BitAFL: Provide More Accurate Coverage Information for Coverage-guided Fuzzing

Hang Xu[a], Zhi Yang*, Xingyuan Chen, Bing Han, Xuehui Du

PLA Information Engineering University, Zhengzhou, China

[a]e-mail: xuhangzzz@outlook.com
*Corresponding author's e-mail: zynoah@163.com

Abstract. CGF (Coverage-guided fuzzing) has found a large number of software vulnerabilities with its low cost and adaptability. CGF mutates at the bit or byte level, so most of the mutated test cases cover the same paths. But no previous work had quantified the percentage of test cases that covered the duplicate paths. Therefore, we designed the experimental framework GSPR (get same path rate) based on AFL. We fuzzed seven applications using GSPR and found that approximately 70% of the test cases covered duplicate paths. Based on the above experimental results, we solve the hash collision issue in AFL. We analyzed the various situations that cause hash collision, and introduced the concepts of local collision and global collision. Because a large number of test cases cover duplicate paths, there are much repeated global collision. Based on these findings, we propose different solutions to hash collision according to the size of target program. We extended AFL to implement BitAFL and evaluated it on seven applications. In a comparison experiment with AFL, the results show that our method can completely eliminate hash collisions in small programs. In large programs, BitAFL is able to reduce collisions by more than 80%. In addition, on average, BitAFL found 8.87% more paths than AFL. In summary, our approach provides AFL with more accurate coverage information and can find more paths.

Keywords: fuzzing, vulnerability, hash collision, bit operation, instrumentation

1 Introduction

CGF (Coverage-guided fuzzing) is the most popular fuzzing technology at present, which plays an important role in protecting software security. AFL (American Fuzzy Lop) [1] is the most successful representative work in the CGF field, and academia and industry have formed an ecosystem based on AFL. Existing studies AFLFast [2] and EcoFuzz [3] show that when fuzzing target program with AFL, most test cases do not cover new path. UnTracer [4] also notes that very few test cases cover new path, so it discards most test cases that do not increase coverage. However, no work has been done on an experimental basis to quantify the percentage of test cases that execute duplicate paths (For the sake of presentation, we'll call it repetition rate for short). Therefore, we

© The Author(s) 2024
A. Rauf et al. (eds.), *Proceedings of the 3rd International Conference on Management Science and Software Engineering (ICMSSE 2023)*, Atlantis Highlights in Engineering 20,
https://doi.org/10.2991/978-94-6463-262-0_54

designed and implemented an experimental framework GSPR (get same path rate) for counting repetition rate. Importantly, our experimental framework can be integrated into the AFL extension based coverage-guided fuzzers. We integrated the experimental framework into AFL and AFLFast, and fuzzed 7 real applications. According to the statistics, the average repetition rate is as high as 70%, which also indicates that most hash collisions are repeated.

Coverage information guides the exploration direction and its accuracy has become the key to measure the success of a fuzzer. AFL uses edge coverage, which provides more accurate coverage information than the basic block coverage adopted by Honggfuzz [5] and LibFuzzer [6]. AFL uses a 64KB shared bitmap to record the edge coverage information, and each byte in the bitmap records the hit count for one edge. The ID of one edge is calculated by a specific hash function, and also as the offset in the shared bitmap. However, there is hash collision in the calculation of the offset. Different edges may have the same offset in the shared bitmap, so they share a same byte to store the hit count. Hash collision leads to inaccurate edge count and hides new paths. Further, inaccurate coverage information may affect the exploration direction of fuzzers.

Aiming at the hash collision issue in AFL, we analyzed various situations that cause hash collision, and analyzed the probability of various collisions quantitatively. On the basis of the above repetition rate experiment and collision analysis, we divided the programs into large programs and small programs according to the number of edges. For different programs, we use different methods to resolve hash collision. We extended AFL to implement BitAFL and evaluated BitAFL on 7 applications. The results show that our method can completely eliminate hash collisions in small programs. In large programs, the number of collision bytes decreased by an average of 83.05%, and the number of collisions per test case decreased by an average of 97.31%. In addition, on average, BitAFL found 8.87% more paths than AFL, and covered 6.33% more edges than AFL.

