictionary Attack

As brute-force attacks are not feasible for long names, we also implemented a dictionary-based attack. In a dictionary attack the name generation step boils down to looking up a name in a static list.

Using a dictionary alone to break hashes can only yield as many results as it has entries. To increase the number of candidate names we applied some basic combination/transformation rules. Therefore, the dictionary list is enhanced by an insertion wordlist. For every pair from both lists the insertion word is inserted at every position of the dictionary word. For example, a dictionary $D := \{abc, def, ...\}$ and an insertion wordlist $I := \{12, 34, ...\}$ yields a combined list $L := \{abc, 12abc, a12bc, ..., def12, 34def, ...\}$. The number of total candidate names is $|D| + |I| \sum_{w \in D} 1 + len(w)$.

Possible choices for the insertion wordlist are for example all words of the dictionary up to a specific length or using its most frequent n-grams.

C. Markov Attack

Markov chain-based approaches use a language model to estimate how probable a word is [14]. For every letter in a word the previous letters are used to judge how likely the letter is. Multiplying each single letter probability yields the probability of the whole word. The order of the Markov chain is here defined by the number of previous letters considered in the probability calculation. The language model is generated by a training phase in which wordlists are used to calculate transition probabilities. In our approach, we use a *first-order Markov chain* to prevent overtraining. If the order is chosen too high, a Markov attack will be basically a dictionary attack.

Using this language model it is possible to efficiently enumerate all words down to a defined probability threshold [15]. Parameters for this algorithm are the maximal word length and the minimal word probability. The enumerated words are not sorted by their probability, so the maximal word length and the minimal word probability have to be chosen in such a way that it is feasible to process all resulting candidate names in a reasonable time—otherwise highly probable candidate names might not be enumerated in a given time frame. It is possible to efficiently calculate how many words would be enumerated given a certain parameter set [14]. With a given computing power in terms of checked words per time and a given time span to perform this attack it is possible to choose the appropriate parameters.

We propose to use the names found by the brute-force and dictionary attacks to train the Markov attack specific to a DNS zone. That way we do not have to assume that registered names are derived from some natural language. Instead, the language model is based on a subset of actually registered names in the DNS zone.

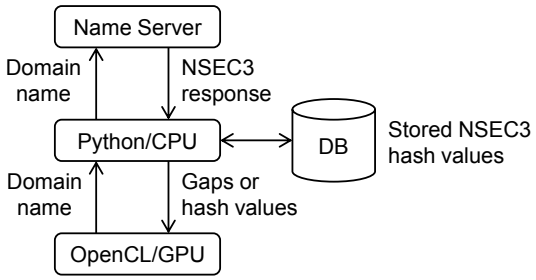

Fig. 5. System architecture.

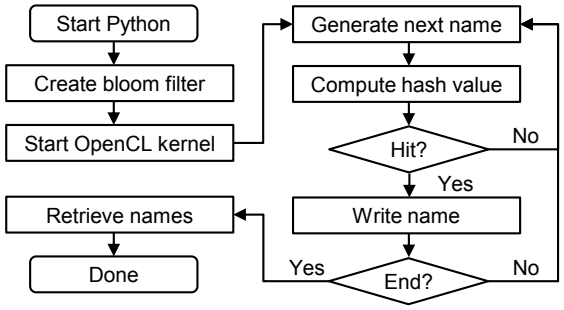

Fig. 6. Hash breaking program flow on CPU (left column) and GPU (right column).

VI. Implementation

We implemented the NSEC3 crawling and hash breaking algorithms in Python and OpenCL. Fig. 5 shows the system architecture. The Python code running on the CPU comprises the overall program logic. It sends domain queries to authoritative name servers and invokes the OpenCL *kernels*, which run on the GPU for computationally intensive tasks. Results are saved in a PostgreSQL database.

