On-device IoT Certificate Revocation Checking with Small Memory and Low Latency

Xiaofeng Shi
xshi24@ucsc.edu
University of California
Santa Cruz, CA, USA

Shouqian Shi
sshi27@ucsc.edu
University of California
Santa Cruz, CA, USA

Minmei Wang
mwang107@ucsc.edu
University of California
Santa Cruz, CA, USA

Jonne Kaunisto
jkaunist@ucsc.edu
University of California
Santa Cruz, CA, USA

Chen Qian
cqian12@ucsc.edu
University of California
Santa Cruz, CA, USA

ABSTRACT

Allowing a device to verify the digital certificate of another device is an essential requirement and key building block of many security protocols for emerging and future IoT systems that involve device-to-device communication. However, on-device certificate verification is challenging for current devices, mainly because the certificate revocation (CR) checking step costs too much resource on IoT devices and the synchronization of CR status to devices yields a long latency. This paper presents an on-device CR checking system called TinyCR, which achieves 100% accuracy, memory and computation efficiency, low synchronization latency, and low network bandwidth, while being compatible with the current certificate standard. We design a new compact and dynamic data structure called DASS to store and query global CR status on a device in TinyCR. Our implementation shows that TinyCR only costs each device 1.7 MB of memory to track 100 million IoT certificates with 1% revocation rate. Checking the CR status of one certificate spends less than 1 microsecond on a Raspberry Pi 3. TinyCR can also be updated instantly when there are new certificates added or revoked.

CCS CONCEPTS

• **Security and privacy → Mobile and wireless security**.

KEYWORDS

IoT security, authentication, certificate revocation checking

ACM Reference Format:
Xiaofeng Shi, Shouqian Shi, Minmei Wang, Jonne Kaunisto, and Chen Qian. 2021. On-device IoT Certificate Revocation Checking with Small Memory and Low Latency. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3460120.3484580

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8454-4/21/11.
https://doi.org/10.1145/3460120.3484580

1 INTRODUCTION

Recent years have witnessed the rapid growth of Internet of Things (IoT) devices widely deployed in various applications [7]. With the growing trend that IoT services scale from local area domains to wider area domains, there is an increasing demand for secure peer-to-peer communication protocols in a universe with millions of IoT devices. Under this context, many security protocols for IoT should be re-designed. For example, future IoT devices can use any untrusted access point (such as a public 5G AP) to connect to the Internet or use the short-range wireless media such as Bluetooth and visible lights to communicate with another device directly.

Thus, peer-to-peer *device authentication* becomes a fundamental security problem of novel IoT and the building block of many emerging critical IoT security protocols for communication privacy (such as the TLS-style protocols) as well as IoT data authenticity and integrity (such as the digital signature protocols) [1, 2, 20, 23]. The state-of-the-art solution of device authentication is to use *device certificates* based on the Public Key Infrastructure (PKI) [1, 2]: each device is assigned a device certificate by a Certification Authority (CA), which can be used as a digital signature of its public key. For example, the IoT architecture of Symantec enterprise security (now Broadcom) allows assigning certificates to millions of devices [1] and verifying the device certificates by the device management servers [2]. Note that the "CA" for IoT devices refers to not only the general public SSL certificate authority, but the private certificate issuer in a service managed by the service provider.

On-device certificate verification [32, 35], i.e., allowing one IoT device to verify the certificate of another device, remains a challenging problem mainly due to the high latency and bandwidth cost of the revocation-checking step. On-device certificate verification is a vital step for IoT security: 1) Many emerging and future IoT applications require secure peer-to-peer communication directly or via the Internet, such as autonomous robotic systems, vehicular communication, wearable healthcare systems, smart industrial control, and IoT-based post-disaster management. A device should use its own data and power to verify the certificate of another communicating device, to further build a secure channel using protocols such as DTLS [27]; 2) An IoT device may need to process the sensing data collected from other devices, which carries the digital signatures from the sensing sources [20]. Verifying the public-key certificates is essential in validating digital signatures to ensure data authenticity and integrity. For example, in a smart city, an IoT sensor has to authenticate the mobile devices of the authorized users so that it can only provide the sensing data to the users who subscribe to the service. Meanwhile, user devices have to verify the signatures of sensing data to ensure the data are not tempered by an attacker.

Why on-device certificate verification is challenging? Verifying a digital certificate takes three main steps: 1) check its validity period; 2) validate the CA's digital signature using CA's public key; 3) verify the certificate revocation (CR) status. Step 1 is simple. Step 2, although involving expensive public key cryptography, takes bounded time and memory. Step 3 is considered an expensive process even for a desktop machine [19, 29, 38]. Some issued digital certificates may have been revoked by the CAs [17, 38, 39] due to a number of reasons: 1) the device is stolen; 2) the private key of a device could be compromised by attackers; 3) the CA may find that a certificate is a mis-issuance; 4) a device may unsubscribe from an IoT service while its certificate is still within its valid time period; 5) the database of an IoT service provider or device manufacturer might be hacked and the private key information could be leaked. Upon being notified with these situations, the CA should immediately labeling these certificates as "revoked".

For SSL certificates, a certificate revocation list (CRL) [15] containing all revoked certificates is prepared by the CA and sent to web browsers for revocation checking [29, 38, 39]. The CRL introduces substantial overhead even if it runs on a desktop machine, because the CRL size is proportional to the number of revoked certificates, which can be in millions [29, 38, 39]. For IoT, the overhead problem is more severe, because: 1) the number of IoT devices could be more than that of web servers; 2) the memory, CPU, and network resource of an IoT device is much weaker than those of a desktop. An alternative solution is to use Online Certificate Status Protocol (OCSP) [22], which increases CR verification latency and the risk of leaking user privacy (such as accessing history of the device) [30].

In addition, unlike web servers, IoT devices are small in size, have better mobility and are usually maintained by individual users, which also means the devices are much easier to be hacked or stolen. At service-level, it is also more common for an IoT device to be unsubscribed from a service while the certificate is still in valid period. Hence, revocations for IoT certificates happen more frequently and have to be properly handled. When a revocation happens, how soon other parties are aware of the revocation and no longer trust the device becomes a rather critical metric of the security property in the protocol.

The main requirements of practical on-device IoT CR checking are summarized as follows:

(1) **Accuracy**: A device should determine a certificate revocation status without error.
(2) **Efficiency**: The protocol should cost small memory, computation, and network resource on IoT devices.
(3) **Low latency**: Two types of latencies are essential, namely the synchronization latency and query latency (defined in Sec. 3). Both latencies should be maintained low.
(4) **User privacy:** The protocol should not leak the identities of the accessing devices, locations, and/or communication pattern/frequency of users.
(5) **Compatibility:** The protocol is required to be compatible with current certificate standards and existing certificates.

To our knowledge, there is no solution for on-device IoT CR checking that satisfies all above requirements. Recent works on web certificates [19, 29] may focus on a subset of them, but fail to address all, as analyzed in Sec. 2.

This work presents TinyCR, **the first IoT certificate system for on-device CR checking, which achieves all the five listed requirements**. Our key innovation is a new compact and fast data structure named Dynamic Asymmetric Set Separator (DASS) to represent the revocation status of all certificates on IoT devices **with zero error**. TinyCR also includes a management program running on a server maintained by the IoT service provider to synchronize the DASS on devices, which can be easily replicated to avoid a single point of failure. We have implemented both the management and on-device programs. TinyCR is very efficient: it only needs 1.7 MB on-device memory to track the CR status of 100 million certificates with 1% revocation rate and verifies a certificate revocation status within 1 microsecond on a Raspberry Pi 3. The device can also be instantly synchronized when new revocations happen. Hence, TinyCR uses very small resource to effectively protect the whole IoT network from the attackers who intend to abuse the revoked certificates. Our source code of TinyCR is open to public for re-use and results reproduction *(link)* [3].

Based on our analysis and evaluations, TinyCR is the ideal solution for CR in any scenario that satisfies: 1) Users need fast or frequent authentication. TinyCR has a clear advantage in latency compared to OCSP and equivalent performance compared to CRLite [19] (the state of the art). 2) Each user device has a limited size of memory that can be used to store the CR list, such as several MBs. TinyCR costs slightly less memory than CRLite and much less than other CRL solutions when the revocation ratio is low. 3) Low CRL synchronization latency is crucial for better security. The faster the devices can realize a revocation made by the CA, the lower is the risk for certificate abuse. To our knowledge, TinyCR is the first on-device CR checking protocol that supports real-time or high-frequency updating in response to the certificate set changes. 4) User devices prefer low bandwidth cost and the dynamics of certificates are moderate. Experiments show that the bandwidth cost of TinyCR is orders of magnitude lower than that of CRLite, if the number of new certificates added per day or number of revoked certificated per day are fewer than 1% of the existing certificate sets. If these changes per day are on the same order of magnitude of the size of total certificates – although unlikely in practice – TinyCR still wins if high updating frequency (i.e., one update per hour) is required, while costs more bandwidth for infrequent update settings (i.e., one update per day). We believe these situations well characterize the scenarios of IoT P2P communication.

The rest of this paper is organized as follows. In Sec. 2, we review the state-of-the-art approaches for CR verification. In Sec. 3, we present an overview of the TinyCR system and the threat model. The data structure and the optimization methods of DASS are illustrated in Sec. 4, and the system design of TinyCR is shown in Sec. 5. We present the experimental results of TinyCR in Sec. 6. We discuss the real deployment concerns in Sec. 7. We present the security analysis in Sec. 8 and conclude the paper in Sec. 9.

Method	Memory cost	Query time	Δ-msg size per update	Δ-msg size per day	Sync. latency	Push model	CA compat.
CRL [15, 21]	~ 38 MB	≫ 250 ms	-	-	≫ 250 ms	×	✓
OCSP [21, 22]	~ 1 KB/req.	≤ 250 ms	-	-	-	×	✓
Othello [37]	29.1 MB	**< 1 μs**	**0~100 B**	**0~20 KB**	**< 1 ms**	✓	✓
CRLite [19]	**1.7 MB**	**< 1 μs**	-	0.53 MB	1 day*	✓	✓
Let's Rev. [29]	**1.3 MB**	~ 10 ms	-	62.6 KB	1 day	✓	×
TinyCR (ours)	**1.7 MB**	**< 1 μs**	**0~108 B**	**2.8~21.6 KB**	**< 1 ms**	✓	✓

Table 1: Comparison of certificate revocation verification protocols with 100 million certificates, assuming 1% revocation rate and 0.02% new revocations per day. $*$ Can be shorter with higher daily delta message cost.

2 RELATED WORK

Existing approaches for checking CR status are mainly based on either remote or local queries. A typical remote querying protocol is the Online Certificate Status Protocol (OCSP) [11, 22]. In OCSP, an authorized OCSP server returns the signed revocation status for every single certificate validation request from the client. An important weakness of this on-demand remote checking protocol is that the OCSP clients suffer from privacy leakage of their visiting history and trace pattern to the OCSP servers, as the server knows the exact time when one device builds an SSL session with another. In addition, it requires the devices to have access to the OCSP servers when validating the certificate and introduces additional network latency for each query.

On-device CR checking preserves user privacy by allowing them to check CR status locally through a compact data structure model, such as CRLSets[18], OneCRL[13], CRLite[19] and CRV[29], which is periodically synchronized from the CAs or device management servers. These methods are also known as the push-based models. For these methods, the protocol designers need to consider the on-device memory cost and query/updating efficiency of the data oracle, as the CRLs are usually large. For example, CRLSets[18] and OneCRL[13], which have been used in web browsers including Chrome and Firefox, trade checking accuracy for efficiency by maintaining a subset of the CRLs.

Recent methods of SSL/TLS CR checking, such as CRLite [19] and Let's Revoke [29], have been designed to achieve both query or memory efficiency and accuracy on the client-side. The key idea of the two methods is utilizing an efficient set-query data structure as a compact summary of the large CRLs. Our proposed TinyCR also adopts a similar design framework to enable efficient on-device error-free CR checking. However, existing methods do not meet the requirements of IoT CR checking as listed in Sec. 1. For example, CRLite [19] uses a data structure called filter cascade to check CR status with small memory costs. The cost of CRLite delta message updated is high as any revocation will cause a reconstruction of the filter cascade. Hence CRLite is designed for 1-4 updates per day and could have more with significantly higher bandwidth cost. The low updating frequency yields a longer unprotected time, during which the attackers can abuse the revoked certificates in IoT applications. Let's Revoke [29] resolves the high bandwidth cost of CRLite, but it requires a new Revocation Numbers (RN) extension filed in the certificate. CAs are required to issue the RNs and maintain/update the revocation checking structure. Thus, the protocol cannot be supported by the existing X.509 certificates and the current CA's workflow to revoke a certificate. Since the RN filed is generated based on how many certificates have been issued by this CA for a given expiration date, it also leaks additional behavior information of the CAs. For example, anyone who sees a few certificates may infer how many certificates issued by this CA will expire on a particular date. Let's Revoke is also updated daily.