2 Hash Collision in AFL

2.1 Hash Algorithm in AFL

Algorithm 1: Hash algorithm in AFL

1. *MAP_SIZE* = 2^H
2. *BBID* = compile_time_random(*MAP_SIZE*)
3. *EdgeID* = (*srcBBID* >> 1) ^ *desBBID*
4. *Bitmap*[*EdgeID*]++

Next, we will introduce how AFL records edge coverage information. In Algorithm 1, *Bitmap* represents the shared bitmap, which stored in bytes. *MAP_SIZE* indicates the size of shared bitmap. Each byte in the shared bitmap stores the hit count for an edge. For each basic block of the target program, AFL randomly assigns it an ID at compile time. For one edge, *srcBBID* is the ID of the source basic block, *desBBID* is the ID of the destination basic block, and [*srcBBID*, *desBBID*] is a tuple. The hash function is

applied to the tuple to calculate the edge ID, *EdgeID*, which is also used as the offset in the shared bitmap. The hit count of *EdgeID* is stored in the corresponding offset byte in the bitmap.

Because of the randomness of the basic block ID, different edges may get the same edge ID (that is, the same offset in bitmap) after calculation by hash function. Then they share a same byte to store the hit count, so the fuzzer cannot distinguish between these two edges, which is called a hash collision.

In the execution of AFL, hash collision will occur within a single test case, which we call **local collision**, and also between test cases, which we call **global collision**. Local collision causes the coverage of test case to be lower than the real value, which leads to the lower score of the test case, and ultimately affects the decision in the mutation stage. The impact of global collision is more serious. For example, if a test case covers a new edge, but conflicts with the previously covered edge, the fuzzer will not regard it as a new coverage. Then the opportunity to further explore the new path will be missed.

2.2 Three Types of Hash Collision

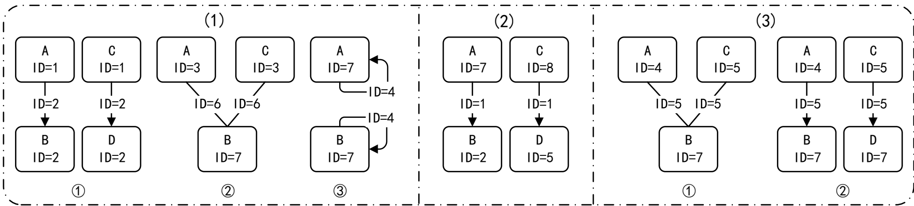

Fig. 1. Three types of hash collision

According to the hash algorithm in AFL, we analyzed the collision types and took the similarities and differences between *srcBBID* and *desBBID* as the classification basis. Since the basic block ID is randomly allocated, hash collisions can be divided into the three types, as shown in Fig. 1. Where H is the size of the hash space, that is the bitmap size, and n is the number of existing edges.

Type 1. The *srcBBID*s and *desBBID*s of both edges are same, so their hash values are same.

1) One edge from A to B, and another edge from C to D, the basic block ID of A and C are same, the basic block ID of B and D are same. The collision probability is n/H^2.

2) In particular, when two basic blocks A and C with the same ID are passed into the same basic block, hash collision will occur. The collision probability is $1/H$.

3) Basic blocks A and B both have self-cyclic edges, and the IDs of A and B are same. The collision probability is $1/H$.

Type 2. The *srcBBID*s and *desBBID*s are different.

Because of the particularity of exclusive or operation, two different values may get the same value. As shown in Fig. 1, although the IDs of basic blocks A, B, C, and D are all different, the calculated hash values are the same. The collision probability is n/H.

Type 3. While *srcBBID* is different, *desBBID*s are same.

1) We notice that there is a shift operation in the hash operation (*srcBBID* >> 1), where two numbers with different lowest bit move to the right one bit will get the same value. As shown in Fig. 1, the IDs of A and C differ only in the lowest bit, and when they are passed into the same base block B, the hash value are same. The collision probability is $1/H$.

2) When they pass to different basic blocks B and D, since the basic block ID of B and D are same, the ID of two edges are same. The collision probability is n/H^2.