The program flow of the NSEC3 hash breaking is shown in Fig. 6. The Python program passes the set of NSEC3 hash values and the bloom filter to the GPU and starts the OpenCL kernel corresponding to the type of attack (i.e. brute-force, dictionary, Markov). The search space is split into chunks, which are processed in parallel by OpenCL *work-items*. All work-items are independent of each other except for competing global memory accesses e.g. to the bloom filter. Work-items

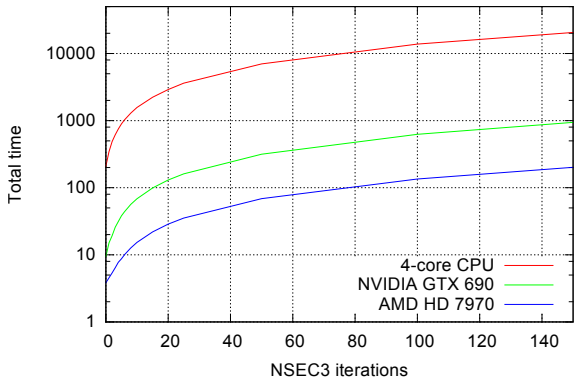

Fig. 7. Time needed for hashing 2.5 billion names (less time is better).

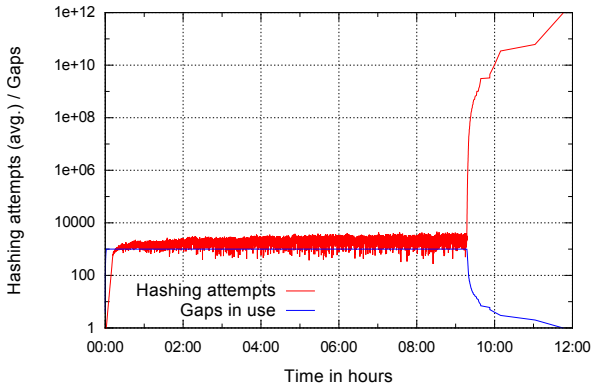

Fig. 8. Hashing attempts to find a new NSEC3 record in com.

hold the program state in private memory except for the input buffers (around 70 Mbytes) because they exceed the size of the private memory (less than 256 kbytes).

VII. Evaluation

The rationale for using OpenCL was to utilize AMD graphic cards which have been reported to work faster than NVIDIA cards with 32-bit integer operations [16] like SHA-1 uses. To test this assertion, we hashed $37^6 = 2.5$ billion names with various iterations on a quad-core Intel 2.67 GHz CPU, an NVIDIA GTX 690 and an AMD HD 7970. Fig. 7 shows that the HD 7970 needs the least time to finish. For $i = 0$, the CPU computed 12 Mhash/s, the NVIDIA GPU computed 270 Mhash/s and the AMD GPU computed 667 Mhash/s.

We will now evaluate our proposed attack methods by applying them in a case study on the com TLD.

A. Hash Crawling

Recall that the hash crawling algorithm (Section IV) consists of keeping track of gaps and finding NSEC3 records that fill the gaps. The number of hashing attempts needed for a certain chance to find a new NSEC3 record can be derived from the sum of sizes of gaps. For example, when the sum of sizes of gaps constitutes 50% of the overall hash space, there is a 50% chance that the next hashing attempt will cut a gap. With 10% gaps, $\frac{1}{0.1} = 10$ hashing attempts will be needed on average. With each found NSEC3 record, the sum of sizes of gaps strictly decreases and the average number of hashing attempts needed to find a gap strictly increases, except when resolving inconsistencies after zone updates (occurred 5 times).

An implementation detail is that we pass only the 1000 leftmost gaps to the GPU. Working with many gaps increases the computation time of certain implementation parts, e.g. for assembling the gap buffer when passing it to the GPU and for the binary search over the list of gaps. However, 1000 gaps suffice to find a new NSEC3 record in an insignificant amount of time, even with CPU computation. This can be seen in Fig. 8: the number of necessary hashing attempts oscillates around an average of 2059 ($\pm$ 671) when the number of gaps in use is capped at 1000. Two thousand SHA-1 hashes can be computed in less than 1 ms. The number of hashing attempts needed does not increase significantly until the number of pending gaps falls clearly below 1000. After 12 hours of hash crawling, we retrieved 345,664 NSEC3 hashes from the com TLD. Note that the crawling process was consciously not further optimized for highest possible throughput to avoid overloading of the com servers.