A comparison of CR checking methods is shown in Table 1, where the results of Let's Revoke [29], CRL [15] and OCSP [22] are from the original papers or a measurement paper [21]. We mainly concern with the on-device memory cost, query efficiency, updating message size, extra synchronization latency (excluding network latency), whether the model leaks user's accessing history and whether the model is back-compatible with the existing X.509 certificates. Note that all the methods are required to provide zero error assuming the on-device data models are synchronized with the latest CRLs provided by the CAs. However, for push-based methods, when the local data models are not consistent with the newest CRLs, revoked certificates may still be accepted by the checking model. Of the compared methods in the table, only Othello [37] and our proposed TinyCR can support real-time synchronization of the on-device data model, whereas Othello requires significantly more on-device memory than TinyCR in practice.

In addition, other methods such as OCSP Stapling [11, 14], Revocation in the Middle (RITM) [31] and Certificate Revocation Guard (CRG) [16]offload the CR checking process to the server (the responder of the connection) or a middle-box intercepting TLS traffic. However, the responder-based checking is not a scalable solution when the responder is another IoT device, due to its high cost on the device and the CA side. In addition, if the accessed server fail to provide a valid OCSP Staple in the handshake, the connect cannot be established or can never be secure.

3 SYSTEM AND THREAT MODELS

3.1 System Model

Secure communication in an IoT network requires that the devices can authenticate each other, which is nowadays achieved by digital certificates based on the Public Key Infrastructure (PKI). The TinyCR system enables the IoT devices to maintain a compact representation of the CRL with 100% query accuracy through a data structure named Dynamic Asymmetric Set Separator (DASS). Fig. 1 illustrates the system model of TinyCR. The CR checking protocol is designed on top of the current IoT/Mobile device management

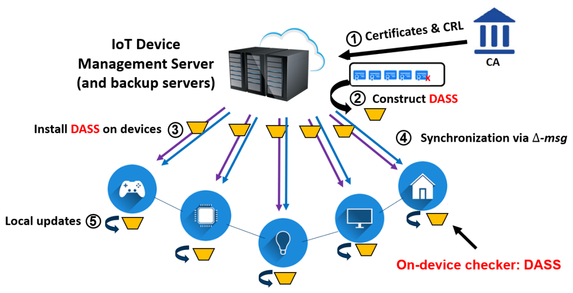

IoT Device Management Server (and backup servers)
① Certificates & CRL
CA
② Construct DASS
Install DASS on devices ③
④ Synchronization via Δ-msg
Local updates ⑤
On-device checker: DASS

Figure 1: System model of TinyCR

system (MDM) [2], where an IoT device management (IDM) server requests the certificates from CAs for users and delivers the certificates (usually through a patch file for installation) to the end devices when the devices are registered to the service. Note that the CA could be the world-wide issuers or the PKI service that is managed by the service provider (such as Symantec Managed PKI Service [2]). The CA issues new certificates at the request of IDM and actively sends updated CRLs to the IDM server when a new revoke happens (Operation 1). The IDM server constructs and updates DASS based on the global certificate database and the newest CRLs (Operation 2). For each device, the IDM server will install DASS on it (Operation 3) when the device is enrolled to the service. The DASS installation process could be conducted together with the certificate installation on the device. Whenever the CRLs have changed, the IDM server would send update messages if necessary in a pushing way (Operation 4). In the system, each device can check the CR status of *any certificate* completely based on DASS and perform local updates to DASS (Operation 5). The model fits or can be easily extended to most IoT management systems.

We define two types of latencies in the whole process: 1) **synchronization latency** is defined as from the time of the CA revoking a certificate to that of a device being able to find this revocation event from its local state; 2) **query latency** is defined as the time used to get the CR status on a device. TinyCR aims to minimize these two latencies. There is another type: revoking latency, defined as from the time a certificate being hacked to that of the CA revoking the certificate, which is out of scope of this work.

3.2 Threat Model

Since the certificates issued by the CA might be revoked, an attacker can effectively abuse the revoked certificates. The security vulnerability in this process is apparent: the revoked certificate are still valid if a device only verifies the expiration dates and CA signatures (called time- and signature-valid). Hence the on-device maintenance of all revoked certificates is necessary. We are mainly concerned about the attacker who can obtain a set of time- and signature- valid but revoked certificates and the corresponding private keys, such that the attacker can masquerade as legitimate users in the IoT to perform Man-in-the-Middle (MITM) attacks during TLS setups or tamper with the sensing data. We summarize the threat model and assumptions in this paper:

1. The IDM server and the CAs are trusted and they communicate via a secure channel with integrity. Each device also maintains a channel from/to the IDM server with integrity.

2. The attacker can acquire a set of time- and signature-valid certificates. But this behavior could be detected by the CA and those certificates are revoked.

3. The attacker can obtain all information of the shared DASS, but is not able to tamper it.

4. The size of the certificate universe in IoT is large. Note that the current number of web server certificates is on a scale of 100 million [19, 29]. It is a reasonable estimate that the future IoT devices should be much more than the number of web servers.

5. The number of revoked certificates is smaller than that of legitimate ones in an IoT network by at least an order of magnitude. Otherwise, the CA who issues many revoked certificates will not be trusted. This assumption is validated by measurements [34].

6. IoT devices have limited memory and computing resources, while the IDM server and attackers can be powerful. The IDM server knows all time- and signature-valid certificates.

7. We do not consider deny-of-service attacks.

4 DASS DESIGN

CR checking can be modeled as a binary set query problem.

Definition 4.1 (Binary set query problem). Let U be a finite set of keys that can be divided into two disjoint subsets P and N, and $U = P \cup N$. The binary set query problem is that given $k \in U$, determine if $k \in P$ or $k \in N$.

All certificates that are checked for CR status are both time-valid and signature-valid, otherwise they will be rejected in expiration and signature checks. The IDM server knows all time- and signature-valid certificates (U) and they can be classified into to two finite sets: one for the legitimate certificates (‘negatives’ N) and the other for the revoked ones (‘positives’ P). Hence the CR checking result can be either 0 (not revoked) or 1 (revoked).

TinyCR achieves binary set queries by a compact data structure called DASS. We design DASS using an *innovative combination of existing algorithmic tools*. We first briefly introduce these tools.

Filter tools. A filter data structure is used for approximate membership queries. The most well-known tools are the Bloom filters [8] and Cuckoo filters [12]. For a given set S of keys, a filter F answers each query of key k and returns $F.\texttt{Query}(k) = 1$ if $k \in S$. However, filters introduce a small number of false positives: for a key $k \notin S$, the filter returns 0 in most cases but may also return 1. The space cost of a filter is proportional to $|S|$.

Using filters cannot meet the requirements of IoT CR. If we set N as S, then a revoked certificate may be determined as legitimate. If we set P as S, a legitimate certificate may be determined as revoked, which also brings problems. CRLite [19] uses a filter cascade to eliminate false positives. However, the cost to update a filter cascade is high as shown in Sec. 6.

Existing set query tools are not ideal. A set query tool can do exactly what we want for binary set queries of CR checking. It returns 1 if $k \in P$ and 0 if $k \in N$ for any $k \in P \cup N$. Recent compact set query tools include Bloomier filters [9, 10], Othello hashing [37], SetSep[40], and Coloring Embedder [33]. The space cost of a set query data structure is proportional to $|U| = |P| + |N|$.

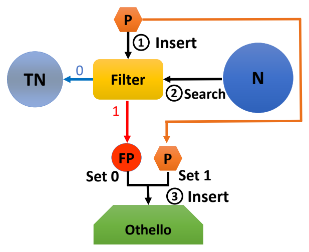


Figure 2: DASS Construction

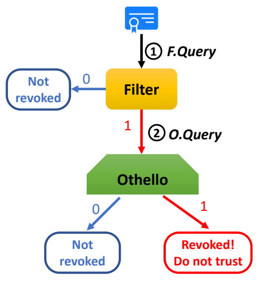


Figure 3: DASS Query

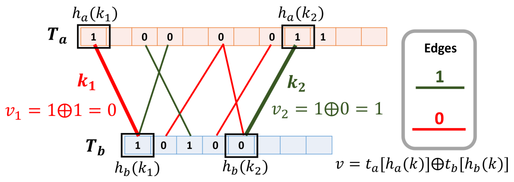


Figure 4: Othello for binary set query. The value of the key is stored by the edge between the two hash positions.

However these methods still introduce non-trivial memory cost for CR checking because $|N|$ is usually extremely large.

The key reason that DASS can further optimize the memory cost of existing set query tools is that we utilize the important observation: practical measurements show that the revoked certificates only contribute to 1% of all certificates [19, 34], hence $|N| \gg |P|$. DASS is particularly designed based on this fact.

Recall P is the set of revoked certificates and N is the set of legitimate certificates, and $|N| \gg |P|$. We construct DASS as shown in Fig. 2. DASS has two levels. The first level is a filter implemented by a Cuckoo filter [12] with $S = P$. Hence we create a Cuckoo filter F based on set P and insert all keys in P to it (Step 1) – each key is a certificate and is represented as a small number of bits in F. In Step 2, we test set N against the filter F. Most certificates of N will be tested 'negative' and they are true negatives (set TN). However a few certificates of N are tested 'positive' due to the fundamental limitation of a filter, and they are false positives (set FP). In Step 3, we construct an Othello data structure O for binary set classification and use FP as set 0 and P as set 1. Note both FP and P are very small sets compared N, hence DASS saves the majority memory cost compared to simply using Othello.

The query of DASS about a certificate k is executed as shown in Fig. 3. In Step 1, k is tested by the filter F. If $F.\texttt{Query}(k) = 0$, we must have $k \in N$ and k is legitimate. If $F.\texttt{Query}(k) = 1$ then k is either revoked or false positive. Then it is tested by Othello O. If $O.\texttt{Query}(k) = 0$, k is legitimate. If $O.\texttt{Query}(k) = 1$, k is revoked.

Compared to CRLite [19], DASS can be easily updated for a new certificate insertion or certificate revocation. Both the update time and message cost is very small. We will show this in later sections.

Tool Choices in DASS. As discussed in Sec.1, for CR checking, we majorly concern with the memory/computing efficiency for query and updating cost when the CRL changes. Although most of the state-of-the-art filter tools and set query tools mentioned above use similar resources for lookups when appropriately configured, their updating cost varies significantly. For example, Cuckoo Filter can support key deletion at a low cost, whereas standard Bloom Filter has to be rebuilt to remove a key from the membership set. We choose to use a (2, 4)-Cuckoo Filter for the first filter layer, namely, each item has two candidate bucket positions and each bucket has 4 available slots. The setting of two candidate positions is optimal for efficient query and updating, as the minimal numbers of hashing and memory read operations are required. Four slots per bucket can yields the minimal or close to minimal memory cost when the expected false positive rate ε is between 0.001% to 1% [12].

Among the above set query tools, Othello[37] is a dynamic data structure that supports new key-value pair insertion and value-flipping of existing keys at $O(1)$ cost, while the other methods are built on a fixed and static key-value set. Thus, we use Othello in our work for the optimal updating efficiency.

The structure of a one-bit Othello for the binary classification case is illustrated as Fig 4, in which each bucket of the hashing tables contains a one-bit slot. Suppose the lengths of the two hashing tables T_a and T_b are m_a and m_b, and the corresponding uniform hash functions are $h_a(x)$ and $h_b(x)$. Othello is constructed by finding an acyclic undirected graph $G = (V_a, V_b, E)$, where E is the edge set, V_a, V_b are the vertex sets with each node $v_a^i \in V_a$ $(0 \le i < m_a)$ and $v_b^j \in V_b$ $(0 \le j < m_b)$ representing the ith and jth bucket of T_a and T_b. For any key-value pair (k, v) with $k \in U$ and $v \in \{0, 1\}$, v can be stored in graph G by inserting a new edge $\left(v_a^i, v_b^j\right)$ in E, where $i = h_a(k)$ and $j = h_b(k)$ (as shown by the red or the green edges in Fig 4). The query function $f : U \to V$ for the key-value mapping is defined as: $\boldsymbol{Query}(k) = t_a[i] \oplus t_b[j]$, where $t_a[i]$ and $t_b[j]$ represent the entry in the ith and jth bucket of T_a and T_b respectively.