Through static analysis, we can get that the collision situation in type 2 will occupy the vast majority. For example, with the default bitmap size (64KB) in AFL, when there are 100 edges, the probability of the next collision edge being type2 is 97%. When there are 1000 edges, the probability that the next collision edge is type2 is 99.7%. The more edges, the greater the probability of collision due to the type2. There are almost no target programs with fewer than 1000 edges in the fuzzing. Therefore, if we can solve the type2, we can avoid most collisions.

3 Method and Implementation

3.1 GSPR and Repetition Rate

As mentioned earlier, the mutation-based fuzzers use a bit or byte level mutation strategy. When the seed size is 100KB, 819200 bit flips and 102400 byte flips are required in AFL. For the target program, when the mutation occurs in the data area, the mutated test case is no different from the seed, so the covered path is no different. Furthermore, most hash collisions are repetitive.

To date, no work has systematically measured the percentage of test cases that execute duplicate paths. Stefan Nagy et al. [4] have only experimentally measured the percentage of test cases that increase coverage, considering only coverage, not execution path. Our target is execution path, that is, to consider the order in which basic blocks are executed. We count the relative order in which basic blocks are executed, that is, the order in which each basic block is first executed.

The basis for determining the execution of duplicate paths is as follows: (1) The test case generated by mutation is little different from its parent seed. Therefore, we compare the execution path of the current test case to the execution path of its parent seed. (2) When the mutation occurred in the metadata area and new path was covered, deterministic mutation would cause next mutation to occur in the adjacent location of the metadata area, so we compare the execution path of the current test case with the execution path of last test case.

We implemented an experimental framework, GSPR, which can be used in conjunction with the popular coverage-guided fuzzers. Algorithm 2 shows the workflow of integrating the framework into the AFL. We first use LLVM to instrument the target program at compile time, statically insert the code of recording relative execution sequence of the basic blocks to get the instrumented target program. Then fuzzer sets up the fuzzing environment (line 1). Save the execution path of each initial seeds (line 2). In the main cycle (lines 3-20), a seed s is selected according to the scheduling strategy (line 4). A large number of test cases were generated by mutation, and the test cases are

successively input into the target program for execution (lines 5,6). Check the execution result of test case *t* to see if it covers new path, and if so, add it to the seed queue and save the new path (lines 7-9). Otherwise, check whether the execution path is the same as the execution path of the last test case, and if so, set the flag *isSame* to true (lines 11,12). If not, update the execution path *LP* of the last test case (line 14). Determine if the execution path is the same as that of the parent seed, and if so, set the flag *isSame* to true (lines 15,16). Finally, if the flag *isSame* is true, the current test case executes a duplicate path, and the result is recorded (lines 17-19).

Algorithm 2: Algorithm of GSPR-AFL

```
Data:
    LP: path of last test case
    Prog: instrumented target binary
fuzzerSetup( )
LP = saveInitialPath( )
While True do
    s = selectOneSeedFromQueue( )
    t = mutate(s)
    runTarget(t, Prog)
    If hasNewCoverage( )
        addToQueue(t)
        LP = savePath( )
    Else
        If sameAsLast( )
            isSame = True
        Else
            LP = updateLastPath(t)
            If sameAsParent( )
                isSame = True
    If isSame is True
        record( )
        isSame = False
End While
```

We integrated GSPR into two popular coverage-guided fuzzers, AFL and AFLFast, which we chose because they are representative of coverage-guided fuzzers. They are frequently adopted by other works [4][9-12]. We evaluated 7 real open-source applications. As shown in the Table 1. The result shows an average repetition rate of 68% in the AFL, 64% with last test case and 33% with parent seed. The average repetition rate of 72% in AFLFast, 69% with last test case, and 43% with parent seed. Taken together, about 70% of test cases execute duplicate paths, which also means that more than 70% of hash collisions are repetitive.

