Estimating Worst-case Resource Usage by Resource-usage-aware Fuzzing*

Liqian Chen[1] (✉), Renjie Huang[1,2], Dan Luo[1], Chenghu Ma[1,2], Dengping Wei[1] (✉), and Ji Wang[1,2]

[1] College of Computer Science, National University of Defense Technology, Changhsha, China
{lqchen,renjiehuang,luodan,machenghu,dpwei,wj}@nudt.edu.cn

[2] State Key Laboratory of High Performance Computing, Changhsha, China

Abstract. Worst-case resource usage provides a useful guidance in the design, configuration and deployment of software, especially when it runs under a context with limited amount of resources. Static resource-bound analysis can provide sound upper bounds of worst-case resource usage but may provide too conservative, even unbounded, results. In this paper, we present a resource-usage-aware fuzzing approach to estimate worst-case resource usage. The key idea is to guide the fuzzing process using resource-usage amount together with resource-usage relevant coverage. Moreover, we leverage semantic patch to make use of static analysis information (including control-flow, function-call, etc.) to instrument the original program, for the sake of aiding the subsequent fuzzing. We have conducted experiments to estimate worst-case resource usage of various resources in real-world programs, including heap memory, stack depths, sockets, user-defined resources, etc. The preliminary experimental results show the promising ability of our approach in estimating worst-case resource usage in real-world programs, compared with two state-of-the-art fuzzing tools (AFL and MemLock).

Keywords: Fuzzing · Resource Usage · Static Analysis

1 Introduction

Resources refer to any abstractions offered to a process by system calls, apart from the process itself. Typical resources in practice include heap/stack memory, sockets, file descriptors, threads, database connections, gas consumed in Solidity smart contracts, etc. In addition, there exist a variety of user-defined application-dependent resources in applications, such as buffers, memory pools, number of licenses consumed, etc. Worst-case resource usage provides a useful guidance in the design, configuration and deployment of software, especially when the software runs with limited amount of resources, e.g., under the context of modern

* This work is supported by the National Key R&D Program of China (No. 2017YFB1001802), and the NSFC (Nos. 61872445, 62032024).

cyber-physical systems, mobile systems and IoT devices, etc. Unexpected or uncontrolled resource usage may degrade program performance, or even leads to CWE (Common Weakness Enumeration) vulnerabilities (such as *uncontrolled-resource-consumption*, *file-descriptor-exhaustion*, etc.).

Static resource-bound analysis can provide sound upper bounds of worst-case resource usage but may provide too conservative, even unbounded, results. Moreover, most of existing static resource-bound analysis techniques [1,2,4,5,8, 9,14] focus on deriving the upper-bound number of accesses to a given control location or simply the bound of iterations of a loop (or recursion). The programs under analysis are often of small-scale, and complex syntactic constructs are usually being abstracted away for simplicity.

In real-world programs, resources are often manipulated via specific APIs which may involve complex structures. Moreover, the usage amount of resources often depends on not only such parameters, but also the running system environment. For example, considering *malloc(n)* in C programs, its actual allocation amount of heap memory depends on the running environment (due to factors such as alignment, the current first available free slot, etc.) and is somehow non-deterministic before execution. The allocation may fail or may allocate memory with size larger than n (e.g., due to alignment). In such cases, dynamic analysis methods are highly desired.

In this paper, we present a resource-usage-aware fuzzing approach to estimate worst-case resource usage. We use resource-usage amount together with resource-usage relevant coverage to guide the fuzzing process, so as to generate inputs triggering large resource-usage amount. More clearly, we use a different definition of branch coverage and additionally add resource-usage amount to guide the fuzzing process. Moreover, we also leverage semantic patch [11] to make use of static analysis information (including control-flow, function-call, etc.) to instrument the original program. Such information is helpful in aiding the subsequent fuzzing during runtime. We have conducted experiments to estimate worst-case resource usage of various resources in real-world programs, including heap memory, stack depths, sockets, user-defined resources, etc. Preliminary experimental results show the promising ability of our approach in estimating worst-case resource usage in real-world programs, compared with two state-of-the-art fuzzing tools (AFL and MemLock).