According to Yu et al. [37], it takes $O(1)$ time to find a proper pair of hash functions h_a, h_b that can successfully allocate the whole key set if the size of Othello $(m_a + m_b)$ is larger than $2.33|U|$, where $|U|$ is the size of the key set. In addition, using this setting, it takes $O(1)$ cost to query the data structure, and amortized $O(1)$ time to insert and update the data structure incrementally.

Therefore, in the DASS, we particularly use a variant of Cuckoo filter and Othello, which have been demonstrated as one of the most efficient filter/set-query tools for lookup and updating [28] to our best knowledge. We include the detailed preliminaries of Cuckoo filters and Othello implementation in Appendices A.1.1 and A.1.2. Note that the filter and set query components in DASS can be replaced with other alternative tools that satisfy the efficiency requirements of CR checking.

Tradeoff analysis of DASS. Despite its simplicity, DASS is rather memory-efficient to memorize the binary values of keys, especially when the sizes of the negative key set and positive key set are highly imbalanced (namely, set ratio $r = |N|/|P|$ is large). Here we show how to optimize DASS so that the total memory cost is minimized for the given two key sets P and N.

In DASS, there exists a trade-off between the sizes of the filter F and that of the Othello O. The false positives will be fewer if F uses more space, and hence O needs less space.

Let ε be the false positive rate of the F in the first layer, then the expectation of the number of false positives of F is $\varepsilon\,|N|$. Since Othello costs 2.33 bits per key, O needs $2.33\,(\varepsilon\,|N| + |P|)$.

Meanwhile, let the memory cost of the first layer Filter F be M_f, such that the expected false positive rate of F is no greater than ε. According to the recent implementation of Cuckoo filters [36],F produces a false positive result when the fingerprint of a negative key collides with at least one stored fingerprint in the two candidate buckets, with each bucket containing b entries. Therefore, the upper bound of the probability of a false positive fingerprint collision is $1 - (1 - 1/2^f)^{2b} \approx 2b/2^f$, where f is the number of bits of the fingerprint. Hence, $\varepsilon \geq 2b/2^f$, and we get

$$f \geq \lceil \log_2(2b/\varepsilon) \rceil = \lceil \log_2(1/\varepsilon) + \log_2(b) + 1 \rceil. \tag{1}$$

Then, the amortized space for each positive key stored in the filter is f/α, where α is the load factor of the Cuckoo hashing table. Thus, if we use the (2,4)-Cuckoo hashing table in F, and the expected load factor rate of 0.95 to initialize the Cuckoo Filter (which is a common setting for the filters to guarantee the success rate of insertion and efficiency of query), the amortized space for each positive key in F is $(\log_2(1/\varepsilon) + 3)/0.95$ [12, 36]. In addition, the Cuckoo Filter implemented with the semi-sorting trick [12] can further save one bit per fingerprint. Hence, the total cost of F with semi-sorting implementation is $|P|\,(\log_2(1/\varepsilon) + 2)/0.95$.

Let $r = |N|/|P|$. In total, DASS uses M bits where

$$M = \left(\left(\log_2(1/\varepsilon) + 2\right)/0.95 + 2.33\varepsilon r + 2.33\right)|P| \tag{2}$$

Since $|P|$ and r are constant for the given certificate sets, we can minimize the total memory cost by letting $\frac{\partial M}{\partial \varepsilon} = 0$. Hence, M is minimized when $\varepsilon \approx \frac{0.652}{r}$ and $M_{\min} = (1.05\log_2 r + 6.604)\,|P|$. The result further instructs us to set the fingerprints of the Cuckoo filter to be $\lceil 3.6 + \log_2 r \rceil$ bits according to Eq. 1.

Compared to the memory cost of Othello, $2.33(|N| + |P|) = \Theta(r|P|)$, DASS significantly reduces the memory cost to $\Theta(|P|\log r)$. The optimal filter cascade used in CRLite [19] costs $|P|(1.44\log_2 r + 4.2)$ bits, which is similar to DASS. But CRLite does not support *in-place* incremental updates.

5 PRACTICAL DESIGNS OF TINYCR

We present the detailed design considerations of TinyCR. The TinyCR system contains two programs: the *tracker* running on the IDM server and the *verifier* running on the devices. The tracker is responsible for receiving new certificates and revocations from the CAs, constructing DASS, and sending the DASS update messages to devices. The verifier is the compact DASS data structure running on the IoT devices to support CR checking. This section discusses how the tracker and verifier should execute and communicate.

5.1 Updates of Cuckoo Filter and Othello

Cuckoo filter supports key addition to and deletion from S by calling $F.\texttt{Insert}(k)$ and $F.\texttt{Delete}(k)$ respectively. Both functions cost constant time on average [12]. Othello supports key addition, deletion, and value flipping. Adding a key k to set 1 is by calling $O.\texttt{Insert}(k, 1)$, indicating the value of k is 1. Adding a key k to set 0 is by calling $O.\texttt{Insert}(k, 0)$, indicating the value of k is 0. Deletion and value flipping is by $O.\texttt{Delete}(k)$ and $O.\texttt{Flip}(k)$. All these functions cost constant time on average [37]. Due to space limit, we include details of these functions in Appendix A.2.

However, it is important to note that insertion and deletion of keys in Cuckoo Filters would impact the distribution of the potential false positive keys in the whole key space. More precisely, inserting a new fingerprint into the Cuckoo hash table would create a set of new potential false positive keys that match the fingerprint stored in the corresponding bucket. Similarly, deleting a fingerprint from the table would eliminate a fraction of potential false positive keys. For simplicity of the design description, we temporarily ignore this issue in Sec. 5.2. We then look back and discuss the solution to address this issue in Sec. 5.3.

5.2 Updating DASS on the Tracker

On-device DASS needs to be updated when 1) a new certificate is issued by CAs, 2) a certificate is revoked by CAs, 3) a certificate is expired, or 4) in rare cases CA un-revokes a revoked certificate. All these situations can be addressed by the following three update functions on the tracker.

- ***Insertion***: adding a certificate to N or P (very rare cases).
- ***Value Flipping***: moving a certificate from P to N (very rare cases) or from N to P.
- ***Deletion***: removing a certificate from P or N.

For each update, the tracker will compute the *delta message*, including only the bit positions that need to change for on-device DASS. Using the delta message instead of the complete DASS significantly saves bandwidth cost.

5.2.1 *Insertion.*

When a device joins the network with a new certificate, this information should be immediately reflected in DASS. Otherwise other devices may reject this certificate if DASS returns 1. In rare cases, the CA may also revoke a certificate before it is actually installed on any device.

Let k be the new certificate. If k is added to the positive set P, according to the design, k should first be inserted to the filter F and then inserted to the Othello O in the second layer with its corresponding value $O.\texttt{Query}(k') == 1$. On the contrary, if k is inserted to the negative set N, we can check whether F tests it as positive. If $F.\texttt{Query}(k) == 0$, then the original DASS classifies k correctly and no updating is required. Otherwise, k' is a false positive and should be inserted to O with $O.\texttt{query}(k) == 0$. Both $F.\texttt{Insert}(k)$ and $O.\texttt{Insert}(k, v)$ take $O(1)$ time to complete in average.

5.2.2 *Value Flipping.*

When a valid certificate is revoked by the CA if, for example, the device is compromised by an attacker, the revocation status of this key should be updated from 0 to 1 in DASS. In another case, the CA may also want to un-revoke a revoked certificate, implying the revocation status should be updated from 1 to 0. In both cases, all devices in the network should be noticed with the updating information to avoid abuse of the revoked certificates or mistakenly rejecting a legitimate one.

Suppose a key k is moved from N to P. The tracker first checks whether k is considered as a (false) positive key by the filter, then inserts k to the filter F. If k is a false positive, k has already been stored in the second layer O. In this case, the tracker needs to execute $O.\texttt{Flip}(k)$ to change the stored value of k. Otherwise, the tracker inserts k to O with corresponding value 1 by $O.\texttt{Insert}(k, 1)$.

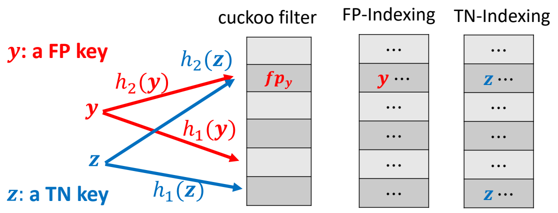


Figure 5: TN-indexing table and FP-indexing table

In another value flipping case, k is moved from P to N. In such case, k should have been already inserted in both F and O. Therefore, to update the DASS, the first layer filter F first removes k's fingerprint from its cuckoo hashing table and then check whether k would be recognized as a false positive key after removal. If k is not a false positive, it should be deleted from O. Otherwise, O flips the value of k using O.Flip(k).

5.2.3 Deletion. Certificates may expire. Although the removal of these certificates from DASS is not necessary – the expired certificates are rejected in early steps – it helps to maintain the DASS compact. DASS has to be rebuilt when it is too full to insert new certificates, which would cost considerable computation resources and network bandwidth. Hence, removing expired certificates can avoid unnecessary rebuilds.

Let k be the key that should be removed from either P or N. If $k \in P$, both of the two layers need to remove k by calling their delete functions. Otherwise if $k \in N$, we need check whether k is a false positive for the first layer F. If it is, then the second layer O needs to delete it. Otherwise, neither F nor O store the information k, thus no operation is required.

5.3 Handling Inconsistency of Updating

All above updating algorithms assume the false positive set of the first layer filter for the given certificate set remains stable. However, after inserting or deleting a key from the filter, the assumption may no longer hold, because the fingerprint added or removed from the cuckoo filter would increase or decrease the distribution of potential false positive keys.

If a former TN (true negative) key becomes a FP (false positive) key after an insertion, the key should be recorded in the second layer O, such that the key can be correctly queried. Similarly, if a former FP key becomes a TN key after a deletion, then the key needs to be removed from O. Although the correction process is simple, the detection of these influenced keys from the entire negative key set is challenging. A naive solution is thoroughly checking the negative key set with the updated cuckoo filter to find the influenced keys. However, this solution is extremely time-consuming as the negative key set is usually big, causing $O(|N|)$ rather than $O(1)$ updating cost in the worst case.

In TinyCR tracker, we propose to solve the problem by using two additional indexing hash tables that have similar number of buckets as the cuckoo filter to index the sets of the potentially influenced keys for every fingerprint in the cuckoo filter.

Specifically, at the construction time of DASS, when we iterate through the entire negative set N to find the FP sets by querying F, we insert the TN keys into the “TN-indexing" hash table and FP keys into the “FP-indexing" hash table at the exact two bucket positions that are queried in F to lookup the fingerprint (as shown in Fig 5). Therefore, when a fingerprint is inserted into a particular bucket in F at the updating time, only the TN keys stored at the same bucket positions of the TN-indexing table would be potentially influenced by the insertion. Hence, only these TN keys need to be queried with F again to check whether they become FP keys after the insertion. Then those new FP keys are inserted to the O in the second Othello layer. Similarly, when a fingerprint is deleted, only the FP keys at the corresponding buckets in the FP-indexing table need to be checked again. Then the keys that become TN keys after the deletion are removed from O.

Since $|N| = r|P|$ and the number of buckets in F is $O(|P|)$, the amortized length of each bucket in FP-indexing and NP-indexing is $O(r)$. Thus, the updating cost decrease from $O(|N|)$ to $O(r)$ in worst case with this indexing strategy. Meanwhile, the total size of the indexing tables is $O(|N|)$. **Since these tables are maintained by the server and not related to the devices, the cost is affordable.** By properly handling the inconsistency issues, the tracker is able to create a perfect DASS that yields ***zero*** query error.

5.4 Updates on Devices

Though the TinyCR tracker requires $O(|N| + |P|)$ extra space to maintain the certificates, each on-device verifier requires much less memory and computational resources to support updating. In the verifier, only the cuckoo filter and Othello are stored in memory, costing approximately $(1.05 \log_2 r + 6.604) |P|$ bits. The inference of DASS in verifier can be simply accomplished by at most four hashing and memory read operations.