Table 1. Repetition rate in AFL and AFLFast

Applications	AFL			AFLFast		
	rate	*last*	*parent*	*rate*	*last*	*parent*
cflow	0.51	0.42	0.29	0.58	0.51	0.42
pngfix	0.38	0.37	0.01	0.70	0.69	0.35
tiffset	0.82	0.8	0.38	0.82	0.80	0.45
pdffonts	0.79	0.77	0.25	0.79	0.76	0.47
tcpdump	0.87	0.84	0.58	0.84	0.80	0.51
readelf	0.77	0.75	0.36	0.74	0.72	0.41
xmllint	0.64	0.56	0.45	0.61	0.54	0.40
Average	**0.68**	**0.64**	**0.33**	**0.72**	**0.69**	**0.43**

3.2 Implementation of BitAFL

If we assign a unique ID to each edge, we can completely avoid hash collisions, but this is not always possible. Our idea is to assign a unique ID to each edge as far as is feasible, otherwise rehashing is used, which is a standard hash collision resolution. When the number of edges is less than the threshold *T*, we assign a unique ID to each basic block and use bit operation to ensure the uniqueness of each edge ID. Otherwise, we rehash and reassign ID to the collision edge. The specific process of our method is shown in Algorithm 3.

Algorithm 3: Hash algorithm of BitAFL

```
EdgeCount = get  edges( )
If EdgeCount < T
    MAP_SIZE = 2^2H
    For BB in BBs:
        BBID = compile_time_unique(2^H);
    EdgeID = (srcBBID <<H) ^ (desBBID); // runtime
    Bitmap[EdgeID]++
Else
    MAP_SIZE = 2^H
    For BB in BBs:
        BBID = compile_time_random(2^H)
    EdgeID = (srcBBID >> 1) ^ (desBBID | 2^(H-1)) // runtime
    If collision:
        EdgeID = srcBBID >> 1
    Bitmap[EdgeID]++
```

Bit operation. We determine the size of the bitmap according to the number of edges or basic blocks, which must be a power of 2 (line 3). Assign a unique ID for each basic block at compile time, and ensure that the ID in $[0, 2^H)$ (lines 4,5). Calculate the edge ID at run time, specifically, *srcBBID* moves *H* bit left, then xor with *desBBID*, to ensure that the edge ID is unique. Update the hit count of the corresponding byte in the bitmap (lines 6,7).

Rehashing. We determine the size of the bitmap according to the number of edges or basic blocks, which must be a power of 2 (line 9). Randomly assign an ID to each basic block at compile time (lines 10,11). Calculate the edge ID at run time, specifically, let 2^{H-1} or with desBBID, ensure that the highest bit must be 1. The operation srcBBID >> 1 ensures that the highest bit must be 0. Then the highest bit must be 1 after xor operation, which ensure that the edge ID in [2H-1, 2H) (line 12). If a collision occurs, rehash the edge ID as srcBBID >> 1, ensuring that the edge ID in [0, 2H-1) (lines 13,14). Update the hit count of the corresponding byte in the bitmap (line 15).

Next, we will show the stored process for the hit count. Fig. 2 (a) shows an example of storage when the number of edges is less than the threshold *T*. First, the source basic block ID is stored in the high *H* bit and the destination basic block ID is stored in the low *H* bit to obtain the unique edge ID, which can be realized by simple shift and xor operation. The hit count for the corresponding byte is then updated based on the edge ID.

Fig. 2 (b) shows an example of storage when the number of edges is greater than the threshold *T*. For any edge, the edge ID obtained for the first time must be within the range of $[2^{H-1}, 2^H)$. If there is no collision, the updated hit count is directly stored at the current byte. If there is a collision, the new edge ID is calculated using the pre-defined formula, and the hit count is updated at the new byte.

