Another Path for Software Quality? Automated Software Verification and OpenBSD

Moritz Buhl
genua GmbH
Ludwig-Maximilians-Universität München
mbuhl@moritzbuhl.de

Abstract

CPACHECKER is a platform for software-verification created to be extensible by implementing an interface of configurable program analysis. It comes with various configurations for checking a program for correctness or to produce a counterexample after falsification. This paper displays the strengths and weaknesses of CPACHECKER based on the key insights gained from the *Application of Software Verification to OpenBSD Network Modules* [1]. This is to provide a view on the applicability of formal verification on the OpenBSD source code with the goal of improving its quality.

1 Introduction

The OpenBSD project is proud to ensure a high quality standard. This is due to a combination of a strong review policy, guiding development tools as well as additional audits in the BSD community [2], performance and regression tests[1] [3], and other testing methods like fuzzing [4]. Further promising approaches are applied within the BSD community, e.g. kernel sanitizers [5].

However, a rather unproven way of ensuring software quality – that stands in contrast with the usual approach of simplicity in the OpenBSD project – is formal verification. The amount of effort associated with verification and specification as well as the complexity of time and space limitations usually outgrow the benefits of formally proven source code.

Previous work [1] shows that in addition to the usual weaknesses of automated verification, the OpenBSD kernel causes more complications because of assumptions by the tooling based on a GNU/Linux exclusive view on UNIX. E.g. the runtime environment model used during verification is Linux based and CPACHECKER is officially only running on Linux. Further problems emerge since the BSD kernels differ from Linux in their internal implementations. Moreover, the use of inlined assembler is unsupported and therefore the extensive use of assembler adds more complications.

On the other hand, formal verification offers promising analyses for memory and concurrency safety, reachability and termination checks and also overflow detection. Which means that many program defects can be ruled out or counterexamples can be generated. And as the Linux Driver Verification (LDV) project[2] shows, these methods can be applied successfully.

One tool that enables verification and falsification is CPACHECKER. It uses a combined approach of static code analysis and model checking by implementing an interface for program analysis to test if a program satisfies a given specification.

2 CPAchecker

2.1 Configurable Program Analysis

CPACHECKER implements an interface for configurable program analysis (CPA) [6], [7]. A CPA consists of abstract program states, a relation that connects the abstract state to the program code, and two operations, one that decides on new abstract states, and another one to decide when the analysis will terminate. The program code is converted from the source code to a control-flow graph (CFG) to apply a graph algorithm.

Each CPA has different trade-offs in accuracy and supported features. It is possible to combine multiple CPA and use them in combination on different parts of

[1] http://bluhm.genua.de

[2] http://linuxtesting.org/ldv

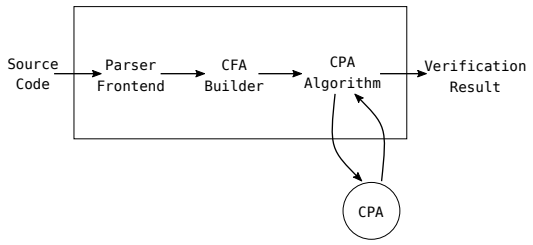

Figure 1: The workflow of CPACHECKER

```
Inline assembler ignored, analysis is probably
    unsound!
Assuming external function err to be a pure
    function.
Function pointer *(&putc) with type int (*)(
    int, FILE *) is called, but no possible
    target functions were found.
Unrecognized C code ...
Using unsound approximation of ints with
    unbounded integers and floats with
    rationals for encoding program semantics.
```

Figure 2: Example error messages from CPACHECKER

the program, this is called partial verification. Examples for the various features of a program and how these affect the tooling can be viewed online[3].

Figure 1 shows the process of verifying a program: first, its source code is converted to a CFA. Then a graph algorithm runs a (possibly multiple) CPA on it and checks if the program satisfies the given specification. If multiple CPA are used it is possible to make them interact with each other or to refine the analysis based on previous results [8].

2.2 Setup and Usage

CPACHECKER[4] uses the Java JVM and therefore is mostly platform independent with exception to the dynamic libraries that are not currently available on OpenBSD. To set up CPACHECKER, a Java environment is needed and the sources need to be built with Apache Ant. On OpenBSD adjusting the maximum data size limit in the `login.conf` file is recommended, especially when analyzing bigger C programs. A port is currently not available.

Once set up, CPACHECKER requires a specification or a configuration file to analyze a C program: `cpa.sh [-config CONFIG_FILE] [-spec SPEC_FILE] SOURCE_FILES`. A selection of configurations is available in the `config` folder.

Source code requires preprocessing before it can be analyzed. GCC and Clang provide an `-E` option to use the C preprocessor on a source file, alternatively the `-preprocess` switch of CPACHECKER can be set. To preprocess the OpenBSD kernel code according to its configuration, it is necessary to use the kernel build system. The necessary modifications are available online[5].

