A note on securing insertion-only Cuckoo filters

Fernando Virdia¹ Mia Filič²

¹ NOVA LINCS & Universidade NOVA de Lisboa, Lisbon, Portugal
² Applied Cryptography Group, ETH Zürich, ETH Zürich, Switzerland

Abstract. We describe a small tweak to Cuckoo filters that allows securing them under insertions using the techniques from Filič et al. (ACM CCS 2022), without the need for an outer PRF call.

In [FPUV22], Filič, Paterson, Unnikrishnan and Virdia define simulation-based security notions that capture some privacy and correctness guarantees for Probabilistic Data Structures handling Approximate Membership Queries (AMQ-PDS). They proceed to prove that Bloom filters [Blo70] trivially achieve these guarantees when the hash functions used by the filter are replaced with pseudorandom functions (PRF). They then attempt to prove the same for “insertion-only” Cuckoo filters [FAKM14] (i.e., Cuckoo filters where only the insertion functionality is made available to the adversary) but notice that simply replacing hash functions with PRFs is not sufficient in this case. They address this issue by pre-processing inputs to the Cuckoo filter by first passing them through a PRF.

In formal terms, the issue with Cuckoo filters (solved by adding an outer PRF call) is their lack of function-decomposability (c.f. Definition 3.1 and Section 4.2 of [FPUV22]). Here, we propose a different tweak to Cuckoo filters that provides them with function-decomposability without adding an outer PRF call on top of the two original hash-function calls. In the following, we use the same notation used by Filič et al. First, we recall the description of Cuckoo filters given by [FPUV22].

Definition 1 (Cuckoo filter). Let pp = (s, λ₁, λ₄, num) be a tuple of positive integers. We define an (s, λ₁, λ₄, num)-Cuckoo filter to be the AMQ-PDS with algorithms defined in Figure 1, making use of hash functions Hₚ : D → {0,1}^{λ₄} and Hₜ : D → {0,1}^{λ₁}.

In order to achieve function-decomposability without pre-processing inputs, we increase the output length of Hₚ by λ₁ bits, such that Hₚ : D → {0,1}^{λ₄+λ₁}. We then read from the output of Hₚ (rather than from the output of Hₜ) the index i₁ of the first candidate bucket used to “store” x ∈ D, and use Hₜ only to derive the index i₂ of the second candidate bucket. This would mean defining qₚₜ and upₚₜ as shown in Figure 2.

Definition 2 (Function-decomposable Cuckoo filter). Let pp = (s, λ₁, λ₄, num) be a tuple of positive integers. We define an (s, λ₁, λ₄, num)-function-decomposable Cuckoo filter to be the AMQ-PDS with algorithms obtained by applying the changes in Figure 2 to Figure 1, making use of hash functions Hₚ : D → {0,1}^{λ₄+λ₁} and Hₜ : D → {0,1}^{λ₁}.

Lemma 1. Function-decomposable Cuckoo filters from Definition 2 with oracle access to a random function F are F-decomposable [FPUV22, Def. 3.1], reinsertion invariant [FPUV22, Def. 3.2], and satisfy consistency rules of insertion-only AMQ-PDS [FPUV22, Def. 3.10].

Proof. Let F ← Func[D,Ù] where Ù = {0,1}^{λ₄+λ₁} ⊂ D. Replace Hₚ with F in Definition 2. Then, F-decomposability follows from observing that

\[ up^{F,H₁}(x,σ) = up^{Idₚ,H₁}(F(x),σ) \forall x \in D, σ ∈ Σ, \]
\[ qₚₜ^{F,H₁}(x,σ) = qₚₜ^{Idₚ,H₁}(F(x),σ) \forall x \in D, σ ∈ Σ, \]

where up^{Idₚ,H₁} and qₚₜ^{Idₚ,H₁} lack oracle access to F, which is truly random. Reinsertion invariance and the other consistency properties follow from inspection of up^{F,H₁} and qₚₜ^{F,H₁}, in the same way they hold for Cuckoo filters.
setup\(pp\)