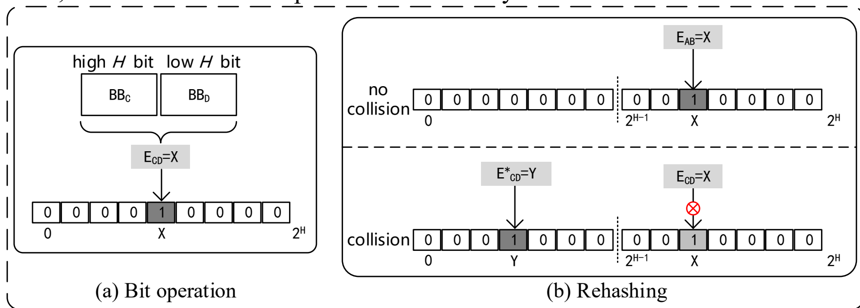

Fig. 2. Store process of hit count

It is worth mentioning that hash collision is inevitable if the number of edges exceeds the bitmap size, so we must determine the bitmap size based on the number of edges. Secondly, the rehashing method described above does not solve the situation in which collision occur at one byte more than twice. According to our early experiments, the probability of this happening is very low, less than 2% on average, which is almost negligible. Therefore, it is not worthwhile to deal with this situation in a special way. In addition, the rehashing formula we designed can ensure that the two hash values must be different, and the result of the second hash is included in the process of the first hash, so as to overcome the shortcomings of the rehashing method which increases the computation cost.

According to the ball in the box algorithm, the collision rate is about 7% when there are 10,000 edges, about 14% when there are 20,000 edges, and about 30% when there are 50,000 edges [8]. Considering the limitation of shared memory and avoiding the waste of hash space by rehashing, we finally set the threshold *T* to 2^{13}, which is 8192.

4 Evaluation

4.1 Experimental Environment

The experiment in this paper is carried out on a 64-bit virtual machine equipped with 4-core Intel i7-10710U@1.10GHz processor, 4G memory, Ubuntu20.04.1. We provide one initial seed for each program, and the seeds required for the experiment are all provided by AFL. Seven popular open-source Linux applications were selected as the test set to verify the validity and universality of our approach. As shown in Table 2.

We did a comparative trial with the AFL because the AFL is one of the most successful fuzzer in academia and industry. Although CollAFL [7] solves hash collision, it is unfortunately not open source, we only compare BitAFL with native AFL to show the effectiveness of BitAFL.

Table 2. Statistics of applications

Applications	basic block	edge	collision ratio	version	input format
cflow	6216	5505	4.08%	1.7.0	c
pngfix	5035	6625	4.89%	1.6.38	png
tiffset	8903	8504	6.22%	4.4.0	tiff
tcpdump	37914	18154	12.66%	4.9.0	pcap
pdffonts	39747	26148	17.54%	4.0.4	pdf
readelf	67306	28082	18.67%	2.39	elf
xmllint	64792	52146	31.04%	2.9.11	xml

4.2 Evaluation Criteria

Collision statistics. Effective hash collision resolution can eliminate hash collisions or greatly reduce hash collisions. Therefore, the number of collisions is the most intuitive hash collision measurement scheme. Seven target programs were fuzzed for 24 hours using AFL and BitAFL respectively. The result is shown in Table 3, where *bytes* represents the number of bytes that generated hash collision in the bitmap, *frequency* represents the average number of hash collisions per test case and *#dec* represents the rate of decrease. In small programs (cflow, pngfix), our scheme can completely eliminate hash collisions. In large programs (the rest), collision bytes are reduced by 83.05% on average, and the average number of hash collisions per test case has been reduced by 97.31%, which can effectively reduce collisions.

Table 3. Statistics of collision and coverage

Applications	AFL	BitAFL	AFL	BitAFL	AFL	BitAFL	AFL	BitAFL
	bytes	*#dec*	*frequency*	*#dec*	*paths*	*#inc*	*edges*	*#inc*
cflow	38.67	100%	18.67	100%	1572	-4.71%	2150	1.4%
pngfix	28	100%	13.14	100%	825	9.09%	1973	2.18%
tiffset	163	84.87%	9.82	93.79%	2129	6.9%	5193	5.87%

tcpdump	110.33	72.81%	1.45	99.31%	7814	15.2%	12465	4.8%
pdffonts	58	90.81%	565.24	99.96%	1062	20.43%	2123	22.61%
readelf	194.67	77.74%	2.31	94.37%	18201	8.61%	13140	3.58%
xmllint	188.33	89.02%	93.94	99.11%	2829	6.54%	6315	3.9%
Average	**111.57**	**87.89%**	**100.65**	**98.08%**	**4918**	**8.87%**	**6194**	**6.33%**

