CHEF: A Configurable Hardware Trojan Evaluation Framework

Christian Krieg and Daniel Neubacher
Institute of Computer Technology
Vienna University of Technology
Gusshausstr. 27–29 / 384, 1040 Wien, Austria
christian.krieg@alumni.tuwien.ac.at, neubacher@ict.tuwien.ac.at

ABSTRACT

Evaluating approaches to hardware Trojan detection often involves the use of benchmark hardware designs in order to maintain comparability and reproducibility of experiments. However, such benchmark designs may have to be adjusted to the tool chain that is used for evaluation. Also the tool chain may have to be adjusted to an individual benchmark design. Such adjustments can be time consuming, particularly if these adjustments have to be made multiple times. In order to maintain reproducibility, such adjustments have to be documented in a way that an experiment can be repeated. It is also important that the inclusion of malicious functionality is clearly documented. In this work, we present a methodology to clearly document changes to benchmark designs and tool chains. We facilitate the advantages of the extensible markup language (XML) in order to create a set of rules that is applied to the original benchmark design or tool chain which then yields the desired adjusted benchmark or tool chain. Additionally, we provide facilities to describe an experiment in a structured way, which allows to automatically perform such an experiment. Also, it is possible to share an experiment description such that an experiment can be repeated by peers. That way, the derived conclusions from such an experiment can be dramatically strengthened.

Keywords

Experiment automation, test data normalization, hardware Trojan detection, electronic design automation

1. INTRODUCTION

Hardware Trojans have been under active research in the past few years [12, 7]. Hardware Trojans are digital or analog/mixed-signal systems that incorporate malicious functionality. Malicious functionality is functionality that is unspecified, undocumented and serves a shadow purpose besides legitimate functionality.

WESS'15, October 04-09 2015, Amsterdam, Netherlands
 ACM 978-1-4503-3667-3/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2818362.2818371

The payload functionality of hardware Trojans is usually triggered on rare events or after a considerably long time span in order to circumvent detection during functional tests. Hardware Trojans can be inserted at any stage of the development process. In this work, we primarily focus on hardware Trojans inserted at the design phase. In order to evaluate approaches aiming at the detection of Hardware Trojans often benchmark designs are used to which malicious functionality is added. Such benchmark designs may require to be adjusted to the tool chain in use, in order to maintain compatibility. Also, the tool chain (or, the *evaluation flow*) in use may have to be adjusted to any individual benchmark, e.g. load different cell libraries or use a different front end to read the design. In this paper, we present a framework which makes it possible to adjust each benchmark and the evaluation flow to each individual benchmark design. This way, experiments can be automated easily and are well-documented such that reproducibility of experiments is properly ensured.

2. THE FRAMEWORK

We present a configurable hardware Trojan evaluation framework (CHEF) that enables automated experimentation in scenarios where malicious functionality is incorporated into hardware designs at design-level. Basically, the framework is divided into two parts:

1. Benchmark normalization
2. Automated experimentation

Benchmark normalization is vital in order to adjust the benchmarks to the evaluation flow and vice versa. Automated experimentation refers to the formalized definition of the evaluation flow and the specification of the test data.

2.1 Benchmark normalization

Certain issues can arise when using publicly available benchmark designs. Prominent examples for publicly available benchmark suites are:

- ISCAS'85 combinational benchmark circuits [4]
- ISCAS'89 sequential benchmark circuits [3]
- ITC'99 register transfer level (RTL) benchmark models [6],
- OpenCores.org, a free collection of open-source hardware intellectual property (IP) cores [9]

- IWLS 2005 benchmark collection, basically a collection of the benchmarks mentioned above [1]
- Trust-HUB.org [13] provides a collection of hardware description language (HDL) models that incorporate malicious functionality at different levels of abstraction (from RTL to gate level).