In addition, the DASS verifier can also be synchronized with *delta messages*. When an update is necessary, the tracker sends a delta message patch to all devices. The delta message includes the certificate digest and the indexes of the bits that need to be changed in O's hash tables, and is small in size (9 to 150 bytes on average for 100 million certificates). Note that the indexes of the flipped bits in O are tracked as an intermediate result while updating the Othello (see Appendix A.2.2). Thus, there is no extra cost to compute the indexes after the update is done. Our experiment also shows the raw delta message does not scale with the size of the certificate sets. Then the tracker **signs the delta message** and attach the signature to the updating patch data to guarantee the integrity. This updating strategy differs from other CR checking synchronization methods that use static data structures, such as CRLite [19], which needs to rebuild the entire data structure for every update (if correctness of verifier is obligatory at any time) and sends it to all clients. The raw delta message of CRLite is much larger than that of DASS.

In our design, the raw delta-msg is encoded as Fig 6. Specifically, the updating instruction for F uses only 9 bytes, including 1 byte for the operation type (insert, delete or do nothing) and 8 bytes for the 64-bits digest of the certificate. Then the F in the verifier DASS can insert or delete the certificate through the corresponding operations of the local Cuckoo filter.

Meanwhile, the updating instruction for the O is a list of 32-bit integers representing the bucket positions at which the stored value should be flipped. For every position index pos, if $pos \leq |T_a|$, we flip

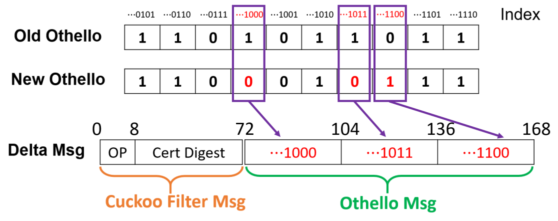
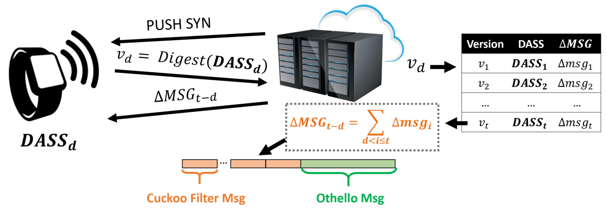


Figure 6: Structure of a delta message


PUSH SYN
$v_d = Digest(\boldsymbol{DASS_d})$
ΔMSG_{t-d}
$\boldsymbol{DASS_d}$
v_d
$\Delta MSG_{t-d} = \sum_{d<i\leq t} \Delta msg_i$
Cuckoo Filter Msg
Othello Msg

Version	DASS	ΔMSG
v_1	$\boldsymbol{DASS_1}$	Δmsg_1
v_2	$\boldsymbol{DASS_2}$	Δmsg_2
...	...	...
v_t	$\boldsymbol{DASS_t}$	Δmsg_t

Figure 7: Multi-way version control protocol.

the entry at bucket pos of T_a; otherwise, we flip the entry at bucket $pos - |T_b|$ of T_b, where T_a and T_b are the two maintained hash tables in Othello [37]. In our evaluation, we will show on average only a small number of buckets in O (if any) need to be flipped.

5.5 DASS Version Control

Since TinyCR uses delta messages to update the on-device checker, the new state of DASS relies on the previous state. Thus, the system may suffer from potential system/network failures that cause the packet loss of the delta messages. To solve this problem, we introduce a multi-way DASS version control protocol as an optional design choice(as illustrated in Fig. 7).

In Fig. 7, the IDM server initiate a *PUSH-SYN* packet when a new tracker $DASS_t$ is generated. Then the device sends back the digest v_d of its local verifier $DASS_d$. Meanwhile, the IDM server maintains a mapping table to keep track of a history of t recent verifier DASS version IDs and the corresponding delta-msg increments. According to our evaluation in Sec. 6.4.2, the average delta-msg increment size is fewer than 100 bytes. Then the IDM server simply retrieves all the missed delta-message increments and concatenates them to generate the cross-version delta message ΔMSG_{t-d} that denotes the differences between $DASS_d$ and $DASS_t$. In the ΔMSG_{t-d} that skips over multiple versions, we could include multiple Cuckoo Filter Msg fields and one single Othello Msg field using the similar encoding format as shown by Fig. 6. If v_d is not maintained by the version table, that means the device has missed a large amount of updates. Then the server directly send the $DASS_t$ instead of the delta message to the device. Optionally, the device returns an ACK when the local DASS updating is accomplished.

If the updating frequency of certificate sets is too high in some scenarios, it is not practical for the IDM server to send a signed delta message after each update and track every DASS version. In such case, we can use the version control design to batch the updates with a bounded time granularity. For example, the IDM server can only send one single aggregated delta message in per-hour, and maintain only 24 delta message increment versions in each day.

6 IMPLEMENTATION AND EVALUATION

6.1 System Implementation

We implement the TinyCR tracker on a Google Cloud VM instance with 64 vCPUs and 624 GB memory using C++. The on-device DASS verifier is implemented on a Raspberry Pi 3 with one single 1.4 GHz processor and 1 GB RAM. Note the device used in the experiments is just an example of **a wide spectrum of devices that can use TinyCR**. TinyCR can be easily deployed on more powerful devices like mobile phones and less powerful devices as long as they have available memory (see Table 2. For real Censys data that contains 28.6M certificates, it requires 448KB).

In addition to TinyCR, we also implemented the CRLite filter cascades [19] and Othello hashing [37] data structures with similar synchronization settings as the TinyCR protocol for performance comparison. The parameters for CRLite and Othello are set according to the authors' suggestions [19, 37]. Both TinyCR and Othello can support dynamic updating of the revocation checking list, while CRLite has to be rebuilt for most updates.

6.2 Metrics and Dataset

We evaluate the CR methods by the following metrics:

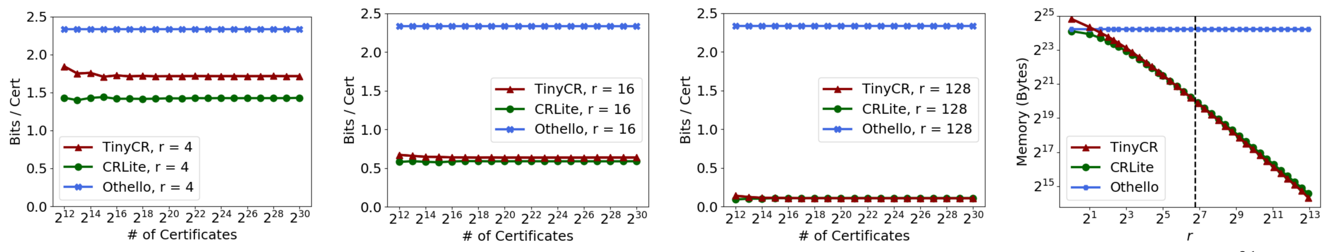
- On-device memory cost: the overall memory cost of the data structures on a device.
- Update time and synchronization latency: the time for each update on the IDM server.
- Bandwidth: the message cost caused by updates.
- Query cost: the delay to get a CR checking result.

We use both real-world and synthetic certificate datasets for the evaluation. Since there is no IoT certificate dataset available, we use the Censys web certificate dataset[4, 19] to evaluate how those protocols perform in real-world CR verification scenarios. We downloaded 30 millions items of historical NSS trusted certificates over 3 months from Censys using Google BigQuery [5]. After removing the duplicated certificates, there are totally 28,593,752 items in the dataset. Then we use the CRLs or OCSP to obtain the revocation status of all downloaded certificates. Among the 28.6 million certificates, 274,926 were revoked, i.e., the ratio between the legitimate and revoked certificates is 103 : 1. To evaluate the scalability, we create synthetic datasets containing up to 1 billion certificates with different revocation ratios from 20% to 0.01%.

6.3 Memory cost

We construct the on-device data structures of TinyCR (DASS), CRLite (filter cascade), and Othello respectively using the entire Censys certificate data. We find TinyCR, CRLite, and Othello requires 430 KB, 439 KB, and 8,328 KB memory respectively to maintaining the CR status of the 28.6 million certificates.

Then we conduct experiments on the synthetic dataset to investigate how the memory sizes scale with the sizes and distributions of the keys. In Fig. 8, we show the amortized memory cost (i.e. bits per certificate) with respect to the total size $|N| + |P|$ of the certificates by setting $r = |N|/|P|$ as 4 (Fig 8 a), 16 (Fig 8 b) and 128 (Fig. 8(c))



(a) $r = 4$ (b) $r = 16$ (c) $r = 128$ (d) $|P| + |N| = 2^{26}$

Figure 8: Plot (a) to (c): Amortized memory for different key sizes when $r = |N|/|P|$ is 4, 16, 128 respectively. Plot (d): Memory cost for 2^{26} keys with respect to r.

respectively. Meanwhile, in Fig. 8(d), we present the total memory cost (in bytes) for storing the revocation status of 2^{26} certificates, by varying the ratio r. The vertical dash line in Fig. 8(d) represents the ratio r of the Censys dataset in real-world scenario.

Fig. 8 shows the memory cost per certificate of all three data structures keeps stable when r is fixed. For example, the amortized memory sizes for TinyCR, CRLite, and Othello are around 0.108 bits, 0.111 bits, and 2.333 bits per certificate respectively for arbitrarily large key sets when $r = 128$. The amortized memory for Othello is independent with r (it is controlled by a hyper-parameter and is set as 2.33 bits), whereas both TinyCR and CRLite use much less memory as r grows. It can also be seen from the graph that both TinyCR and CRLite use less than 1 MB to store the around 64 million certificates when $r = 100$ (which is close to the ratio for real-world CR lists) and use less than 8 MB when $r = 10$, while Othello always requires around 20 MB.

6.4 Updating efficiency

In this section, we evaluate the update and synchronization overhead of the data structures regarding any change of the global CRL. Specifically, we utilize the Censys certificates and synthetic data sets to simulate the following updating scenarios.

Short-term insertion/value flipping: We use a certificate dataset to initialize the CR verification data structures through a static approach, then evaluate the latency of the inserting/value flipping operation on the initial data structures without reconstructing the data structures (except for filter cascades).

Long-term insertion: We use 100 million certificates to initialize TinyCR, then insert another 100 million certificates item by item to them. In the simulation, we assume the revocation ratio of the initial and the inserted certificate sets are consistent.

Long-term value flipping: We use 100 million certificates to initialize TinyCR. Then we randomly sample $|P|$ validate certificates and revoke those certificates, where $|P|$ is the number of revoked certificates in the initial set. We simulate the scenario where the number of revoked certificates is doubled during the usage period before expiration. Note that the revocation of the sampled set is a gradual process, i.e., one certificate is revoked at each timestamp when the CA decides to revoke it. In TinyCR, the reference value of a newly revoked certificate should be changed from 0 to 1.

6.4.1 Overhead on the IDM server (tracker).

The tracker on the IDM server is required to react quickly for every update (insertion and value flipping) of the CRLs. In Table 2, we show the on-device memory cost and the average computational latency of the tracker to update the data summaries and generate the delta message in short-term updating scenarios. Specifically, we simulate the scenarios with the Censys dataset and the synthetic data sets of different sizes to evaluate the scalability of the methods in an IoT with billions of devices. In our synthetic data, we set the certificate revocation ratio to be 1%, which is close to the ratio of the Censys dataset. We discuss the insertion of revoked certificates and legitimate certificates (the more common case) separately in the fourth and fifth columns, as they will cause different updating overhead based on the algorithms. Similarly, we also evaluate the value flipping case where a revoked certificate is moved to the legitimate list, and the case where an legitimate one is moved to the revoked list (the more common case) in sixth and seventh columns respectively.

# of Certs	Method	Mem	Add P	Add N	$P \rightarrow N$	$N \rightarrow P$
Censys 28.6M	CRLite	458 KB	3.2 s	3.2 s	3.2 s	3.2 s
	Othello	8.3 MB	11.4 μs	9.9 μs	10.1 μs	9.2 μs
	TinyCR	448 KB	349.9 μs	1.6 μs	27.0 μs	345.3 μs
10M	CRLite	172 KB	1.0 s	1.0 s	1.0 s	1.0 s
	Othello	2.9 MB	4.6 μs	5.0 μs	4.6 μs	4.4 μs
	TinyCR	169 KB	280.9 μs	1.2 μs	16.6 μs	289.9 μs
100M	CRLite	1.7 MB	10.1 s	10.1 s	10.1 s	10.1 s
	Othello	29.2 MB	8.5 μs	7.5 μs	7.1 μs	7.0 μs
	TinyCR	1.7 MB	304.9 μs	1.6 μs	21.6 μs	311.5 μs
1B	CRLite	17.2 MB	153.9 s	153.9 s	153.9 s	153.9 s
	Othello	291.7 MB	10.0 μs	10.2 μs	8.2 μs	7.0 μs
	TinyCR	16.9 MB	296.0 μs	2.7 μs	27.3 μs	319.5 μs

Table 2: On-device memory cost and average updating latency on the tracker for different set sizes. The revocation ratio for synthetic data is 1%.

