Verification of Xen tool (xl console) using SMACK

Pankaj Kumar

School of Computing, University of Utah, USA
pankajk@cs.utah.edu

Abstract. Hypervisor[7] is a kind of software which allows booting of multiple virtual machines on a single host hardware. Xen[2] is widely used hypervisor these days. Xen's tool stack is a part of trusted computing base. An error in the Xen tool stack can open holes in the system for unauthorized hypervisor and host hardware access. Conventional code coverage tools might fail to uncover bugs in such cases as developer has to manually create conditions to cover all the branches in the code. There are possibilities that some paths might still remain unexplored using such tools. Formal verification methods can be employed to explore all the possible branches in code and expose bugs in such cases. One such tool is SMACK[1] which can be used to verify correctness using formal verification methods.

1 Introduction

Xen is complex piece of software. Its verification using conventional code coverage tools cannot guarantee the correctness. Tools like Bullseye[12] works at runtime and one has to manually generate the conditions to cover all the branches in the source code. There is always a possibility that some source code branch remain unexplored and bugs might remain hidden in the code. Formal verification techniques plays a very good role in such cases because conditions are generated automatically to cover all the branches in the source code and hence expose the bugs.

In this section, I will be giving basic overview of Xen hypervisor, Xen console (xl console) & SMACK.

1.1 Xen Hypervisor

Xen is a Type-1 bare metal hypervisor. It boots directly on the top of host hardware. First virtual machine or the control domain hosts a tool stack which is used to manage other virtual machines like starting and stopping virtual machines. Tool stack's source code is the part of Trusted Computing Base. As a part of my project, I have started with the verification of small part of tool stack which can be extended in near future.

1.2 Xen Console - xl console

Xen console tool consists of a daemon (xenconsoled) and a helper (xenconsole) binary. Daemon (xenconsoled) is launched during priviliged virtual machine (control domain) startup. While user has to execute helper (xenconsole) binary to attach console device to the virtual machine. On execution, helper (xenconsole) forwards the console device request to daemon which sets up a console device and writes the terminal device information to xenstore database. Helper (xenconsole) then read this terminal device name from database and starts using it as console device for the virtual machine. Xen console is written in the C programming language and total LOC is around 2000.

1.3 SMACK

SMACK[1] is a formal verification tool which fully supports the verification of the softwares written in the C programming language and handles pointers arithmetic and memory allocation efficiently. It has been developed at the SOARlab[3], University of Utah. It takes LLVM[6] IR (Intermediate Representation) as input and parses it to create another IVL (Intermediate Verification Language) for verifier which can be Boogie[6] or Corral[5]. This IVL contains statements like assumptions, assertions etc. Verifier checks all the statements for validity and return results. For reported bugs, a trace is printed which can be used by a developer to locate the bug location.

2 Major Milestones

This sections lists major milestones in the project.

2.1 Feasibility

SMACK works on LLVM IR. In order to use SMACK on some source code, following properties should be satisfied:

1. Source code should be clang compatible
2. Source code should be compiled without any optimization (O0)
3. System should be closed.

Xen console tool source code is compatible with clang. I have used clang 3.7 for my project because this clang version is compatible with SMACK. Also, source code can be compiled without any optimization (O0 - No optimization). In order to close the system, we need to provided proper models.

2.2 Compilation & Bitcode file generation

To compile the Xen console source code, I made use of WLLVM[4] package. WLLVM is a compiler wrapper and it works in two steps:

1. Create bitcode file along with every object file and store the location of the bitcode file in a new dedicated section in the object file.
2. During linking, concatenate all the bitcode file paths in one dedicated section in the final binary.
3. Use extract-bc tool to extract the bitcode contents from the binary file using the dedicated section.

2.3 Verification

SMACK works on IR file generated from last step. It take as input the content of the IR and create another intermediate output called IVL (Intermediate Verification Language). This intermediate output contains statements for verification. SMACK passes this intermediate output to the verifier which finally verifies the validity of the statements and return the results. Verifier in this case is Corral. In case of errors, a trace is printed which can be used by the developer to track the location of error. Its always good to do the experiments with high loop unroll factor because it is likely that some bugs are not exposed with low loop unrolling.

3 Experiment Environment

Experiments have been performed using following setup:

Tool	Version	Specs
Clang/wllvm(Compiler wrapper-clang)	3.7.1	–
SMACK	1.7.2	–
Linux	3.16.0-55-generic	Ubuntu x86_64
Vagrant(Virtual Machine)	1.8.1	6GB RAM

4 Issues found

I performed my experiments specifically to find memory safety and integer overflow issues in Xen console tool source code.

4.1 Memory Safety

Memory issues can be found using following command:
"$smack\ --memory-safety\ \ IRfile\ \ --unroll\ N$".
It is likely that more bugs are reported with high loop unroll factor.