Let us examine the following scenario: an algorithm is developed that analyzes a digital circuit. In order to evaluate the algorithm, it is applied to various benchmark circuits from different benchmark suites. The benchmark circuit is to be pre-processed and synthesized to a netlist representation. Evaluation is to be carried out in an automated fashion by repeatedly applying the algorithm to the different benchmarks. Possible issues are:

- **Different abstraction levels of the benchmarks:** The abstraction levels of the benchmarks are not uniform. Therefore, the desired evaluation flow changes from benchmark to benchmark in order to yield the target abstraction level
- **Benchmarks can have bugs:** Some benchmarks are provided as entire projects, comprising numerous design files in an arbitrary directory structure. Some configurations of such projects can require to specify paths on the user's machine. It is possible that such paths are hard-coded by the provider and therefore have to be manually adjusted by the end-user. An example for such a bug is to specify include files by absolute paths. These paths are only valid on the machine of the benchmark developer, but they are not at the machine of the end-user
- **Benchmarks can be named inconsistently:** When following a classification scheme to name the respective benchmarks, it can happen that inconsistencies are introduced when doing so. These can lead to errors if the end-user intends to use such a benchmark (e.g., "file not found"). Examples for such inconsistencies are upper-case/lower-case variations, forgotten hyphens, and typographical errors

This is where our evaluation framework comes into play. With our framework, we aim at reducing these issues by solving them once and make the solutions persistent in a "*normalization database*". Therefore, normalizing benchmarks can be automatically repeated at any time. This requires a very low time effort from the end-user. In addition, the normalization database can be shared, such that other end-users also benefit from normalized benchmark designs.

Especially in the case of evaluating algorithms aiming at detecting hardware Trojans, it can be hard to exactly reproduce the inclusion of malicious functionality. One possibility is to label malicious functionality in the hardware description with comments, or choosing names for signals and registers that enable to identify malicious functionality. However, this way of indicating malicious functionality is not suitable when evaluating code review processes, or when challenging algorithms in blind studies, because this way malicious functionality is overt (usually, a malicious designer seeks to hide malicious functionality).

Therefore, our framework provides facilities to insert malicious functionality into the original design in a clearly defined way. This has two major advantages: First, the designer of benchmark designs that incorporate malicious functionality does not have to think about how to store malicious parts of the design. For instance, the benchmark designs available from Trust-HUB.org [13] have complete directory trees for both the Trojan-free and the Trojan-infested design. This way of organizing Trojan storage can be optimized in terms of storage efficiency, but also in terms of readability. As a Trojan may be spread over the entire file set of the design, it may be hard for researchers to find each and every portion of the Trojan immediately (in order to study the Trojan implementation). Our framework "patches" the original design, therefore clearly indicating the Trojan and also avoiding parallel directory trees. The second major advantage is that users of the benchmark designs can easily choose between Trojan-free and Trojan-infested versions of the design without changing directory trees for the project, which helps in maintaining consistency.

Finally, obtaining benchmark designs from online sources can be time-consuming, especially if they are available as single compressed archives instead of one compressed archive that contains the entire benchmark suite. It can be even more time-consuming if one single benchmark design is split into multiple compressed archives. Also, the compression format of archives in one suite can vary, thus requiring various software tools to decompress the archives. In order to circumvent these issues, our framework provides functionality to automatically download any benchmark design from any site on the Internet and it allows to specify individual commands for decompressing each benchmark design. This facility greatly improves time efforts necessary to obtain benchmarks from the Internet, especially when this has to be done multiple times.

2.2 Automated experimentation

For benchmark designs that are normalized, our framework provides facilities to automatically conduct experiments on these benchmark designs. The CHEF allows to specify per-benchmark batch scripts, and it is also possible to specify batch scripts that are applied to a set of benchmark designs.

Batch scripts describe the evaluation flow to which the benchmark design is fed. A batch script can incorporate calls to synthesis tools, data analysis tools and data visualization tools. In order to provide maximum generality and flexibility, any tool is supported that provides a command-line interface.

In our example above, the evaluation flow incorporates the following tools:

1. A synthesis tool that transforms the RTL descriptions of a benchmark into a gate-level netlist
2. The application that implements the algorithm. It emits experimental data in comma-separated values (CSV) format

This way, experiments can be easily repeated and shared with the scientific community, which leads to improved reproducibility and the possibility to acknowledge scientific conclusions by others.

In the following, we explain in detail the functionality of the CHEF framework. While the CHEF can be used for any benchmark suite, even for custom benchmark designs, in this work we focus on its usage with the hardware Trojan benchmark designs available at `trust-hub.org` [13].