2 Approach

In this section, we describe the basic process of our approach (shown in Fig. 1).

2.1 Static analysis and instrumentation

For the target program, we first identify all program locations (i.e., program points) of the calls to resource-usage operations in the program. Such resource-usage operations can be APIs provided by systems or libraries, as well as application programmer-defined APIs. From the point of view of increasing/decreasing

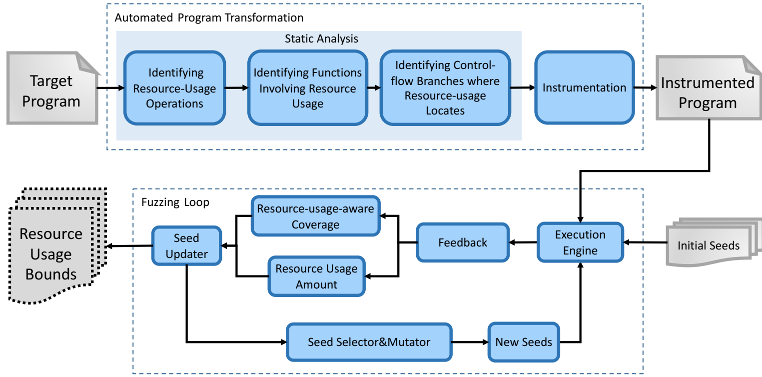

Fig. 1. Workflow of resource-usage-aware fuzzing

resource-usage amount, all operations changing resource-usage can essentially be reduced into allocation (i.e., increasing) and deallocation (i.e., decreasing) operations. To this end, we define two basic modeling functions

- *__RAlloc*(*int n*), to model allocating n number of resources, and
- *__RDealloc*(*int n*), to model deallocating n number of resources.

We will instrument invocations of these two basic modeling functions to explicitly model the resource usage for each resource-usage operation in the original program, according to its semantics. For example, to model *pFile* = *fopen*(...), we will instrument (afterwards) *__RAlloc*(*pFile* != *NULL*?*1* : *0*). To model *free*(*p*), we will instrument (beforehand) *__RAlloc*(*malloc_usable_size*(*p*)), wherein the *malloc_usable_size*(*p*) function (which is a C library function) returns the number of usable bytes in the block pointed to by p. To model the change of call-stack depths, we instrument *__RAlloc*(*1*) and *__RDealloc*(*1*), respectively at the entry and exit (before *return* statement) of each function. Note that each time of resource-usage fuzzing, we consider only one type of resources. The fuzzing engine will track the invocations of *__RAlloc*(*int n*) and *__RDealloc*(*int n*) and capture their parameters to maintain the current amount and the historical peak amount of resource usage at runtime.

On the other hand, many functions and basic blocks in the program are useful for implementing functionality of the program but not relevant to resource usage. Based on this insight, we propose to guide the fuzzing process to cover functions and basic blocks that are relevant to resource usage.

– First, we make use of the call graph of the target program to identify the list of all functions that directly or indirectly invoke resource-usage operations.[3]

[3] Specially, to track stack depth, we first collect a set *FSet* of functions that directly or indirectly call recursive functions. For other functions, we calculate for each function the depth from the *main*() function to that function according to the call graph, and add into *FSet* the top-K percent (e.g., top 30%) functions with large depths.

Then we instrument coverage-label function *__covl*() before the invocation of these functions. We use *__covl*() to identify basic blocks that involve resource usage, which will be further used to define resource-usage-aware coverage.

– Second, for each program block containing invocations of resource-usage modeling functions (i.e., *__RAlloc*(), *__RDealloc*()), label function *__covl*() or exit function *exit*() (as well as similar functions such as *raise*()), we instrument label function *__covl*() before the control-flow branch where this block locates in (e.g., in the *then* branch) and also at the beginning of the block in the other branch (e.g., the *else* branch). We conduct instrumentations of *__covl*() in a bottom-up manner, i.e., from inside to outside blocks.