Code **coverage.** Code coverage is one of the most common criteria to evaluate the performance of a fuzzer. The more code coverage, the higher the probability of triggering vulnerability. Since the AFL only stores test case into the seed pool when it discovers new path, we use the number of seeds in the seed pool to represent the number of paths discovered by the fuzzer. In addition, we use the number of bytes occupied in the bitmap to represent the number of edges covered. We compare the code coverage of different fuzzers in terms of the number of paths found and the number of edges covered. Seven target programs were fuzzed for 100 hours using AFL and BitAFL respectively. Each target program was fuzzed three times and the data was averaged. As shown in Table 3, *paths* represents the number of the paths found, *edges* represents the number of edges covered and *#inc* represents the increased rate. The result shows that BitAFL outperforms AFL by 8.87% in terms of the number of paths found and 6.33% in terms of the number of edges covered. Overall, it shows that BitAFL can find more paths and cover more target codes.

5 Conclusion

In this paper, we studied the hash collision issue in AFL, and designed an experimental framework GSPR to count the rate of test cases executing duplicate paths. The result shows that about 70% of test cases executing duplicate paths, that is, most of the hash collisions generated in AFL are also repeated. In addition, we proposed the concepts of local collision and global collision for the first time, and systematically discussed several situations that cause hash collision. Based on experimental results and quantitative analysis, we designed a solution to hash collision, and extended AFL to achieve BitAFL. Our evaluation of BitAFL on seven open source applications showed that our method can completely eliminate hash collisions in small programs and significantly reduce hash collisions in large programs. At the same time, BitAFL can discover more paths and cover more codes. For large programs, our scheme does not completely eliminate hash collisions. In the future work, we will continue to optimize our method, and orthogonal integration of the method in this paper with more AFL optimization schemes, so as to more efficient fuzzing.

References

1. Michal Zalewski. (2019) american fuzzy lop. https://github.com/google/AFL.
2. Böhme, M. Pham, V. T. Roychoudhury A. (2016) Coverage-based Greybox Fuzzing as Markov Chain. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. Vienna Austria. pp. 1032–1043. doi:10.1145/2976749.2978428.

3. Yue, T. Wang, P. Tang, Y. et al. (2020) EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In: 29th USENIX Security Symposium (USENIX Security 20). Boston. pp. 2307-2324. https://www.usenix.org/conference/usenix-security20/presentation/yue.
4. Stefan, N. and Matthew, H. (2019) Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In: 2019 IEEE Symposium on Security and Privacy. San Francisco. pp. 787–802. doi:10.1109/sp.2019.00069.
5. Google. (2019) Honggfuzz. https://github.com/google/honggfuzz.
6. Kosta, S. (2016) Continuous Fuzzing with libFuzzer and AddressSanitizer. In: 2016 IEEE Cybersecurity Development. Boston. pp. 157–157. doi:10.1109/SecDev.2016.043.
7. Gan, S. Zhang, C. Qin, X. et al. (2018) CollAFL: Path Sensitive Fuzzing. In: 2018 IEEE Symposium on Security and Privacy. San Francisco. pp. 679–696. doi:10.1109/SP.2018.00040.
8. Michal Zalewski. (2019) Technical whitepaper for afl-fuzz. https://github.com/google/AFL/blob/master/docs/technical_details.txt.
9. Zhu, X. Feng, X. Meng, X. et al. (2020) CSI-Fuzz: Full-speed Edge Tracing Using Coverage Sensitive Instrumentation. IEEE Transactions on Dependable and Secure Computing, pp. 1–1. doi:10.1109/TDSC.2020.3008826.
10. Zhu, X. Feng, X. Jiao, T. et al. (2019) A Feature-Oriented Corpus for Understanding, Evaluating and Improving Fuzz Testing. Auckland, New Zealand. pp. 658–663. doi:10.1145/3321705.3329845.
11. She, D. Abhishek S. and Suman J. (2022) Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis. In: 2022 IEEE Symposium on Security and Privacy. San Francisco. pp. 2194–2211. doi: 10.1109/sp46214.2022.9833761.
12. Pham, V.T. Boehme, M. Santosa, A. E. et al. (2019) Smart Greybox Fuzzing. IEEE Transactions on Software Engineering. pp. 1–1. doi:10.1109/tse.2019.2941681

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial 4.0 International License (http://creativecommons.org/licenses/by-nc/4.0/), which permits any noncommercial use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.