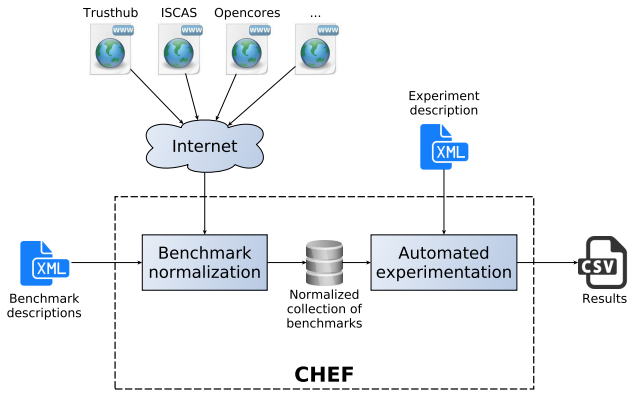

Figure 1: Overview of CHEF

3. METHODOLOGY

Figure 1 provides an overview over the components of the CHEF framework. The two parts of the framework are implemented as one Perl script. In order to perform benchmark normalization, we provide the script information about which modifications have to be applied to a respective benchmark. This is done via an extensible markup language (XML) database, that holds one record for each benchmark. In the following, we call such an XML record a *benchmark record*.

3.1 Benchmark obtaining and normalization

In order to load benchmark designs from any source (for simplicity, we consider the case of an Internet location), a benchmark record holds an *archive* element for every file a benchmark consists of. Every archive element itself contains a *url* element which specifies where to download the benchmark from. In case that the archive is compressed, the corresponding decompression command is stored in a *decompress* element. Listing 1 shows the benchmark record for an Ethernet core taken from the TrustHUB benchmark suite [13]. Lines 3 to 10 reveal that the benchmark is available as a two-part compressed archive. The corresponding decompression command is given in line 11. In order to normalize the benchmark design, a *normalize* element stores commands to do so. Lines 12 to 15 in Listing 1 for example show a command to make all directory and file names lowercase. The *normalize* element can also contain directives to adjust the benchmarks to a certain tool chain. One such example is given in Listing 2, which shows a snippet of the *normalize* element of the benchmark record of an advanced encryption standard (AES) core. The *yosys* [14] synthesis suite [15], which we use for our synthesis tasks, requires the file name to be the same as the module name in order to automatically detect the given module. Therefore, symbolic links are created to point to the correct files (Listing 2).

3.2 Bug fixing

As already pointed out, a benchmark design can have bugs that prevents it from being used on the end-user's machine. An example for a bug-fixing directive is given in Listing 3, which is a part of the benchmark record of an RS232 transceiver and reflects the *bugfix* element of the record. The bug fix in this example is to replace an absolute path that points to an include file by a relative path via a patch.

3.3 Trojan insertion

When evaluating approaches to Trojan detection, it is vital that the insertion of malicious functionality is clearly defined and therefore reproducible. This requires that every piece and portion of the malicious functionality is known, and also its target location. We solve the issue by storing a patch that incorporates malicious functionality. This is shown in Listing 1, lines 16 to 39. The patch is part of a *trojan_insertion* element and is applied to a temporary copy of the original design. All experiments that operate on the Trojan-infested version of a design consult this copy. This way, it is easy to perform experiments on both the Trojan-free and Trojan-infested versions of a benchmark design, while keeping legitimate and malicious functionality well-separated.

3.4 Automated experiments

In order to automate experiments targeted at Trojan detection, we provide the framework with an experiment description (as depicted in Figure 1). An example for such an experiment description is provided in Listing 4. In this example, we perform a batch evaluation that processes for each benchmark design the same evaluation flow. In our example, we call the yosys synthesis tool and provide it with a yosys script that contains the necessary commands to process the benchmark designs (lines 3 to 5).

An example for such a yosys script is provided in Listing 5. Instead of yosys, any other tool can be used. The example in Listing 5 reads the the benchmark design and a cell library, synthesizes processes and flattens the design. Then, the algorithm under evaluation (`trojan_detect`) is called, which emits experimental results to the specified output file. The script uses templates which are surrounded by '`%`'. The templates are substituted by the CHEF as specified in the experiment description.

The experiment description further contains information on the location of the results file (lines 6 to 7 in Listing 4). Lines 8 to 27 specify the benchmarks that are used for the evaluation. Each benchmark is specified by a *benchmark* element that contains a *uid* element that refers to the respective benchmark description (as, for example, the benchmark description in Listing 1) and a path to the directory in which the benchmark description is located.