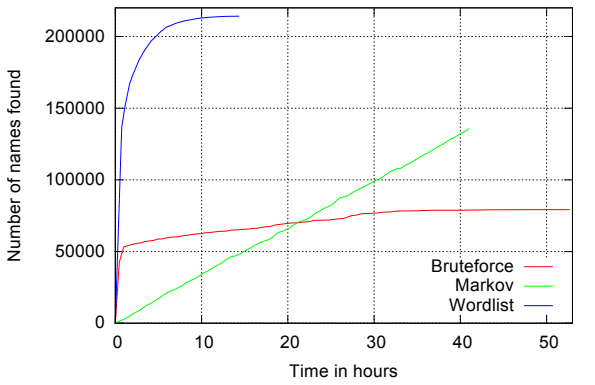

Fig. 9. Comparison of attack methods

B. Hash Breaking

The numbers of names found by each of the three attack methods are shown in Fig. 9. The time has been measured with one AMD HD 7970 GPU. We were using a 37 character set for all attacks ('-', '0'...'9', 'a'...'z'). We omitted the underscore character and the dot because they are not used for domain names in TLD zones.

The **brute-force attack** finished in less than 5 minutes for all words with up to 7 characters and found 32,795 names. The brute-force attack for 8 and 9 characters took 1.4 hours and 51 hours. The attack becomes infeasible for longer names on a single GPU because the computing time multiplies with 37 for each additional character. Overall, the brute-force attack for up to 9 characters checked 1.3×10^{14} candidate names and found 79,243 names (22.92% of NSEC3 hashes).

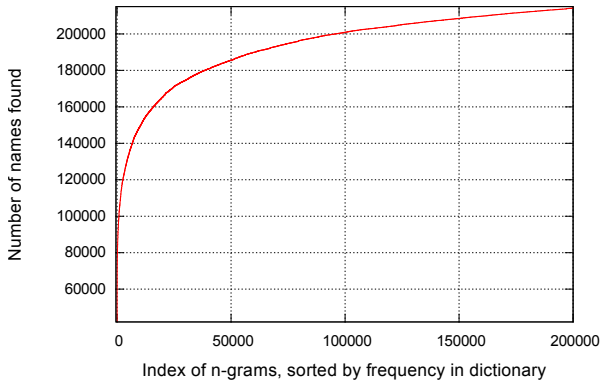

Fig. 10. Effectiveness of insertion words

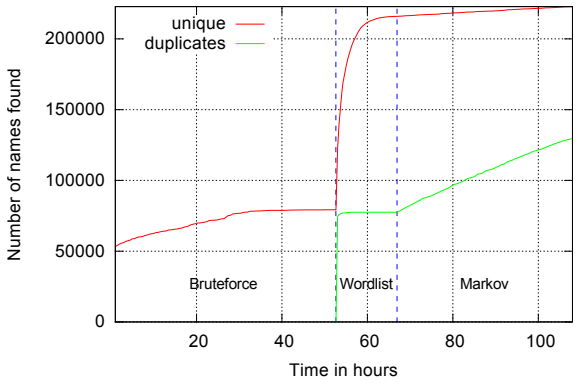

Fig. 11. Names found over time

The success of the **dictionary attack** depends on the quality of the dictionary. We extracted domain labels from the Alexa[5] and Quantcast[6] lists of top 1 million websites and similar lists into a common dictionary. Another data source was the list of reverse DNS names from the Carna botnet[7]. Note that the list of reverse DNS names does not by far comprise all domain names. The majority of reverse names are structured names that would increase the dictionary size without providing gain, e.g. names like `ip-1-2-3-4.example.net` that differ only in numerical labels. We processed the list of reverse DNS names to remove structured names and keep mostly alphabetic words. In total, the dictionary consisted of 7.1 million words. Based on this dictionary, we created a list of insertion words consisting of the 200,000 most common n-grams with n=1 to n=15. This lead to 1.7×10^{13} candidate names, out of which 214,181 names were found after 14 hours of computation (61.96% of NSEC3 hashes).

Interestingly, the dictionary attack has found all but 1725 names (2.17%) that the brute-force attack had found. This suggests that the quality of the dictionary is high, at least when used for the `com` TLD. The use of the insertion wordlist is also a very effective mechanism: the number of names found increased from 40,045 with the plain list by a factor of ≈ 5.34. As shown in Fig. 10, the most frequent n-grams are also the most effective insertion words. Nevertheless, even the less frequent ones contribute to the number of names found.

For the **Markov attack**, we ported code provided by [14] to OpenCL. The Markov attack is the slowest of the three methods due to sophisticated program logic and extra memory accesses. The brute-force attack computes more than 700 Mhash/s with long-running jobs, the dictionary attack computes 350 Mhash/s and our Markov implementation computes 120 Mhash/s. We trained the language model with the names found so far by the brute-force and dictionary attacks. It took 41 hours to compute 1.3×10^{13} candidate names, which lead to 135,574 names found (39.22% of NSEC3 hashes).