On OpenBSD the following properties either need to be set in a configuration file or with the `-setprop` flag, as other SMT solvers are not available:

```
solver.solver                       SMTInterpol
cpa.predicate.encodeBitvectorAs     INTEGER
cpa.predicate.encodeFloatAs         INTEGER
```

While using CPACHECKER, it is likely to run into the warnings and errors mentioned in Fig. 2. Especially when working with kernel code, warnings will occur due to inlined assembler. These places need to be looked at individually and need to be worked around. Other warnings require implementations of known C functions because a mechanism for linking files is missing. And other times either an analysis cannot work with a specific C construct or the conversion to a CFA was erroneous. The last warning mentioned appears on OpenBSD because other SMT solvers are not supported.

2.3 Working around Problems

To still receive a result from CPACHECKER, it is necessary to work around these errors. This is mostly achieved by adjusting the source code – which means changing it to an extent that it is not easy to prove to behave exactly the same. E.g. by replacing a call to a function pointer with the actual function because function pointers are not always tracked. The LDV project too has to work around the weaknesses and does so by adding another compilation layer with an intermediate language.

Application of Software Verification to OpenBSD Network Modules [1] had the plan to refind previous errata with the prospect of applying the same strategies in the future to find new bugs. But it quickly became clear that CPACHECKER is not practicable for this task as it is not battle-tested on real source code. It is possible

[3] `http://sv-comp.sosy-lab.org`
[4] `http://cpachecker.sosy-lab.org`
[5] `http://github.com/bluhm/preproc`

to refind errors like a double-`free(3)` after swapping the memory management implementation, removing inline assembler, replacing calls to function-pointers, fixing C syntax parsing in CPACHECKER and manually merging compile units together.

3 Conclusion

Using CPACHECKER to easily find new bugs is currently not imaginable. It is possible to reproduce already known bugs in the kernel but as the kernel functions used for resource management require individual abstractions, this is associated with manual labor.

In addition to this, the bugs in CPACHECKER make it uneasy to use on real programs. It tries to support the ISO/IEC 9899:1999 (C99) standard and does so by reprogramming the bugs, other compilers and parsers already made. The manifold configurability is great for developing new approaches but might cause problems with usability when applying it on real programs, as the barrier to entry is increased with the knowledge required on each CPA. The lack of use with real programs is the main reason for this.

4 Future Work

Because of the mentioned problems, it is necessary to use CPACHECKER on real userland programs. Starting with small POSIX programs like `true`, `yes` or `w` before considering to take a look kernel code might be a better approach to fix CPACHECKER.

Especially since OpenBSD introduced a dynamic approach with `pledge(2)` that ensures that a system call cannot be called again after a pledge, it would be interesting to see if a CPA can find violations of a pledge with a static approach.

Another complicated problem that accumulates a lot of different possible states that is known to be tricky is multiprocessing. Approaches in CPACHECKER exist to verify POSIX thread programs. If this can be applied successfully on userland programs, it might be interesting to see, if a similar approach can be used on the kernel.

References

[1] M. Buhl, *Application of software verification to OpenBSD network modules*, Bachelor's Thesis, LMU Munich, Software Systems Lab, Sep. 2018. [Online]. Available: `http://www.moritzbuhl.de/bachelor-thesis/thesis.pdf`.

[2] M. Villard, *Network security audit*, Article, May 2018. [Online]. Available: `http://blog.netbsd.org/tnf/entry/network_security_audit`.

[3] J. Klemkow, *Openbsd testing infrastructure behind bluhm.genua.de*, Presentation, Paris: EuroBSDcon, Sep. 2017. [Online]. Available: `http://klemkow.org/eurobsdcon_2017_obsd_test_infrastructure.pdf`.

[4] A. Lindqvist, *Fuzzing the openbsd kernel*, Presentation, BSD Users Stockholm, Sep. 2018. [Online]. Available: `http://www.openbsd.org/papers/fuzz-slides.pdf`.

[5] S. Muralee and K. Rytarowski, *Taking netbsd kernel bug roast to the next level : Kernel sanitizers*, Presentation, Bucharest: EuroBSDcon, Sep. 2018. [Online]. Available: `http://netbsd.org/~kamil/eurobsdcon2018_ksanitizers.html`.

[6] D. Beyer, T. A. A. Henzinger, and G. Théoduloz, "Configurable software verification: Concretizing the convergence of model checking and program analysis", in *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007, Berlin, July 3-7)*, W. Damm and H. Hermanns, Eds., ser. LNCS 4590, Springer-Verlag, Heidelberg, 2007, pp. 504–518.

[7] D. Beyer and M. E. Keremoglu, "CPACHECKER: A tool for configurable software verification", in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011, Snowbird, UT, July 14-20)*, G. Gopalakrishnan and S. Qadeer, Eds., ser. LNCS 6806, Springer-Verlag, Heidelberg, 2011, pp. 184–190. [Online]. Available: `https://cpachecker.sosy-lab.org`.

[8] D. Beyer, S. Löwe, and P. Wendler, "Refinement selection", in *Proceedings of the 22nd International Symposium on Model Checking of Software (SPIN 2015, Stellenbosch, South Africa, August 24-26)*, B. Fischer and J. Geldenhuys, Eds., ser. LNCS 9232, Springer-Verlag, Heidelberg, 2015, pp. 20–38. [Online]. Available: `https://www.sosy-lab.org/research/cpa-ref-sel/`.