To sum up, the Perl script that implements the CHEF reads the benchmark and experiment description. The benchmark designs are downloaded if needed and normalized as specified in the benchmark description. Next, the experiments are performed on the benchmark designs. An example call to the Perl script `chef.pl` is given in Listing 6. In this example, the script reads any benchmark description file that is stored in the directory `benchmarks/xml/` and the experiment description `trojan_detection.xml` is passed to the script via the command line.

4. RELATED WORK

Automating scientific experiments is essential in nearly all research areas. However, to the best of our knowledge, it seems that there is no tool that facilitates both benchmark normalization and reproducible, automated experimentation in the hardware Trojan detection community. In [5], Caldwell et al. state that in the VLSI CAD hypergraph partitioning community, the quality of experimental results always depend on the quality of the test data. A poorly cho-

sen data set can lead to misleading results. Therefore, the authors call for reproducible descriptions of experiments in order to strengthen conclusions derived from scientific experiments. While the authors theoretically examine the impact of reproducible experiments, no tangible software tool or framework is presented that implements their methodology. Bartolini et al. present WS-TAXI, a tool that implements test automation for web services [2]. The XML-based framework generates test suites and test inputs for web services under test. A web service is automatically generated from a given XML schema. This is a major difference to our approach, as we are normalizing test data and create reproducible experiment descriptions. In [10], Quereilhac et al. present the NEPI experiment management framework that allows to automate experiments for multi-host environments in the network simulator ns-3 [8]. By specifying all configuration tasks within an experiment description, configuration efforts and manual work is heavily reduced. Also, the framework supports to perform experiments without requiring advanced system administration skills to set up experiments. While the NEPI framework basically covers similar functionality as the CHEF, our approach is specialized for automating experiments in the field of electronic design automation (EDA). Another approach to benchmark design evaluation in EDA is followed by OA Gear which is a set of tools and libraries built on top of the OpenAccess application programming interface (API) and reference database [1]. OA Gear provides tools to place components and analyze the timing of the resulting circuits. However, the last activity in the development of OA gear was observable in 2004 [11], therefore development can be considered inactive.

5. CONCLUSIONS AND FUTURE WORK

We presented a framework that allows to specify experiment descriptions that enable to repeat and share experiments with the scientific community. Reproducible experiments greatly strengthen the conclusions that are derived from experimental results, therefore providing an essential benefit to the quality of such experiments. Besides experiment descriptions, also the test data (i.e., benchmark designs) is described and modifications made to the test data is clearly documented by benchmark descriptions.

While, at present, the framework provides valuable support in increasing time efficiency for experiments, there are still open issues to be solved. At the moment, benchmark and experiment descriptions have to be created by hand, and the Perl script has to be called via the command line. In order to increase user friendliness, we are working on a graphical user interface (GUI) for the CHEF framework that simplifies these tasks. Also, it would be helpful to establish an online infrastructure for sharing benchmark designs that incorporate malicious functionality. In order to foster the exchange of knowledge about hardware Trojan implementations, we plan to establish a service similar to rich site summary (RSS), that offers hardware Trojan implementations to subscribers. This way, it is also easy to provide updates to such benchmark designs and to notify subscribers when new benchmarks or updates are available

Acknowledgment

This work has been supported under contract no. P1405088 by the Austria Wirtschaftsservice GmbH (aws) PRIZE program.

References

[1] C. Albrecht. *IWLS 2005 Benchmarks*. Presentation given at the fourteenth International Workshop on Logic and Synthesis. June 2005. URL: http://iwls.org/iwls2005/benchmark_presentation.pdf.

[2] C. Bartolini et al. "WS-TAXI: A WSDL-based Testing Tool for Web Services". In: *Software Testing Verification and Validation, 2009. ICST '09. International Conference on*. 2009, pp. 326–335. DOI: 10.1109/ICST.2009.28.

[3] F. Brglez, D. Bryan, and K. Kozminski. "Combinational profiles of sequential benchmark circuits". In: *Circuits and Systems, 1989., IEEE International Symposium on*. 1989, 1929–1934 vol.3. DOI: 10.1109/ISCAS.1989.100747.

[4] F. Brglez and H. Fujiwara. "A Neutral Netlist of 10 Combinational Benchmark Circuits and A Target Translator in FORTRAN". In: *International Symposium on Circuits and Systems, Proceedings of the*. 1985.