We leverage program transformation tool Coccinelle [12], to automatically instrument statements invoking resource-usage modeling functions as well as coverage-label function *__covl*() into the original program. Coccinelle is a program matching and transformation engine which allows us to write so-called *semantic patches* [11] for specifying desired code matches and transformations. Particularly, the transformation engine of Coccinelle is defined in terms of control flow, and thus it fits well to instrument coverage-label functions for desired control-flow branches where resource-usage locates.

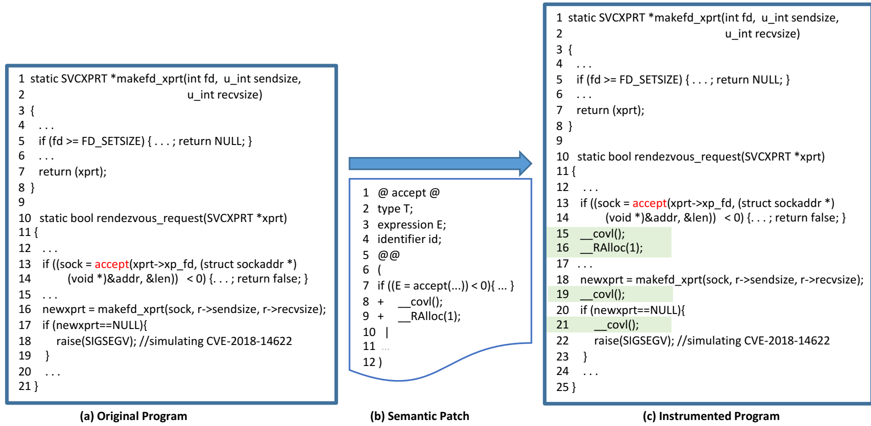

Fig. 2. Example illustration

Example illustration Fig. 2 illustrates the above process via an example (named libtirpc_slice) extracted from an old version of *libtirpc* (that is a Transport-Independent RPC library for Linux) which contains a known CVE vulnerability [4]. The cause of this CVE is that the return value of *makefd_xprt()* was not checked in all instances, which could lead to a crash when the server exhausted the maximum number of available file descriptors. Fig. 2(a) shows the slice extracted from the original code of *libtirpc*. Fig. 2(b) shows part of the semantic patch applied for instrumentation. The instrumented program is shown

[4] https://ubuntu.com/security/CVE-2018-14622

in Fig. 2(c). This program consumes socket connections, e.g., by calling *accept*() as shown on Line 13 in Fig. 2(a). We use semantic patch shown in Fig. 2(b), to instrument resource-usage modeling function *__RAlloc(1)* as well as coverage-label function *__covl*() at the program location when a connection is established successfully. The instrumented code is highlighted in Fig. 2(c).

2.2 Fuzzing loop

Algorithm 1 Resource-usage Aware Fuzzing

Require: an instrumented program P, and a set of initial seeds I_0
Ensure: $(max_res, BuggyS)$ where max_res is the found largest resource usage amount, and $BuggyS$ is a set of test cases triggering resource-usage bugs

```
1: max_res ← 0
2: BuggyS ← ∅
3: SeedQueue ← I0
4: while time not expire do
5:     s ← select(SeedQueue)
6:     s' ← mutate(s)
7:     trace ← execute(s')
8:     n_res ← resPeak(trace)
9:     if n_res > max_res then
10:        max_res ← n_res
11:        SeedQueue ← SeedQueue ∪ s'
12:    else
13:        if find_new_path(trace) then
14:            SeedQueue ← SeedQueue ∪ s'
15:        end if
16:    end if
17:    if trigger_crash(trace) then
18:        BuggyS ← BuggyS ∪ s'
19:    end if
20: end while
21: return (max_res, BuggyS)
```

Algorithm. 1 shows the main procedure of our resource-usage aware fuzzing. The algorithm first selects an input s from the seed pool *SeedQueue*, mutates it and generates a mutant s'. Then, the fuzzer runs the mutant input and monitors its execution. If the mutant input consumes more resources or leads to new resource-usage-aware coverage, it will be added to the seed pool as an interesting input. This process is similar to the process of traditional coverage-based grey-box fuzzers (e.g., AFL). The main difference lies in that resource-usage aware fuzzer uses a different definition of branch coverage and adds resource consumption guidance to retain interesting inputs. Now we give the details.