From Table 2, we find the updating time of CRLite significantly increases with the size of the sets. As a static data structure, filter cascades have to be reconstructed using the entire certificate sets for any updates, which would cause tremendous overhead to the server and large bandwidth overhead. Meanwhile, the long latency of updating can also cause memory concurrency issues for the tracker when the updating pace is high. Therefore, in practice, CRLite is only updated in a batching way, for example, the tracker and verifier are recommended to update once every day [19]. Consequently, this strategy would introduce a synchronization latency of one day – a big security vulnerability. The synchronization latency of TinyCR is the update time plus network latency.

On the other hand, the update latency of TinyCR and Othello is significantly lower than CRLite and scales much better with the size of certificate sets. Overall, Othello achieves the highest updating throughput for most cases, at the cost of around 16x more memory than TinyCR and CRLite. We also notice TinyCR is most computational-efficient for inserting legitimate certificates to the CR status list, which is the most common type of updating. Even in its worst case, the corresponding updating latency is smaller than 1 millisecond for up to 1 billion keys, which is usually overwhelmed by the network latency in practice, showing TinyCR can sufficiently support the real-time synchronization with neglectable extra processing overhead. Thus, TinyCR is a more efficient and secure choice for the IoT CR verification task where the certificate universe is large. The theoretical synchronization latency of TinyCR could be just the update time plus network latency in a real-time updating manner.

However, due to the connection maintenance and signing cost in practice, real-time updating is not always practical when the updating frequency is too high. The recommended practical deployment settings and analysis are presented in Sec. 6.6.

6.4.2 Delta Message Size. The IDM server of TinyCR requires to send updating messages to all devices, so that the devices can update their own CR status classifier locally. Therefore, the delta message size is a critical metric, as a large message size would significantly increase the network traffic overhead and transmission latency.

In Fig. 9, we show the average raw delta message size for each type of updating operations of TinyCR, Othello and CRLite in the short-term updating scenarios using the Censys certificate data. Note that in short-term updating scenarios, we conduct limited numbers of updates such that the data structures (except CRLite) are not reconstructed. For inserting legitimate certificates, TinyCR and CRLite usually do not need to be updated as the certificate key is highly likely to be rejected by the first filter layer. For other cases, we notice the delta message sizes of TinyCR and Othello do not scale with the growth of key sizes for all types of the insertion and value flipping operations. Specifically, both TinyCR and Othello requires around 0 to 100 bytes of the delta message for all different types of updates (though Othello requires 16x more total memory), whereas CRLite requires to push a significantly larger message to all IoT devices. In addition, for the most common certificate insertion operation shown in Fig. 9 (b), TinyCR do not need to send any delta message to devices for most of the insertions (the average delta message size is around 0.1 bytes), whereas Othello has to synchronize a delta message for most of the cases.

In Fig. 10, we show the distribution of the raw delta message size (without the signature) in the long-term insertion and value flipping scenarios. In these scenarios, when DASS is too full to support the desired update, it has to be reconstructed In the figure, the top of each bar in the figure represents the $90th$, $99th$, $99.9th$ percentile of the delta messages sizes.

For the long-term insertion scenario in Fig. 10 (a), the result shows more than 90% and 99% of the delta messages are equal to 0 bytes when the ratios of the legitimate and revoked certificate sizes ($|N|/|P|$) are 100 and 1000 respectively. Namely, for most of the insertions, the verifier DASS do not need to be updated.

In some rare cases, TinyCR can no longer accommodate a space for the new key. Then the DASS need to be reconstructed on the server and then be pushed and reinstalled on the IoT devices. Therefore, a reconstruction of the data structure would cost much higher overhead on both devices' computing resources and network bandwidth. In the experiments, we notice the total times of DASS reconstruction are 44, 31, 28 respectively to insert the 100 million new certificates, when $|N|/|P|$ equals 10, 100 and 1000. On average[*], the bandwidth costs of raw delta messages (not including the signatures) for each insertion are only 12.2, 1.25 and 0.13 bytes when $|N|/|P|$ equals 10, 100 and 1000.

The long-term value flipping result in Fig. 10 (b) shows that revoking an existing certificate costs more bandwidth in TinyCR compared with the insertions. Specifically, most revocation events will trigger an updating of the verifier DASS and more than 90% of the updates need a delta message smaller than 65 bytes for all the three scenarios with different revocation ratios. In addition, less than 1% revocations will cost more than 385 bytes and less than 0.1% revocations (including the cases that require a reconstruction) will cost more than 1 KB for the delta messages. In total, DASS is reconstructed for 64, 31, and 29 times in order to randomly revoke another around 10M, 1M and 0.1M legitimate certificates in the three 100M sets with different initial revocation ratios. The average delta message size[†] for the tree scenarios are 150.58, 108.08 and 119.87 bytes in the three value flipping scenarios.

In summary, TinyCR only needs 0 to 150 bytes on average for any CRL update. Since nearly all types of wireless IoT data links (including Licensed/Unlicensed LPWANs and 3G/4G/5G Cellular, etc.) can provide larger than 1KBps bandwidth in practice, the TinyCR synchronization process introduces a neglectable extra data transmission cost to the overall network latency.

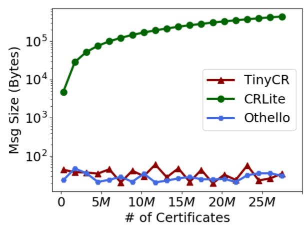
6.5 Query

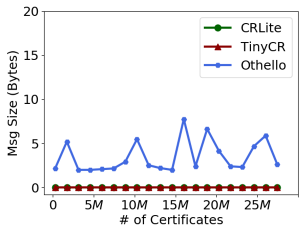
The IoT devices that have installed the DASS verifier would be able to check the CR status of a particular certificate after validating the integrity and expiration date of the certificate. Standard certificate integrity validation requires cryptography computation. Recent works introduce delegated or distributed reference protocols based on the chain of trust [6, 26], which still requires at least millisecond-level latency. Compared with the validation process, the latency for the revocation status checking process using the TinyCR verifiers is neglectable (usually in sub-microseconds).

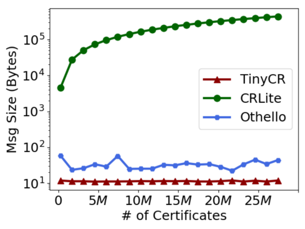
In Fig. 11, we test the average query latency to get a revocation status using the CenSys dataset on the Raspberry Pi 3 testbed and compare the result with CRLite and OCSP. For OCSP, we use a local 8-core CPU server deployed in the local town as the OCSP server. In addition, on the server side, we use DASS instead of the whole CRL to maintain the CR status. As the on-device CR verifiers, TinyCR and CRLite can verify a CR status in sub-microsecond level, which is a few magnitudes faster than OCSP, as both of them only require $O(1)$ hash operations and memory loads for checking. In

[*]the cost of the reconstruction cases is amortized to every insertion

[†]the cost of the reconstruction cases is amortized to every revocation event


(a) Insert (P)


(b) Insert (N)


(c) Flip ($P \longrightarrow N$)

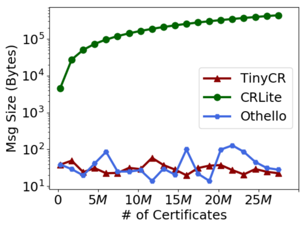
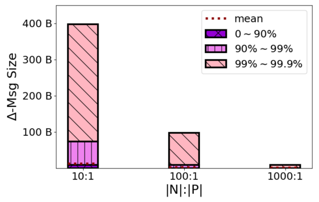

(d) Flip ($N \longrightarrow P$)

Figure 9: Short term insert and value flip delta message size. (a) Insert a revoked certificate. (b) Insert a legitimate certificate, a common operation. (c) Unrevoke a revoked certificate. (d) Revoked a certificate, a common operation.


(a) Insert

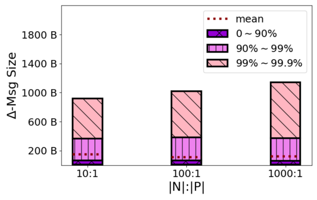
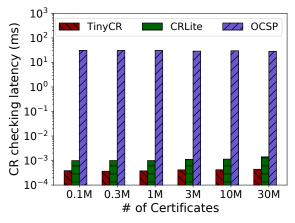

(b) Flip

Figure 10: The average, and the 90th percentile, 99th percentile and 99.9th percentile of the generated Delta Msg sizes for long-term insertion (a) and value flipping (b).


(a) Revoked certificates

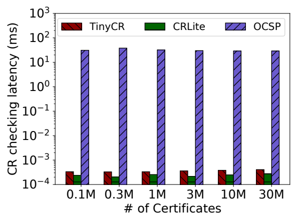

(b) Legitimate certificates

Figure 11: Query latency on Raspberry Pi 3.

particular, the query delay of TinyCR is slightly shorter for the revoked certificates, while the delay of CRLite is slightly shorter for the legitimate certificates.

The major query cost for OCSP is the network delays when inquiring the CR status through a remote server. Thus, OCSP is not an ideal method for the scenarios where the device available bandwidth is limited and the latency is sensitive.

More results of the query performance of DASS are shown in Appendix. A.3.

6.6 Bandwidth vs. Dynamics

In Figs. 12 and 13, we show the delta message cost (each patch includes a 256-byte RSA signature) for keeping the verifier DASS synchronized under different updating scenarios and settings. Specifically in our experiments, we initialize DASS with 100 million certificates, with 1% revoked keys. Then we test two updating scenarios with different daily workloads: (1) 1 to 10^8 new certificates are added to the certificate universe; (2) 1 to 10^7 existing certificates are revoked. In the experiments, we assume the updates happen uniformly over the day. We deploy four different settings for TinyCR: TinyCR-(RT) sends the delta message immediately after each update happens; TinyCR-(1, 4, 24) means we only maintain 1, 4, 24 versions of TinyCR per day and use batching as in Sec. 5.5. Hence, the synchronization latency for TinyCR-(1) is up to one day and for TinyCR-(24) is up to one hour. Similarly, we implement the corresponding versions of CRLite as comparisons. The CRLite is updated for 1, 4, 24 times per day using a bsdiff [25] delta update message. The initial on-device memory costs of TinyCR and CRLite under this setting are both 1.7 MB. We also compare the protocols with OCSP, which has zero update cost on bandwidth and the device side but generates relatively constant traffic load (around 1KB according to prior measurement studies [19, 21]) for each query. Thus, on the y-axis in the right, we show the estimated number of OCSP queries that can be made using around the same amount of traffic load needed by the daily updating of TinyCR and CRLite.

From Figs. 12 and 13, we can clearly observe that TinyCR costs less bandwidth by a few orders of magnitudes compared to CRLite, when the daily updating amount is moderate (for example, less than 1 million inserts or less than 1 thousand revocations per day). On the other hand, when the amount of daily updates is huge, TinyCR has similar total bandwidth cost as CRLite. More specifically, all versions of TinyCR have a similar raw delta message cost if DASS is not reconstructed, while the real-time TinyCR always causes more real-world traffic load due to the high cost of signing the delta messages. When the number of updates is large and DASS has to be reconstructed multiple times, the batching protocol with fewer batches has less bandwidth cost, since at most only one reconstructed and signed data structure needs to be sent in one batch. On the other hand, CRLite always needs a large delta message for synchronization whenever a false positive is found in its first layer of the filter cascades. The total message size of CRLite is in proportional to the updating frequency. When the daily update amount is huge, for example, the certificate universe is doubled or more than 10,000 certificates are revoked per day, CRLite has a similar performance as batching TinyCR. In particular, with higher batching frequency, TinyCR is more efficient; while with lower frequency, CRLite is a better choice.

In addition, from Figs. 12 and 13, we find that the TinyCR cost is proportional to the number of updates while the cost of OCSP is proportional to the number of queries. Note that the TinyCR

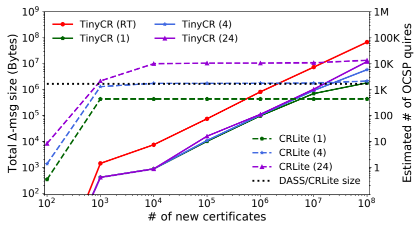


Figure 12: Total bandwidth cost for insertion.