[5] A. Caldwell et al. "Hypergraph partitioning for VLSI CAD: methodology for heuristic development, experimentation and reporting". In: *Design Automation Conference, 1999. Proceedings. 36th*. 1999, pp. 349–354. DOI: 10.1109/DAC.1999.781340.

[6] F. Corno, M. Reorda, and G. Squillero. "RT-level ITC'99 benchmarks and first ATPG results". In: *Design Test of Computers, IEEE* 17.3 (2000), pp. 44–53. ISSN: 0740-7475. DOI: 10.1109/54.867894.

[7] C. Krieg et al. "Hardware Malware". In: *Synthesis Lectures on Information Security, Privacy, and Trust* 4.2 (2013), pp. 1–115. DOI: 10.2200/S00530ED1V01Y201308SPT006.

[8] *ns-3*. URL: https://www.nsnam.org/ Sept. 2, 2015.

[9] *opencores.org*. URL: http://opencores.org/ Aug. 28, 2015.

[10] A. Quereilhac et al. "Automating Ns-3 Experimentation in Multi-host Scenarios". In: *Proceedings of the 2015 Workshop on Ns-3*. WNS3 '15. Barcelona, Spain: ACM, 2015, pp. 1–8. ISBN: 978-1-4503-3375-7. DOI: 10.1145/2756509.2756513. URL: http://doi.acm.org/10.1145/2756509.2756513.

[11] *SourceForge OpenAccess Gear*. URL: http://sourceforge.net/projects/oagear/files/OAGear/0.10/ Sept. 2, 2015.

[12] M Tehranipoor and F Koushanfar. "A Survey of Hardware Trojan Taxonomy and Detection". In: *Design Test of Computers, IEEE* 27.1 (2010), pp. 10–25. ISSN: 0740-7475. DOI: 10.1109/MDT.2010.7.

[13] *trust-HUB.org*. URL: https://www.trust-hub.org/index.php June 30, 2015.

[14] C. Wolf and J. Glaser. "Yosys - A Free Verilog Synthesis Suite". In: *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*. 2013. URL: http://www.clifford.at/yosys/files/yosys-austrochip2013.pdf.

[15] *YOSYS*. URL: http://www.clifford.at/yosys/ June 30, 2015.

APPENDIX

Listing 1: Benchmark record of the TrustHUB hardware Trojan benchmark *EthernetMAC10GE-T720* obtained from [13]

```
<?xml version="1.0" encoding="UTF-8"?>
<benchmark uid="a64d4319-57e4-4739-81ea-bffd6a2a3236" suite="trusthub" name="EthernetMAC10GE-T720" language="verilog">
  <archive>
    <checksum type="sha256"></checksum>
    <url>https://www.trust-hub.org/index.php/resources/674/download/EthernetMAC10GE-T720.part01.rar</url>
  </archive>
  <archive>
    <checksum type="sha256"></checksum>
    <url>https://www.trust-hub.org/index.php/resources/675/download/EthernetMAC10GE-T720.part02.rar</url>
  </archive>
  <decompress>unrar x EthernetMAC10GE-T720.part01.rar</decompress>
  <normalize>
    #!/bin/bash
    find . -depth -exec rename 's/(.*)\/([^\/]*)/$1\/\L$2/' {} \;
  </normalize>
  <trojan_insertion>
    diff -rupN trojanfree/xge_mac_scan.v trojaninserted/xge_mac_scan.v
    --- trojanfree/xge_mac_scan.v       2015-08-20 09:02:18.827949923 +0200
    +++ trojaninserted/xge_mac_scan.v   2015-08-20 09:02:18.871949924 +0200
    [...]
    +  // Trigger ----------------------
    +    AND2X1 Trojan1 (.IN1(n22798), .IN2(n130965), .Q(Tj_OUT1));
    +    AND2X1 Trojan2 (.IN1(n130261), .IN2(n131096), .Q(Tj_OUT2));
    +    AND2X1 Trojan3 (.IN1(n130129), .IN2(n131471), .Q(Tj_OUT3));
    +    AND2X1 Trojan4 (.IN1(n131545), .IN2(n130687), .Q(Tj_OUT4));
    +    NOR4X0 Trojan1234_NOT (.IN1(Tj_OUT1), .IN2(Tj_OUT1), .IN3(Tj_OUT3), .IN4(Tj_OUT4), .QN(Tj_OUT1234));
    +    AND2X1 Trojan5 (.IN1(n131167), .IN2(n130054), .Q(Tj_OUT5));
    +    AND2X1 Trojan6 (.IN1(n130246), .IN2(n130951), .Q(Tj_OUT6));
    +    AND2X1 Trojan7 (.IN1(n130721), .IN2(n131396), .Q(Tj_OUT7));
    +    AND2X1 Trojan8 (.IN1(n130636), .IN2(n131284), .Q(Tj_OUT8));
    +    NOR4X0 Trojan5678_NOT (.IN1(Tj_OUT5), .IN2(Tj_OUT6), .IN3(Tj_OUT7), .IN4(Tj_OUT8), .QN(Tj_OUT5678));
    +    NAND2X0 Trojan_CLK_NOT (.IN1(Tj_OUT1234), .IN2(Tj_OUT5678), .QN(Tj_OUTClock));
    +    RDFFNX1  Counter_BIT1 (.D(1'b1),␣.CLK(Tj_OUTClock),␣.RETN(1'b1), .Q(Tj_Trigger), .QN(Tj_TriggerN));
    +
    +  // Payload ----------------------
    +  OR2X1 Trojan_Payload (.IN1(Tj_Trigger), .IN2(tx_dq0_crc32_d64_21_), .Q(Tj_Payload));
    +
    [...]
  </trojan_insertion>
  [...]
</benchmark>
```