Resource-usage aware coverage Traditional coverage-based grey-box fuzzers use instrumentation to capture basic block transitions, and log edge coverage information during runtime. For example, AFL uses a random number to represent each basic block, and each transition from one basic block to another is marked by the Exclusive-OR (and right shift) result of the two random values. The identifier of each transition is considered as an address and each time of triggering will increment the count of hits at that address. During runtime, AFL records edge coverage information, including whether the edge has been visited, and the count of hits.

In this paper, we concentrate only on resource usage in a program, while many basic blocks in the program are useful for implementing functionality of the program but not relevant to resource usage. Based on this insight, we log only transitions between those basic blocks that contain resource-usage modeling functions (i.e., *__RAlloc*(), *__RDealloc*()), coverage-label function *__covl*() and *exit*() function. E.g., consider an execution trace $B_1^r, B_2, \ldots, B_{n-1}, B_n^r$ wherein only B_1^r, B_n^r contain aforementioned resource-usage relevant functions. We will log it as a transition from B_1^r to B_n^r, and increase the count of hits of this transition. Resource-usage-aware edge coverage is more delicate and sensitive than traditional edge coverage in identifying different resource usage.

Resource-usage amount guidance When resource-usage aware fuzzer runs an input on the instrumented program, it collects not only the resource-usage aware coverage information, but also resource-usage amount. The fuzzing engine maintains two variables, *resc_cur* and *resc_peak*, to track respectively the current amount and the historical peak amount of resource usage. It captures the parameters of *__RAlloc*(n) and *__RDealloc*(n), and updates the current amount as well as the historical peak amount of resource usage.

Overall guidance mechanism As shown in Algorithm. 1, after execution over an input s', we collect the peak resource usage amount of the running trace through *resPeak*(*trace*) (Line 8). If this input leads to more resource usage, it is added into the seed pool for further mutation (Lines 9-11). Besides, if it leads to new resource-usage aware coverage, it is also added into the seed pool for further mutation (Lines 13-14). In addition, if the input triggers a crash, it is added into *BuggyS* which collects the set of test cases triggering resource-usage bugs.

3 Experiments

We have implemented our approach in a prototype fuzzer named ResFuz [5], based on MemLock [16] which is built on top of AFL [17]. We employ Coccinelle [12] to conduct program instrumentation.

We conduct preliminary experiments on several open-source software, including jasper, openjpeg and yara, which are also part of the benchmark used in [16], as well as the small example libtirpc_slice explained in Fig. 2. More specifically, jasper and openjpeg contain many heap resource operations, while yara contains recursive functions. Moreover, jasper and openjpeg contain many user-defined application-specific resource-usage operations. E.g., jasper uses operations like *jas_malloc*(), *jas_free*() to manage a heap memory pool with a user-configurable size. Similarly, openjpeg uses operations like *opj_malloc*(), *opj_free*() to manage a specific type of heap memory. The small program libtirpc_slice contains socket operations, as explained in Sect. 2.1. We compare ResFuz against other two state-of-the-art fuzzers, namely AFL and MemLock [16]. All our experiments

[5] The artifact is available at https://doi.org/10.5281/zenodo.5894821.