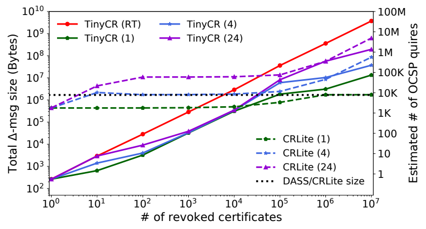


Figure 13: Total bandwidth cost for revocation.

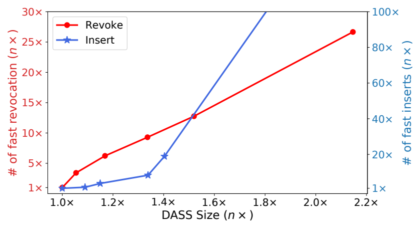


Figure 14: How many updates can be applied before the first rebuild.

mainly consumes the downlink bandwidth while the OCSP mainly consumes the uplink bandwidth. Thus, it is easy to conclude that TinyCR is more bandwidth cost-efficient when the certificate universe and daily updating amount is small and querying is frequent, while OCSP is more cost-efficient in the opposite scenarios.

6.7 Mitigate rebuilds

When TinyCR has to rebuild, the delta message and server resource cost is significantly higher. Therefore, if the CRL is rather dynamic, we could further optimize DASS to make it less likely to be rebuilt. If the certificate universe is smaller than what the devices can maintain with their memory capability, we could choose to slightly increase the DASS size to reserve spaces for future new certificates and revoked certificates. In particular, the two most important parameters that impact the probability of rebuild is the load factor α of the Cuckoo Filter and the table size coefficient β of Othello. The two parameters are set as $\alpha = 0.95$ and $\beta = 2.33$ recommended by the original studies [12, 37] to optimize memory. Thus, if memory allows, a smaller α and a larger β can be used to reduce the probability of rebuild.

In Fig. 14, we show how many updates can be handled by DASS without rebuilding under different memory cost settings. The initialization setting is similar to the setting in Sec. 6.4.2. In the x-axis, $n\times$ means the memory cost of DASS under different settings compared with that of the memory-optimal setting, while in the y-axis, the $n\times$ means how many updates (insertion or revocation) can be processed without rebuilding, compared with that of the memory-optimal setting. For the memory-optimal setting, a rebuild will be triggered after 22 million insertions or 23 thousand revocations in average. From Fig. 14, we can find the capability of accommodating the updates can be significantly improved by increasing the memory cost slightly. For example, by using 1.5× memory, DASS can process more than 13× new revocations or 30× new insertions without reconstructions. This memory allocating strategy is rather effective for keeping the $O(1)$ updating cost in real deployment.

7 APPLICATION SCENARIOS FOR TINYCR

Based on evaluation results, TinyCR is ideal and optimal for the application scenarios where 1) users need fast or frequent on-device authentication, and low synchronization latency for security; 2) each user device has a limited size of memory, such as several MBs; 3) the dynamics of certificates are moderate. In addition, for other scenarios, TinyCR can be used as an alternative with proper configurations or as a complementary of other protocols.

Batching v.s. Real-time Updating.

Based on our analysis in Fig. 12 and 13, the real-time TinyCR updating policy is the optimal choice when CR updates are infrequent. This policy can minimize the synchronization delay and protect devices at any time with limited bandwidth cost. However, due to the high overhead of signing for the delta message, real-time TinyCR yields a high cost when the updating frequency is too high. Thus, we could choose the batching policy for TinyCR and keeping the updating frequency high enough (such as per hour or per 5 minutes) to trade between bandwidth cost and the worst synchronization latency. According to our results using batching, higher but bounded updating frequency does not introduce more bandwidth overhead other than the extra $O(1)$ signing cost. The batching policies are also friendly to the IDM servers if most IoT devices are sporadically connected, as it only needs to maintain a bounded number of DASS versions. In addition, DASS can use a slightly higher memory cost to reduce the reconstruction probability in practice.

When the certificate universe changes significantly every after a short period, TinyCR as well as all other push-based methods will have unacceptable bandwidth cost to keep the synchronization latency low. In such a scenario, we have to sacrifice security for efficiency by reducing the updating frequency, and CRLite is more efficient for one update per day. The on-demand-based methods (such as OCSP) are the other optional choice under this scenario despite its higher verification cost.

Moreover, if the CRL updates are non-uniformly distributed over the data and are predictable by the service providers, we can use a hybrid policy with the batching protocols and real-time protocol. For example, we can batch the updates in the peak hours when updating is frequent, and use the real-time protocol for the rest of the hours when the updating is sporadic.

TinyCR v.s. OCSP/OCSP-stapling.

TinyCR outperforms OCSP in that it is much faster for CR verification. In some IoT scenarios, the verification delay is critical since IoT devices usually have limited data (such as sensing data) to transmitted and the requirement for end-to-end data transmission delay is tight. For example, a smart vehicle is required to read the IoT sensors on streets for decision-making in a short delay while driving fast. In addition, OCSP is not suitable for many IoT applications as it leaks user privacy. This drawback becomes more severe as the IoT data access pattern may include not only the temporal context but also the location information of the user, such as when and where a user reads a static street sensor. Besides, many peer-to-peer communication patterns for IoT usually do not need access to the Internet, for example, IoT devices can be accessed using short-distance communication media, such as WLAN, Bluetooth, and visible light. Hence, on-device CR checking protocols are more suitable for those scenarios. Still, for the rare cases when a new certificate cannot be verified by an outdated DASS, we can choose to fall back to OSCP.

OCSP-stapling is another practical design for CR checking in IoT scenarios as it does not rely on server access upon verification and can protect user's privacy. The major difference between TinyCR and OCSP-stapling is that TinyCR requires the device who verifies the other device to maintain DASS, while OCSP-stapling requires the device who is under verification to provide the time-stamped OCSP response extension. These OCSP-stapling devices have high bandwidth overhead. Thus, in IoT scenarios, if the device who needs to authenticate the other device has more memory/network resource (for example, a smartphone is required to authenticate a sensor), TinyCR is a better choice as it only requires the inquiring device to maintain an up-to-date DASS. On the contrary, if devices to be authenticated are more powerful (for example, a sensor needs to authenticate a smartphone), then OCSP-stapling can be used. If bi-directional authentications are necessary, we can use a hybrid method of DASS and OCSP-stapling to optimize the resource-security trade-off.

8 SECURITY ANALYSIS

We discuss the following attacking behaviors for TinyCR.

(1) *The attacker attempts to masquerade as a legitimate IoT client by using a revoked certificate.* Since the synchronization latency of TinyCR is only on the millisecond level plus the network latency, the attacker has very limited time to conduct such attacks. Compared to prior work that synchronize the devices on daily basis [19, 29], TinyCR significantly reduces the chance of this attack. Note that it is also important for a CA to detect a comprised certificate as early as possible, although this topic is not the focus of this paper.

(2) *The attacker performs the MitM attacks between the IDM server and the IoT devices.* The current methods are sufficient to defend against MitM attacks between the IDM servers and the IoT devices. Each device can get the public keys of the IDM servers and CAs using offline methods during manufacturing or installation. With the public keys, the device can build trusted TLS sessions to IDM servers. Hacking an IDM server or a CA requires much more attacking power than hacking a device. In this paper, we do not consider the scenario where the IDM server is hacked.

(3) *The attacker attempts to manipulate the CRL, DASS or a delta message.* Since the CA-IDM channel can use trust TLS sessions, the integrity of the CRL can be protected. In addition, since the DASS messages are signed by the IDM servers, the attacker cannot manipulate the DASS installation or updating patches.

(4) *The attacker wants to infer private information of other devices, servers, or CAs from the TinyCR install and update messages.* An attacker can easily obtain the TinyCR install and update messages by compromising just one device. However, knowing these messages give the attacker no advantage because the CR information is public. DASS is not constructed for each particular device hence there is no device private information in the DASS messages.

(5) *The attacker can block the update messages between an IoT device and the IDM server, then use a revoked certificate to attack that device.* TinyCR has no specific design to prevent the attacks of blocking the communication to a device – and no other CR method does. However, it is possible to detect such attacks. For example, the IDM server can send heartbeat packets to the devices periodically with the digest of the up-to-date DASS verifier and the current time. If the device does not hear the heartbeat after a period of time, it may detect such communication-blocking attack.

(6) *A compromised IDM server sends wrong DASS information and update messages.* All DASS install and update messages can be easily audited by another IDM server that knows all certificates and the revoked ones. "Auditable" means any party who knows the entire CRL can verify if another DASS version is maliciously modified. The device can forward the DASS messages with signatures to other IDM servers for auditing. The IDM servers can use their maintained certificate universe and the CRL to test the integrity of the DASS. If the DASS information is tempered, the other IDM servers can easily find the malicious IDM server by the signature.

(7) *The adversary acquires and causes a revocation with a strategy to trigger frequent rebuilds of DASS.* An attacker could learn which certificate revocation will trigger a rebuild of DASS (by running a simulation experiment) and then attacks that particular certificate and causes it to be revoked by the CA. To defend against such an attack, we can preallocate extra space in DASS to make it capable of learning more updates without rebuilding and reduce the probability to find a certificate that triggers a rebuild. From our analysis in Fig.14, we find this strategy is effective to defend the attacker with limited power. For example, by doubling the size of DASS upon initialization, it becomes more than 20 times harder to find a certificate that will trigger a rebuild.

9 CONCLUSION

TinyCR is a new system and protocol to allow on-device CR checking for IoT. We develop DASS, a compact and dynamic data structure, to maintain the CR status of the entire IoT network, which costs each device very small memory. We also implement the two communication components of TinyCR: the tracker that run on an IDM server to construct and update DASS and sends the update messages to devices, and the verifier that can synchronize with the tracker and be queried for the CR status on IoT devices. The experiments show that TinyCR costs small memory, short CR checking time, low network bandwidth, and low synchronization latency.

10 ACKNOWLEDGMENTS

The authors were partially supported by NSF Grants 1717948, 1750704, 1932447, and 2114113. C. Qian was partially supported by the Army Research Office and was accomplished under Grant Number W911NF-20-1-0253. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Office or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein. We thank all CCS reviewers for providing the valuable comments that help us improve the paper.

REFERENCES

[1] 2016. An Internet of Things Reference Architecture. White Paper, Symantec.

[2] 2019. Why Digital Certificates Are Essential for Managing Mobile Devices. White Paper, DigiCert, Symantec's Website Security business.

[3] 2020. TinyCR source code. https://github.com/jonnekaunisto/TinyCR.

[4] 2021. Censys. https://censys.io/certificates. Accessed: 2019.

[5] 2021. Google BigQuery. https://cloud.google.com/bigquery. Accessed: 2019.

[6] Arwa Alrawais, Abdulrahman Alhothaily, Xiuzhen Cheng, Chunqiang Hu, and Jiguo Yu. 2018. Secureguard: A Certificate Validation System in Public Key Infrastructure. *IEEE Transactions on Vehicular Technology* 67, 6 (2018), 5399–5408.

[7] Luigi Atzori, Antonio Iera, and Giacomo Morabito. 2010. The Internet of Things: A Survey. *Computer Networks* 54, 15 (2010), 2787–2805.

[8] Burton H Bloom. 1970. Space/Time Trade-offs in Hash Coding With Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426.

[9] Denis Charles and Kumar Chellapilla. 2008. Bloomier Filters: A Second Look. In *In Proceedings of the European Symposium on Algorithms (ESA)*.

[10] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. 2004. The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables. In *In Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 30–39.

[11] Donald Eastlake et al. 2011. *Transport Layer Security (TLS) Extensions: Extension Definitions*. Technical Report. RFC 6066, January.

[12] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *In Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT)*.

[13] Mark Goodwin. 2015. Revoking Intermediate Certificates: Introducing Onecrl. *Mozilla Security Blog* (2015).

[14] Phillip Hallam-Baker. 2015. X. 509v3 Transport Layer Security (TLS) Feature Extension. *RFC 7633* (2015).

[15] Russell Housley, Warwick Ford, William Polk, and David Solo. 1999. *Internet X. 509 Public Key Infrastructure Certificate and CRL Profile*. Technical Report. RFC 2459, January.