1 \(s, \lambda I, \lambda T, \text{num} \leftarrow pp\)
2 // Initialise \(2^\lambda I\) buckets, \(s \lambda T\)-bit slots
3 \textbf{for} \(i \in 2^\lambda I:\) \(\sigma_i \leftarrow \perp\)
4 \textbf{return} \(\sigma \leftarrow (\sigma_i)_i, \sigma_{evic}\)

\text{qry}^{H_T, H_I}(x, \sigma)

1 \(\text{tag} \leftarrow H_T(x)\)
2 \(i_1 \leftarrow H_I(x)\)
3 \(i_2 \leftarrow i_1 \oplus H_I(\text{tag})\)
4 \(a \leftarrow [\text{tag} \in \sigma_{i_1} \text{ or } \text{tag} \in \sigma_{i_2} \text{ or } \text{tag} = \sigma_{evic}]\)
5 \textbf{return} \(a\)

\text{up}^{H_T, H_I}(x, \sigma)

1 \(\text{tag} \leftarrow H_T(x)\)
2 \(i_1 \leftarrow H_I(x)\)
3 \(i_2 \leftarrow i_1 \oplus H_I(\text{tag})\)
4 // check if up was disabled, first
5 \textbf{if} \(\sigma_{evic} \neq \perp : \textbf{return } \perp, \sigma\)
7 \textbf{if} \(\text{tag} \in \sigma_{i_1} \text{ or } \text{tag} \in \sigma_{i_2} : \textbf{return } \top, \sigma\)
9 \textbf{for} \(i \in \{i_1, i_2\} \text{ // in that order}\)
10 \textbf{if} \(\text{load}(\sigma_i) < s\)
11 \(\sigma_i \leftarrow \sigma_i \odot \text{tag}\)
12 \textbf{return} \(\top, \sigma\)
13 // if no empty slots, displace something
14 \(i \leftarrow \{i_1, i_2\}\)
15 \textbf{for} \(g \in [\text{num}]\)
16 \(\text{slot} \leftarrow [s]\)
17 \(\text{elem} \leftarrow \sigma_{i, \text{slot}} \text{ // element to be evicted}\)
18 // swap elem and tag
19 \(\sigma_{i, \text{slot}} \leftarrow \text{tag}; \text{tag} \leftarrow \text{elem}\)
20 \(i \leftarrow i \oplus H_I(\text{tag})\)
21 \textbf{if} \(\text{load}(\sigma_i) < s\)
22 \(\sigma_i \leftarrow \sigma_i \odot \text{tag}\)
23 \textbf{return} \(\top, \sigma\)
24 // could not store \(x\) without an eviction
25 \(\sigma_{evic} \leftarrow \text{tag} \text{ // last value of tag after loop}\)
26 \textbf{return} \(\top, \sigma\)

Fig. 1: AMQ-PDS syntax instantiation for the Cuckoo filter.

<table>
<thead>
<tr>
<th>\text{qry}^{H_T, H_I}(x)</th>
<th>\text{up}^{H_T, H_I}(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (\text{tag} \mid i_1 \leftarrow H_T(x))</td>
<td>1 (\text{tag} \mid i_1 \leftarrow H_T(x))</td>
</tr>
<tr>
<td>2 (i_2 \leftarrow i_1 \oplus H_I(\text{tag}))</td>
<td>2 (i_2 \leftarrow i_1 \oplus H_I(\text{tag}))</td>
</tr>
<tr>
<td>3 (\ldots)</td>
<td>3 (\ldots)</td>
</tr>
</tbody>
</table>

Fig. 2: Changes to apply to Figure 1 to obtain a function-decomposable Cuckoo filter variant. We note that the call to \(H_I\) on Line 20 of \(\text{up}^{H_T, H_I}\) in Figure 1 does not change.
Bibliography