Listing 2: Parts of the *normalize* element contained in the benchmark record of the TrustHUB hardware Trojan benchmark *AES-T100* obtained from [13]

```
  <normalize>
    cd aes-t100/src/tjfree
    ln -s round.v one_round.v
    ln -s table.v table_lookup.v
    cd -
    cd aes-t100/src/tjin
    ln -s lfsr.v lfsr_counter.v
    ln -s round.v one_round.v
    ln -s table.v table_lookup.v
    cd -
  </normalize>
```

Listing 3: Parts of the *bugfix* element contained in the benchmark record of the TrustHUB hardware Trojan benchmark *RS232-T100* obtained from [13]

```
<bugfix>
  diff -rupN rs232-t100/src/uart.v rs232-t100new/src/uart.v
  --- rs232-t100/src/uart.v   2015-03-04 17:32:29.000000000 +0100
  +++ rs232-t100new/src/uart.v        2015-06-09 15:51:53.318095854 +0200
  @@ -13,7 +13,7 @@ module uart       (       sys_clk,

  -`include "/home/salmani_h/Trust_HUB/Trojan_Inserted/inc.h"
  +`include "inc.h"
  [...]
</bugfix>
```

Listing 4: Example of an experiment description

```
<?xml version="1.0" encoding="UTF-8"?>
<workbench>
  <eval_script>
    yosys -s trojan-detection.ys
  </eval_script>
  <results_dir>results</results_dir>
  <results_file>trojan-detection_results.csv</results_file>
  <benchmark>
    <uid>c13c5ccf-04d3-44f9-853e-06380e4dc422</uid>
    <path>benchmarks/xml</path>
  </benchmark>
  <benchmark>
    <uid>1122edfe-ac50-423a-b408-874a7d42c3f2</uid>
    <path>benchmarks/xml</path>
  </benchmark>
  <benchmark>
    <uid>5d73e50c-b733-4fee-aa78-5234b8a44e9d</uid>
    <path>benchmarks/xml</path>
  </benchmark>
  <benchmark>
    <uid>dd6cdb38-0ff9-4a20-a762-d09b634d5df4</uid>
    <path>benchmarks/xml</path>
  </benchmark>
  <benchmark>
    <uid>2f8d2253-2edd-4323-bf75-bfaeee2cb743</uid>
    <path>benchmarks/xml</path>
  </benchmark>
</workbench>
```

Listing 5: Example of an evaluation script (`trojan-detection.ys`) that is provided to the yosys synthesis tool

```
read_verilog %BENCHMARKTOPFILE%
read_verilog -lib %CELLLIBS%
hierarchy -libdir %BENCHMARKDIRECTORY% -top %TOPMODULENAME%
proc
flatten
clean

trojan_detect -setname %BENCHMARKNAME%
trojan_detect -output %RESULTSFILE%
```

Listing 6: Example usage of the Perl script `chef.pl` that implements the CHEF framework

```
./chef.pl -p benchmarks/xml/ -W trojan-detection.xml
```