[16] Qinwen Hu, Muhammad Rizwan Asghar, and Nevil Brownlee. [n.d.]. Certificate Revocation Guard (CRG): An Efficient Mechanism for Checking Certificate Revocation. In *2016 IEEE 41st Conference on Local Computer Networks (LCN)*.

[17] D. Kumar, M. Bailey, Z. Wang, M. Hyder, J. Dickinson, G. Beck, D. Adrian, J. Mason, Z. Durumeric, and J. A. Halderman. 2018. Tracking certificate misissuance in the wild. In *In Proceedings of the IEEE Symposium on Security and Privacy (SP)*.

[18] Adam Langley. 2012. Revocation Checking and Chrome's CRL. *ImperialViolet (blog)* (2012).

[19] James Larisch, David Choffnes, Dave Levin, Bruce M Maggs, Alan Mislove, and Christo Wilson. 2017. CRLite: A Scalable System for Pushing All TLS Revocations to All Browsers. In *In Proceedings of the IEEE Symposium on Security and Privacy (SP)*. IEEE, 539–556.

[20] Xin Li, Minmei Wang, Huazhe Wang, Ye Yu, and Chen Qian. 2019. Toward Secure and Efficient Communication for the Internet of Things. *IEEE/ACM Transactions on Networking* (2019).

[21] Yabing Liu, Will Tome, Liang Zhang, David Choffnes, Dave Levin, Bruce Maggs, Alan Mislove, Aaron Schulman, and Christo Wilson. 2015. An End-to-end Measurement of Certificate Revocation in the Web's PKI. In *In Proceedings of the Internet Measurement Conference (IMC)*. ACM, 183–196.

[22] Michael Myers, Rich Ankney, Ambarish Malpani, Slava Galperin, and Carlisle Adams. 1999. *X. 509 Internet Public Key Infrastructure Online Certificate Status Protocol-OCSP*. Technical Report. RFC 2560.

[23] Alma Oracevic, Selma Dilek, and Suat Ozdemir. 2017. Security in Internet of Things: A Survey. In *In Proceedings of the International Symposium on Networks, Computers and Communications (ISNCC)*. IEEE, 1–6.

[24] Rasmus Pagh and Flemming Friche Rodler. 2001. Cuckoo Hashing. In *In Proceedings of the European Symposium on Algorithms (ESA)*. Springer, 121–133.

[25] Colin Percival. 2003. Binary Diff/Patch Utility. *URL: http://www. daemonology. net/bsdiff* (2003).

[26] Sanaz Rahimi Moosavi, Tuan Nguyen Gia, Amir-Mohammad Rahmani, Ethiopia Nigussie, Seppo Virtanen, Jouni Isoaho, and Hannu Tenhunen. 2015. SEA: A Secure And Efficient Authentication And Authorization Architecture for IoT-based Healthcare Using Smart Gateways. In *Procedia Computer Science*, Vol. 52. Elsevier, 452–459.

[27] Eric Rescorla and Nagendra Modadugu. 2012. Datagram Transport Layer Security Version 1.2. (2012).

[28] Shouqian Shi and Chen Qian. 2020. Ludo Hashing: Compact, Fast, and Dynamic Key-value Lookups for Practical Network Systems. In *Abstracts of the 2020 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems*.

[29] Trevor Smith, Luke Dickinson, and Kent Seamons. 2020. Let's Revoke: Scalable Global Certificate Revocation. In *In Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[30] John Solis and Gene Tsudik. 2006. Simple and Flexible Revocation Checking with Privacy. In *International Workshop on Privacy Enhancing Technologies*. Springer, 351–367.

[31] Pawel Szalachowski, Laurent Chuat, Taeho Lee, and Adrian Perrig. 2016. RITM: Revocation in the Middle. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*.

[32] Samuel Tanner Lindemer. 2019. *Digital Certificate Revocation for the Internet of Things*. Master's thesis. KTH Royal Institute of Technology.

[33] Yang Tong, Dongsheng Yang, Jie Jiang, Siang Gao, Bin Cui, Lei Shi, and Xiaoming Li. 2019. Coloring Embedder: a Memory Efficient Data Structure for Answering Multi-set Query. In *In Proceedings of the IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1142–1153.

[34] Daryl Walleck, Yingjiu Li, and Shouhuai Xu. 2008. Empirical Analysis of Certificate Revocation Lists. In *In Proceedings of the IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 159–174.

[35] Minmei Wang, Chen Qian, Xin Li, and Shouqian Shi. 2019. Collaborative Validation of Public-key Certificates for IoT by Distributed Caching. In *In Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 847–855.

[36] Minmei Wang, Mingxun Zhou, Shouqian Shi, and Qian Chen. 2020. Vacuum Filters: More Space-Efficient and Faster Replacement for Bloom and Cuckoo Filters. In *In Proceedings of International Conference on Very Large Databases (PVLDB)*.

[37] Ye Yu, Djamal Belazzougui, Chen Qian, and Qin Zhang. 2018. Memory-Efficient and Ultra-fast Network Lookup and Forwarding using Othello Hashing. *IEEE/ACM Transactions on Networking* 26, 3 (2018), 1151–1164.

[38] Liang Zhang, David Choffnes, Dave Levin, Tudor Dumitraş, Alan Mislove, Aaron Schulman, and Christo Wilson. 2014. Analysis of SSL Certificate Reissues and Revocations in The Wake of Heartbleed. In *In Proceedings of the Conference on Internet Measurement Conference (IMC)*. 489–502.

[39] Peifang Zheng. 2003. Tradeoffs in Certificate Revocation Schemes. *ACM SIGCOMM Computer Communication Review* (2003).

[40] Dong Zhou, Bin Fan, Hyeontaek Lim, David G Andersen, Michael Kaminsky, Michael Mitzenmacher, Ren Wang, and Ajaypal Singh. 2015. Scaling Up Clustered Network Appliances with ScaleBricks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*.

A APPENDICES

A.1 Preliminaries

A.1.1 Cuckoo Filters. Cuckoo Filter is inspired by Cuckoo Hashing Table [24], in which a key can be stored in two candidate buckets of a hash table, whose positions are calculated with two hash functions. We take a (2, 4)-Cuckoo Filter as an example to illustrate the algorithm. As shown in Fig 15, the Cuckoo Filter maintains a cuckoo hashing table with two hash functions $h_1(x)$ and $h_2(x)$. Each bucket of the table has four slots.

Insert(k): To insert a key k into the Cuckoo Filter, the operation can be accomplished by inserting the fingerprint of k, i.e., $fp(k)$, into either one of the two candidate buckets of cuckoo hashing table. Specifically, the two candidate positions, i.e., $h_1(k)$ and $h_2(k)$, can be calculated using a single uniform hash function $h(x)$ by:

$$\begin{aligned} h_1(k) &= h(k) \bmod m, \\ h_2(k) &= h_1(k) \oplus (h(fp(k)) \bmod m), \end{aligned} \tag{3}$$

where $\oplus$ is the bit-wise *xor* operation, m is the size of buckets. Since it is easy to show $h_1(k) = h_2(k) \oplus (h(fp(k)) \bmod m)$, the cuckoo filter can find the alternate bucket position of k by simply calculating the *xor* of one bucket position and the hash of the fingerprint, i.e.,

$$h_j(k) = h_i(k) \oplus (h(fp(k)) \bmod m), \{i, j\} = \{1, 2\}. \tag{4}$$

If either of the two candidate buckets contains an empty entry, then the fingerprint $fp(k)$ is safely inserted to the empty entry. Otherwise, the insertion algorithm chooses a random entry of the two buckets and reallocate the stored fingerprint FP' into its alternate buckets in the hashing table, then insert $fp(k)$ to that entry. When reallocating FP', if the alternate bucket of FP' is also full, the algorithm will repeat randomly kick off another fingerprint from the table and reallocate the other fingerprint until an empty entry is found, or until the maximal number kicking-off operations is reached, which implies the filter is too full to insert the new key k and the filter should be rebuilt with extra buckets.

Query(k): To lookup whether a key k is a member, we only need to visit the two candidate buckets of the cuckoo filter using Eq. 3. If either of the buckets contains $fp(k)$, then we conclude k is in the set; otherwise it is not.

Delete(k): Similarly, the deletion of a key k from the membership set can be accomplished by simply removing one copy of $fp(k)$ from the found bucket entry.

One limitation of Cuckoo Filter is that the number of buckets m in the Cuckoo Hashing table has to be an exact power of two [36] to support the *xor* operation. Hence, cuckoo filter can hardly achieve its theoretical optimal memory consumption in real applications where the size of keys is arbitrary, as many of the buckets might be wasted. To solve this limitation, a variant of Cuckoo Filter, named Vacuum Filter [36], is proposed. In Vacuum Filter, the alternate bucket position $h_j(k)$ is calculated by:

$$h_j(k) = h_i(k) \oplus (h(fp(k)) \bmod L), \{i, j\} = \{1, 2\}, \tag{5}$$

where L is the size of a mini-chunk (a group of consecutive buckets) within the whole hashing table and L equals to a power of two. The value L is selected from four optional values based on the last two bits of $fp(k)$, where each optional value represents the minimal size of the mini-chunk that can successfully allocate 1/4, 1/2, 3/4

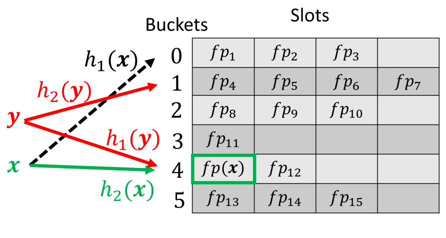


Figure 15: Cuckoo Filter

and 1 fraction of the whole inserted keys with the expected load factor, e.g., 0.95. In this way, the alternate position searching can be performed within the local mini-chunk of size L instead of the entire table of size m. Hence, m can be any positive integer and the Vacuum Filter can always achieve the optimal memory for any key size given the expected false positive rate ε.

A.1.2 Othello. Othello [37] is an efficient zero-error data structure to solve the Multiset Query Problem based on minimal perfect hashing. Othello maintains two hashing tables, with each bucket of the hashing tables containing L bits, where $L = \lceil \log_2 n \rceil$ and n is the number of distinct sets.

Suppose the lengths of the two hashing tables T_a and T_b are m_a and m_b, and the corresponding uniform hash functions are $h_a(x)$ and $h_b(x)$.

Othello is constructed by finding an acyclic undirected graph $G = (V_a, V_b, E)$, where E is the edge set, V_a, V_b are the vertex sets with each node $v_a^i \in V_a$ $(0 \le i < m_a)$ and $v_b^j \in V_b$ $(0 \le j < m_b)$ representing the ith and jth bucket of T_a and T_b.

Initially, $E = \varnothing$. For any key-value pair (k, v) with $k \in U$ and $v \in \{0, 1\}$, v can be stored in graph G by inserting a new edge $\left(v_a^i, v_b^j\right)$ in E, where $i = h_a(k)$ and $j = h_b(k)$ (as shown by the red or the green edges in Fig 4). The query function $f: U \to V$ for the key-value mapping is defined as: ***Query***$(k) = t_a[i] \oplus t_b[j]$, where $t_a[i]$ and $t_b[j]$ represent the entry in the ith and jth bucket of T_a and T_b respectively.

If the graph G remains acyclic after inserting all keys in U, then it can be proved that there exists a solution to fill the buckets in T_a and T_b with either "1" or "0", such that for any $k \in U$ and its corresponding value $v \in \{0, 1\}$, $f(k) = v$. However, when a circle is found while building the graph G, the graph should be rebuilt by using different hash functions $h_a(x)$ and $h_b(x)$. In practice, if the all key-value pairs are known in advance, Othello finds the two valid hash functions $h_a(x)$ and $h_b(x)$ that do not create any circle in the graph first by random searching, and then uses depth-first-search (DFS) order of the resulting acyclic to insert all keys.

Construct(P, N): Let P and N are the two sets used to construct an Othello. Suppose list $(e_1, e_2, ..., e_m)$ be the edge set E sorted in its DFS order. Then for any edge e in the sorted list, we find the corresponding key k which is represented by e, i.e., the indexes i, j of the two vertices are $h_a(k)$ and $h_b(k)$ respectively. Let v be the mapping value of k, namely, $v = 1$ if $k \in P$ (as shown by the green edge in Fig 4), and $v = 0$ if $k \in N$ (as shown by the red edge in Fig 4). Then v can be inserted to the table by the following steps. If both $t_a[i]$ and $t_b[j]$ are empty, we set $t_a[i] = 0$ and $t_b[j] = v$. Otherwise, one bucket of $t_a[i]$ and $t_b[j]$ must be empty since G is acylic and e is visited according to the DFS order of E. In this case, we set the empty bucket to be the "xor" result of the value in the other bucket and v.