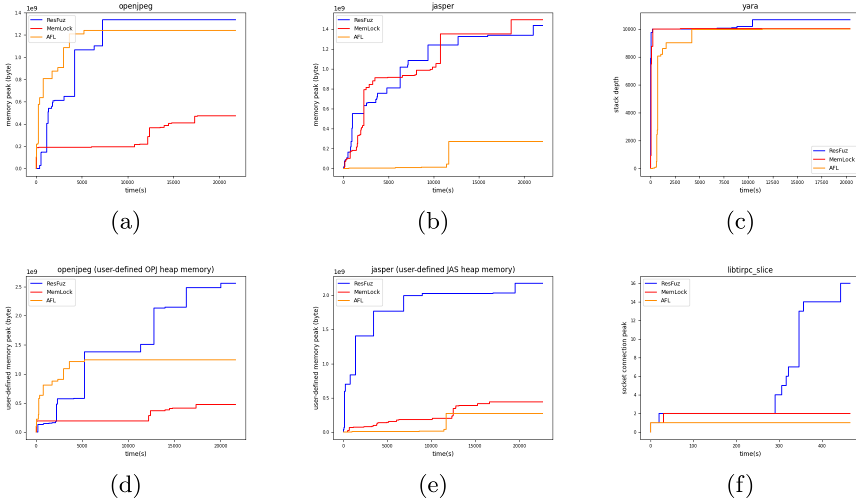

Fig. 3. The growth trend of resource usage

have been performed on machines with an Intel (R) Core (TM) i9-10940X CPU (3.30GHz) and 32GB of RAM under 64-bit Ubuntu LTS 20.04. We run each fuzzer for 6 hours (except 10 minutes for libtirpc_slice) each time, perform each experiment for 3 times, and use their average statistical performance as result.

Fig. 3 depicts the growth trend of the found resource peaks over time through the plots. The vertical axis shows the amount of the peak resource consumed (heaps for jasper and openjpeg, stack depths for yara, sockets for libtirpc_slice). Fig. 3 shows that ResFuz outperforms the two baseline fuzzers in finding large resource consumption for almost all the cases (except for japser shown in Fig. 3(b), for which MemLock performs a little bit better than ResFuz). In particular, as shown in Figs. 3(d-f), for user-defined resources in openjpeg and jasper as well as sockets in libtirpc_slice, ResFuz provides much better results than the other two tools. This is because the guidance mechanism in ResFuz is based on resource-usage amount and resource-usage aware coverage information, which accelerates the process of adding inputs triggering large resource usage into the seed pool. Note that for these user-defined resources and sockets, MemLock uses the consumption of the general heap to guide the fuzzing process, while ResFuz uses respectively the consumption of the specific OPJ heap (in openjpeg), JAS heap (in japser), sockets (in libtirpc_slice) to guide the fuzzing process.

4 Related Work

Using dynamic analysis or fuzzing to find resource-usage relevant bugs has received much attention in recent years. PREDATOR [3] is an automated black box testing tool for detection and identification of local resource-exhaustion vul-

nerabilities in network servers, which computes resource usage profiles for predicting the utilization of every monitored resource for test inputs. Radmin [7] confines the resource usage of a target program from its benign executions to the learned automata and then uses it to detect resource usage anomalies. Both PREDATOR and Radmin do not use fuzzing. MemFuzz [6] uses memory access (rather than memory consumption) instrumentation as addition to branch coverage to guide evolutionary fuzzing. Recently, researchers have drawn attention to the algorithmic complexity vulnerabilities such as SlowFuzz [13], Singularity [15] and PerfFuzz [10]. The basic idea behind is to use the number of executed instructions as the guidance for fuzzing. However, all these works consider time complexity issues.

The most relevant work to our technique is MemLock [16], which uses memory usage guided fuzzing to generate the excessive memory consumption inputs and trigger uncontrolled memory consumption bugs. MemLock also uses memory consumption information to guide the fuzzing process and considers two kinds of memory resources, i.e., stack memory and heap memory. Compared with MemLock, we consider the usage of general resources, including memory, file descriptors, socket connections, user-defined resources, etc. Moreover, MemLock uses default branch coverage of AFL (which considers transitions of all basic blocks) to guide the fuzzing process, while our approach adopts resource-usage-aware coverage (which considers transitions between basic blocks that are relevant to resource usage). In addition, we employ semantic patch to make use of resource-usage relevant call graph and control-flow graph to conduct instrumentation at source code level, while MemLock uses control-flow graph in the same way as AFL (to define branch coverage) and uses call graph only to determine stack memory usage (by instrumenting at the entry and exit of functions).