[5]http://s3.amazonaws.com/alexa-static/top-1m.csv.zip
[6]http://ak.quantcast.com/quantcast-top-million.zip
[7]http://internetcensus2012.bitbucket.org/

However, these are only the statistics for each attack itself. We have to consider duplicate results found by more than one method. Fig. 11 shows the number of unique names found, and additionally the number of duplicate results. Overall, 222,749 NSEC3 hashes (64.44%) were broken after applying the three attacks. The dictionary attack was the most efficient method.

C. Discussion

In case of `com`, NSEC3 has protected the remaining 35.56% of domain names that had not been found after 4.5 days of GPU computation. The name server operator pays for this privacy protection with operational costs in terms of additional network traffic and CPU load for hashing. Typical negative response sizes[8] are 665–693 bytes with NSEC but 983–1000 bytes with NSEC3. Schaeffer [7] determined in a laboratory setup that the maximum number of negative responses that the NSD name server can handle decreases from 47k responses/s with NSEC to 25k responses/s with NSEC3 and $i = 1$ (19k with $i = 15$). The server-side costs are present continuously for all negative responses during normal operation, not just during zone enumeration attacks. The server operator thus needs to consider the NSEC3 overhead when provisioning the authoritative name servers to ensure a decent quality of service.

An attacker attempting zone enumeration has to compute a vast amount of NSEC3 hashes to find the correct domain names. The workload is significantly higher on attacker-side than on server-side, but the computation can be performed in parallel batches on cost-efficient consumer-grade hardware. For example, the AMD Radeon HD 7970 that we used cost € 360 in 2013. Using a spare 5 year old desktop machine this setup had an average power consumption of about 279 watts, which costs a household in Germany about € 8.43 for 4.5 days of computation.

The NSEC3 specification [5] acknowledges that NSEC3 is susceptible to dictionary attacks and advises to adjust the iterations to increase the hashing workload for attackers.

[8]DNSSEC responses for "`_.se`", "`_.us`", "`_.de`" and "`_.com`".

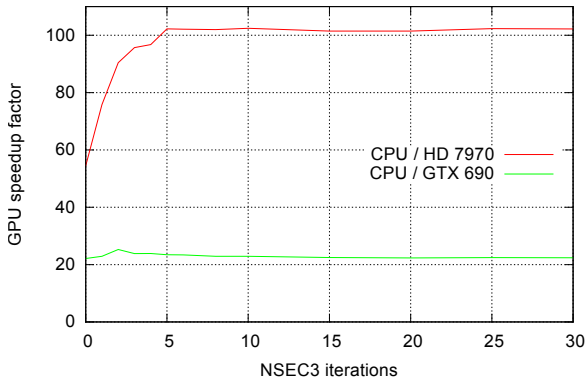

Fig. 12. Performance comparison CPU vs. GPU.

However, this increases the NSEC3 overhead at about the same ratio, i.e. doubling the workload for attackers doubles the server-side CPU overhead for operators. This is different than e.g. increasing the key size of a cipher, where the workload increases slightly for legitimate users but exponentially for attackers. An attacker with one GPU can compensate for a doubling of iterations with an investment of another GPU (plus electricity). In fact, increasing the iterations from low values even improves the relative efficiency of a GPU, which is shown in Fig. 12. The HD 7970 GPU is about 55 times faster than a quad-core 2.67 GHz CPU with $i = 0$, but 100 times faster with $i \geq 5$. The utilization improves until saturation because the stream processors are performing more parallel hashing work between global memory accesses. Although it depends on individual operational costs and percentage of negative responses, it is thus likely that operators of busy servers pay more for increasing iterations than attackers do.

VIII. Conclusion

We presented an attack analysis on the privacy goal of the NSEC3 authenticated denial of existence method that is used in DNSSEC. To the best of our knowledge, this paper presents the first GPU-based NSEC3 zone enumeration attack on a large TLD. We conclude that GPUs are very efficient for NSEC3 hash breaking: 64.44% of all NSEC3 hashes of the `com` TLD could be reversed after 4.5 days of GPU computation.