For Xenconsole daemon(xenconsoled), Corral verifier crashed with single loop unroll factor in 2 minutes. So, it couldn't be verified. From the error stack trace, I found that the error exists in Corral verifier rather than SMACK.

Errors detected in xenconsole(helper) are given below:

1. **Listing 1.1.** Illegal Memory access - Unallocated memory access

```
/*
 * Problem: Pointer accessed without allocating
 * memory to it
 */
int main(int argc, char *argv[]) {
        ...
        usage(argv[0]); //Memory not allocated for argv
}

/*
 * Solution: Declare a dummy pointer char **argv and
 * allocate memory to it for closing system
 */
int main(int argc, char *argv[]) {
        ...
        argv = malloc(3 * sizeof(char *));
        for (int i=0; i<3; i++)
                argv[i] = malloc(20*sizeof(char));
        ...
        usage(argv[0]);
        ...
        for (int m=0; m<3; m++)
                free(argv[m]);
        free(argv);
        ...
}
```

2. **Listing 1.2.** Illegal Memory access - Invalid array index access

```
/*
 * Problem: Invalid array index access
 */
int main(int argc, char *argv[]) {
        ...
        strtol(argv[x], &end, 10); //x = -1
}
/*
 * Solution : Provide valid array index checking
 */
```

Listing 1.3. Memory Leak

3.
```
/*
 * Problem:Program exiting without freeing resources
 */
if (ret < 0) {
        exit(ret); //Exiting without freeing memory
}

/*
 * Solution:Free memory allocated so far before exiting
 */
if (ret < 0) {
        free(argv);
        free(end);
        free(path);
        exit(ret);
}
```

4.2 Integer overflow

We can check integer overflow issues using following command:
"$smack \ \ -signed-integer-overflow \ \ file-name \ \ -unroll \ \ N$".
With small loop unroll factor, no issues were detected in xenconsole daemon and helper program. With high unroll factor, Corral verifier crashed for xenconsole daemon.
No bugs were detected in helper (xenconsole) source code.

4.3 Some Unexplored paths

There were some paths which SMACK can't explore. It simply retured error when such paths appeared. These are the paths which are calling third party library functions like strlen, strtol & select calls. In such cases, I assumed the external library calls to be correct and provided estimated values when required. Like, I replaced strlen(*ptr) call with some number after doing source code analysis.

5 Performance Results

In this section, I am showing the performance of SMACK in terms of time taken to verify the code. I have used "time" binary to measure the execution times.

1. Memory Safety Results

Binary	Loop Unroll factor	Time (in seconds)
xenconsole helper	4	3300
xenconsole daemon	4	–

Xenconsole helper source code could be verified in 55 minutes with loop unrolling of 4. Xenconsole daemon could not be verified because of Corral issue.

2. Integer Overflow Results

Binary	Loop Unroll factor	Time (in seconds)
xenconsole helper	4	7.47
xenconsole daemon	1	18.4

With loop unroll factor of 2 for Xenconsole daemon, Corral crashed.

6 Conclusion

Formal verification techniques can be helpful in finding more bugs because they are capable of exploring all possible branches in the source code. These kind of techniques can be employed to write safe code. Also, more code coverage can be achieved using formal verification methods. Advantage of SMACK is its ease of use and effectiveness in verifying the correctness of the C source codes. SMACK generates helpful stack dump traces for the issues reported in source code.

References

1. Carter, M., He, S., Whitaker, J., Rakamarić, Z., Emmi, M.: Smack software verification toolchain. In: Proceedings of the 38th International Conference on Software Engineering Companion. pp. 589–592. ICSE '16, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2889160.2889163
2. Chisnall, D.: The Definitive Guide to the Xen Hypervisor. Prentice Hall Press, Upper Saddle River, NJ, USA, first edn. (2007)
3. LABORATORY, S.A.R.: Soarlab (2017), http://soarlab.org/, [Online; accessed 11-May-2017]
4. Laguna, I., Schulz, M.: Pinpointing scale-dependent integer overflow bugs in large-scale parallel applications. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 19:1–19:12. SC '16, IEEE Press, Piscataway, NJ, USA (2016), http://dl.acm.org/citation.cfm?id=3014904.3014930
5. Lal, A., Qadeer, S.: Powering the static driver verifier using corral. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 202–212. FSE 2014, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2635868.2635894
6. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization. pp. 75–. CGO '04, IEEE Computer Society, Washington, DC, USA (2004), http://dl.acm.org/citation.cfm?id=977395.977673
7. Wikipedia: Hypervisor (2017), https://en.wikipedia.org/wiki/Hypervisor, [Online; accessed 11-May-2017]