5 Conclusion and Future Work

In this paper, we present a resource-usage-aware fuzzing approach to estimate worst-case resource usage. It employs resource-usage amount and resource-usage-aware coverage to guide the fuzzing process, for the sake of generating inputs to triggering massive resource usage. Moreover, we employ semantic patches to make use of resource-usage relevant call graph and control-flow graph information to conduct instrumentation, for the sake of aiding the subsequent fuzzing process. We have conducted experiments to estimate worst-case resource usage of various resources in real-world programs, including heap memory, stack depths, sockets, user-defined resources, etc. Preliminary experimental results show its promising ability to estimate worst-case resource usage in real-world programs, compared with two state-of-the-art fuzzing tools.

For future work, we plan to conduct experiments on more real-world programs and over more kinds of resources. We also plan to conduct evaluation comparison with more state-of-the-art fuzzing tools. Furthermore, we will evaluate our approach in detecting resource-usage bugs and security-critical vulnerabilities in real-world programs.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of object-oriented bytecode programs. Theoretical Computer Science **413**(1), 142–159 (2012)
2. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Proceedings of the 17th International Static Analysis Symposium (SAS). pp. 117–133. Lecture Notes in Computer Science, Springer (2010)
3. Antunes, J., Neves, N.F., Veríssimo, P.J.: Detection and prediction of resource-exhaustion vulnerabilities. In: Proceedings of the 19th International Symposium on Software Reliability Engineering (ISSRE). pp. 87–96. IEEE (2008)
4. Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C., Giesl, J.: Analyzing runtime and size complexity of integer programs. ACM Transactions on Programming Languages and Systems (TOPLAS) **38**(4), 1–50 (2016)
5. Carbonneaux, Q., Hoffmann, J., Shao, Z.: Compositional certified resource bounds. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 467–478. ACM (2015)
6. Coppik, N., Schwahn, O., Suri, N.: Memfuzz: Using memory accesses to guide fuzzing. In: Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (ICST). pp. 48–58. IEEE (2019)
7. Elsabagh, M., Barbará, D., Fleck, D., Stavrou, A.: On early detection of application-level resource exhaustion and starvation. Journal of Systems and Software **137**, 430–447 (2018)
8. Flores-Montoya, A., Hähnle, R.: Resource analysis of complex programs with cost equations. In: Proceedings of the 12th Asian Symposium on Programming Languages and Systems (APLAS). pp. 275–295. Lecture Notes in Computer Science, Springer (2014)
9. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: precise and efficient static estimation of program computational complexity. In: Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 127–139. ACM (2009)
10. Lemieux, C., Padhye, R., Sen, K., Song, D.: Perffuzz: Automatically generating pathological inputs. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA). pp. 254–265. ACM (2018)
11. Muller, G., Padioleau, Y., Lawall, J.L., Hansen, R.R.: Semantic patches considered helpful. ACM SIGOPS Oper. Syst. Rev. **40**(3), 90–92 (2006)
12. Padioleau, Y., Lawall, J.L., Hansen, R.R., Muller, G.: Documenting and automating collateral evolutions in linux device drivers. In: Proceedings of the 2008 EuroSys Conference (EuroSys). pp. 247–260. ACM (2008)
13. Petsios, T., Zhao, J., Keromytis, A.D., Jana, S.: Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 2155–2168. ACM (2017)
14. Sinn, M., Zuleger, F., Veith, H.: Difference constraints: An adequate abstraction for complexity analysis of imperative programs. In: Proceedings of the 2015 Formal Methods in Computer-Aided Design (FMCAD). pp. 144–151. IEEE (2015)
15. Wei, J., Chen, J., Feng, Y., Ferles, K., Dillig, I.: Singularity: pattern fuzzing for worst case complexity. In: Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT FSE). pp. 213–223. ACM (2018)

16. Wen, C., Wang, H., Li, Y., Qin, S., Liu, Y., Xu, Z., Chen, H., Xie, X., Pu, G., Liu, T.: Memlock: Memory usage guided fuzzing. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE). pp. 765–777 (2020)
17. Zalewski, M.: American fuzzy lop 2.52b. http://lcamtuf. coredump. cx/afl (2017)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits 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.