Our findings suggest that increasing the hash iterations on the DNSSEC server puts the attacker at an economic advantage. This is 1) due to an improvement of the relative efficiency of a GPU compared with a CPU when increasing iterations, and 2) because hash breaking attacks can be performed offline on consumer-grade hardware but defensive measures run online on server-grade infrastructure. DNS operators can use the tools we developed as part of a cost-benefit assessment whether the NSEC3 privacy assertion justifies the extra server costs. The Python and OpenCL tools are open source and available for public download.[9] Another use case was proposed by Rafiee et al. [17]: where IPv6 network scanning is not feasible, IPv6 host addresses can be retrieved with NSEC3 zone enumeration. Our GPU-based NSEC3 hash breaking speeds up this retrieval process by several orders of magnitude.

For future work, we suggest the exploration of GPU-based NSEC3 accelerators for DNSSEC servers. With the utilization of a dedicated GPU, the server operator could free CPU resources for other tasks and regain an economic advantage over attackers.

[9] http://dnssec.vs.uni-due.de/nsec3

References

[1] J. Daley, "Confirming the Nominet position," *namedroppers mailing list*, May 2004.

[2] The European Parliament and the Council of the European Union, "Directive No. 96/9/EC of 11 March 1996 on the legal protection of databases," *Official Journal of the European Communities*, no. L77, p. 20, 1996.

[3] M. Sanz, "DNSSEC and the Zone Enumeration," *European Internet Forum*, Oct. 2004.

[4] O. Schily, "Bundesdatenschutzgesetz (BDSG) [*Federal Data Protection Act*]," *Bundesgesetzblatt*, p. 66, Jan. 2003.

[5] B. Laurie, G. Sisson, R. Arends, and D. Blacka, "DNS Security (DNSSEC) Hashed Authenticated Denial of Existence," RFC 5155 (Proposed Standard), Internet Engineering Task Force, Mar. 2008, updated by RFCs 6840, 6944. [Online]. Available: http://www.ietf.org/rfc/rfc5155.txt

[6] G. Michaelson, "DNSSEC means more traffic," Aug. 2010. [Online]. Available: http://meetings.apnic.net/30/program/full

[7] Y. Schaeffer, "NSEC3 Hash Performance," Mar. 2010, NLnet Labs document 2010-002.

[8] J. Bau and J. C. Mitchell, "A Security Evaluation of DNSSEC with NSEC3," *IACR Cryptology ePrint Archive*, vol. 2010, p. 115, 2010.

[9] D. Eastlake and P. Jones, "US Secure Hash Algorithm 1 (SHA1)," RFC 3174 (Informational), Internet Engineering Task Force, Sep. 2001, updated by RFCs 4634, 6234. [Online]. Available: http://www.ietf.org/rfc/rfc3174.txt

[10] P. Mockapetris, "Domain names - concepts and facilities," RFC 1034 (INTERNET STANDARD), Internet Engineering Task Force, Nov. 1987, updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936. [Online]. Available: http://www.ietf.org/rfc/rfc1034.txt

[11] S. Weiler and J. Ihren, "Minimally Covering NSEC Records and DNSSEC On-line Signing," RFC 4470 (Proposed Standard), Internet Engineering Task Force, Apr. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4470.txt

[12] ICANN, "Base Registry Agreement," Nov. 2013. [Online]. Available: http://newgtlds.icann.org/en/applicants/agb/base-agreement-contracting

[13] D. J. Bernstein, "Breaking DNSSEC," Aug. 2009, keynote lecture at Workshop on Offensive Technologies (WOOT). [Online]. Available: http://cr.yp.to/talks/2009.08.10/slides.pdf

[14] S. Marechal, "Advances in password cracking," *Journal in Computer Virology*, vol. 4, no. 1, pp. 73–81, 2008. [Online]. Available: http://dx.doi.org/10.1007/s11416-007-0064-y

[15] A. Narayanan and V. Shmatikov, "Fast dictionary attacks on passwords using time-space tradeoff," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. New York, NY, USA: ACM, 2005, pp. 364–372. [Online]. Available: http://doi.acm.org/10.1145/1102120.1102168

[16] M. B. Taylor, "Bitcoin and the age of bespoke silicon," in *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 16:1–16:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2555729.2555745

[17] H. Rafiee, C. Mueller, L. Niemeier, J. Streek, C. Sterz, and C. Meinel, "A flexible framework for detecting ipv6 vulnerabilities," in *Proceedings of the 6th International Conference on Security of Information and Networks*, ser. SIN '13. New York, NY, USA: ACM, 2013, pp. 196–202. [Online]. Available: http://doi.acm.org/10.1145/2523514.2527001