It can be proved that if $m_a \geq 1.33n$ and $m_b \geq n$, where n is the number of all keys, then memory is sufficient enough to find the appropriate hash functions, which avoid cycles for the entire key set, with a small researching probability in Othello's construction function ***Construct***() [37]. In addition, with this memory settings, Othello can also support value flipping (change the value of a key k from "0" to "1" or from "1" to "0") ***Flip***(k), deletion ***Delete***(k) and insertion ***Insert***(k, v) functions using $O(1)$ time.

Although Othello is memory and query efficient to store arbitrary key-value mapping: it costs 2.33 bits per key for binary value mapping and only two hashing operations for each query; it is far more from being optimal to solve the CR verification problem, where the sizes of the revoked certificate set and the unrevoked certificate set are highly imbalanced. For example, in Table 1, we show Othello requires moderately smaller memory than the naive CRL approach when only 1% certificate are revoked. In the following sections, we will show a more concise data structure for the CR verification problem by optimizing Othello with a probabilistic filter.

A.2 Updating Functions of Cuckoo Filter and Othello

Every updating in tracker-plane DASS is a combination of updating operations in its maintained Vacuum (Cuckoo) Filter and Othello hashing table. We illustrate and discuss how Vacuum Filter and Othello could be updated in this section.

A.2.1 Updating of vacuum (cuckoo) filter. Cuckoo Filters are known to outperforms Bloom Filters mainly in that they can efficiently support the deletion of keys from the filter. The updating functions (***Delete***(k) and ***Insert***(k)) of Cuckoo Filters are summarized in Section A.1.1. Algorithm and implementation details can be found from Fan, et al.'s [12] and Wang, et al.'s [36] work.

A.2.2 Updating of othello hash table. In this section, we present the inserting, value flipping and deleting methods of othello hashing.

Insert(k, v): Let $G = (V_a, V_b, E)$ be the maintained graph in Othello and t_a, t_b are the hash table arrays. Inserting a key-value pair (k, v) into Othello is equivalent to adding an edge e in G, where $e = (V_a(h_a(k)), V_b(h_b(k)))$, and h_a and h_b are the selected hash functions that map the key k to the graph vertices. If the resulting graph $G = (V_a, V_b, E + \{e\})$ creates a cycle, showing the table is too full to insert the key, then the Othello hash table should be rebuilt by selecting a new pair of hash functions h_a and h_b. Otherwise, the insertion is successful and we need to assign a color (as shown in Fig. 4) for this edge. If the color flag ("0" or "1") of edge e exactly equals $t_a[h_a(k)] \oplus t_b[h_b(k)]$, then nothing needs to be changed. Otherwise, we need the modify the color flag of edge e by tweaking the values of vertices stored in t_a and t_b, namely conduct the value flipping (flip the stored value of a key k) operation.

Flip(k): Let T be the tree that contains the edge e whose color should be modified. Assume T is separated into two sub-trees T_1 and T_2 by e. One method to change the value flag of e is to flip all values stored in the vertices of T_1 or T_2 (whichever is smaller). Yu, et at.'s study [37] shows by setting the total size of Othello hash table as $2.33m$, where m is the number of keys, the value flipping operation costs $O(1)$ complexity, i.e., $O(1)$ number of table entries should be flipped for each insertion and value flipping operation.

Delete(k): Deletion of a stored key k from the othello table can be accomplished by removing the corresponding edge $e = (V_a(h_a(k)), V_b(h_b(k)))$ from G. After deletion, the actual hashing tables T_a and T_b are not changed. Thus, the ***Delete***(k) function is only a logical deletion process: it will not change the inference behavior of othello; it only remove redundant edges to provide space for new keys in the future.

Method	Hash (P)	Hash (N)	LD (P)	LD (N)
TinyCR	4	2-4	4	2-4
Othello	2	2	2	2
CRLite	≥ 2	≥ 1	≥ 2	≥ 1

Table 3: Number of hash or memory read operations for querying the classifier.

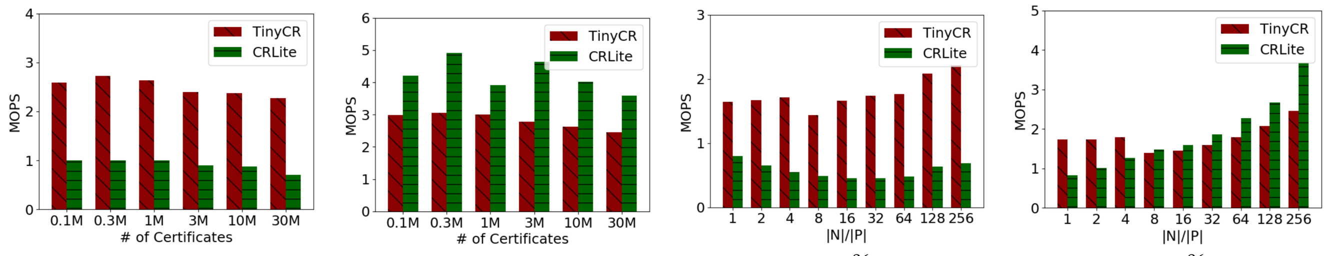
A.3 Query Performance of DASS

In this section, we analyze the query performance of DASS as a conventional set query data structure. Table 3 shows the number of hashes and memory lookup operations required by DASS, Othello and CRLite for a query. In the table, we notice Othello always requires only two hash and memory lookup operations. Thus, Othello is most efficient for query at the cost of higher memory consumption. Compared with Othello, DASS requires totally 3-4 times of hash and memory lookup for querying a revoked certificate and 1-4 times of hash and memory lookup for querying an legitimate certificate. However, for CRLite, the upper bounds of hash and memory operations depends on the depth of the filter cascades.

In addition, we test the query throughput (measured by millions of operations per second, MOPS) using the CenSys dataset on the Raspberry Pi 3 testbed and present the result in Fig. 16. From Fig. 16, the query throughput for TinyCR can be as high as a few millions per second for both revoked and legitimate certificate lookups. In addition, TinyCR is more efficient to detect a revoked certificate than CRLite when using similar memory cost, while CRLite is more efficient for checking a legitimate certificate, as most legitimate certificates can be verified using only the first filter layer.

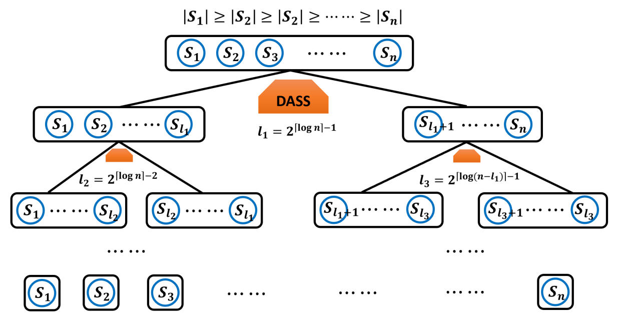
A.4 DASS for Multi-Set Query Problem

A.4.1 Design. In a global IoT or mobile network, devices could be separated into disjoint sub-groups based on their identities or certificates, and devices among different groups could have different trust levels or privileges. For example, in a smart-city IoT network, devices deployed and maintained by the government usually have the highest level of trust by other device clients. For another example, devices with a higher VIP level usually can access more resources or privileges than other clients. In such scenarios, devices are classified by their identities and other devices or third-party should be able to query the group that the device belongs to. Similar as the CR verification problem, those problems can also be solved by just querying the group of the device's certificate while validating the certificate; whereas when the number of groups is larger than 2, the problem becomes a multi-set query problem.



(a) $r = 100$, P (b) $r = 100$, N (c) $n = 2^{26}$, P (d) $n = 2^{26}$, N

Figure 16: Query throughput on Raspberry Pi 3. (a) Query revoked certificates with constant $r = 100$. (b) Query legitimate certificates with constant $r = 100$. (c) Query revoked certificates from a key set of size $n = 2^{26}$. (d) Query legitimate certificates from a key set of size $n = 2^{26}$.



$|S_1| \geq |S_2| \geq |S_2| \geq \cdots\cdots \geq |S_n|$
DASS
$l_1 = 2^{\lceil \log n \rceil - 1}$
$l_2 = 2^{\lceil \log n \rceil - 2}$
$l_3 = 2^{\lceil \log(n-l_1) \rceil - 1}$

Figure 17: Decision tree of DASS for multi-set query.

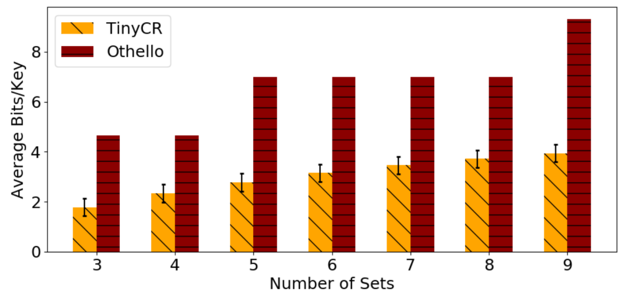


Figure 18: Mean & standard deviation of amortized memory cost for multi-set query.

DASS could also be extended to the multi-set query problem. Specifically, we can construct a binary decision tree, called DT-DASS (as shown in Fig 17), and use a DASS at each tree node to split the keys into two sub-groups. The decision tree stops growing when the leaf node is a pure node, namely, all keys represented by the leaf node are from one specific set S_k. For n groups, DT-DASS can separated the groups with a decision tree of height $\lceil \log_2 n \rceil$.

This decision tree based data structure can be optimized by utilizing the good property of DASS when handling imbalanced classes. A straightforward method is the greedy strategy (as shown in Fig 17): we first sort the groups by their sizes, then greedily choose the split point at each node such that the ratio of the left and right child node sizes is maximized, while guaranteeing we do not introduce an additional tree layer to fully separate all groups.

To show how this algorithm benefits from the imbalanced set distribution, we can compare DT-DASS with Othello, which classify the keys by encoding the group IDs as the stored values for the keys. In Othello, we can consider the encoded value representing the group ID for each key requires at least $\lceil \log_2 n \rceil$ bits. Thus, the $\lceil \log_2 n \rceil$ long bits can also forms a binary decision tree of height $\lceil \log_2 n \rceil$, where nodes in each layer together can be considered as an one-bit Othello. However, in such Othello search tree, the memory cost of the nodes in every layer (i.e., the one-bit Othello size) is constant and independent with the distribution of the keys. Therefore, the total memory cost of Othello is $\lceil 2.33|S| \log_2 n \rceil$, where $|S|$ is the total key size. In contrast, the total number of groups in each layer of DT-DASS is equal or smaller than n, because some nodes could become pure leaf nodes in the intermediate layers and would not be considered again in layers below, when n is not an exact power of 2. Meanwhile, at each split node, the DASS memory size is only determined by the minor sub-group and the ratio between the two sub-groups. Therefore, based on the prior analysis, the total memory cost at each layer is smaller than Othello's layer cost if the sub-groups at the nodes are imbalanced.

In many real applications, the sizes of groups (such as different VIP levels of devices) are significantly different. For example, a common distribution of the group sizes is the "pyramidal shape", i.e., the group sizes scale inversely with the levels of the hierarchy. Meanwhile, n could be any arbitrary integer instead of a pow of 2. Therefore, DT-DASS costs much less memory than Othello for multi-set query when handling imbalanced groups. As a trade-off, the query cost of DT-DASS grows logarithmically with the number of sets n (i.e., $\Theta(\log n)$) in worst case.

A.4.2 Evaluation. We compare the memory performance of DASS with Othello, which can also be used to solve Multiset Query problem.

For Othello, we still use the recommended memory setting (2.33 bits per slot) by the original paper [37]. Meanwhile, for L classes, each key requires $\lceil \log_2 L \rceil$ slots to store the value (class label). Hence, in total, Othello requires constantly $2.33 \lceil \log_2 L \rceil$ bits for each key.

In the experiments, we determine the size of each set by uniformly selecting a random number between 1 to 10,000,000. We conducted 7 groups of experiments using 3 to 9 sets respectively. In each group the experiments are repeated 1000 times. In Fig. 18, we show the average amortized memory cost of each key when using TinyCR or Othello for 3 to 9 sets respectively. We find TinyCR costs 46% to 61% less memory than Othello for multi-set queries.