Bachelor's Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering

Department of Cybernetics

The Use of Symbolic Execution for Testing of Real-Time Safety-Related Software

Martin Hořeňovský

Open Informatics — Computer and Information Science

horenmar@fel.cvut.cz

May 2015

Supervisor: Ing. Michal Sojka, Ph.D.

Acknowledgement

I would like to thank my supervisor Michal Sojka for his patience, guidance and advice. I would also like to thank my friends whom I bugged for proof-reading, even though they had a lot of their own work.

Declaration

I hereby declare that I made this on my own and I declared all used sources according to “Metodický pokyn o dodržování etických principů při přípravě vysokoškolských závěrečných prací”

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

....

Abstrakt / Abstract

Tato práce zkoumá vhodnost nástroje symbolické exekuce KLEE pro verifikaci real-time bezpečnostně kritických systémů. Real-time bezpečnostně kritické systémy jsou ty, u kterých selhání může vést ke ztrátě lidských životů anebo ke škodám na životním prostředí.

V této práci modifikuji dva real-time bezpečnostně kritické systémy, abych je mohl otestovat s KLEE a posoudit, jak komplikovaná jejich modifikace byla a zda má smysl začít používat KLEE k verifikaci real-time bezpečnostně kritických systémů. Došel jsem k závěru, že KLEE může být cenný nástroj pro verifikaci real-time bezpečnostně kritických systémů, ale daný systém musí být navržen s ohledem na KLEE.

Klíčová slova: Symbolická exekuce, KLEE, automatické testování, bezpečnostně kritický software, automobilový průmysl

Překlad titulu: Využití symbolické exekuce pro testování real-time bezpečnostně kritického softwaru

This thesis investigates fitness of symbolic execution tool KLEE for verification of real-time safety-critical systems. Real-time safety-critical systems are those systems, whose malfunction might result in loss of life and/or environmental damage.

In this thesis I modify two pieces of real-time safety-critical software to test them with KLEE and to evaluate how complex the modification was and whether using KLEE for verification of real-time safety-critical systems is viable going forward. I conclude that KLEE can be a valuable tool for verifying real-time safety-critical software, but the software has to be designed with KLEE in mind.

Keywords: Symbolic execution, KLEE, automatic testing, security-critical systems, automotive domain

/ Contents

Tables / Figures

Chapter 1
Introduction

Safety-related software is software whose failure might result in death, injuries and substantial property damage. Every day more and more people rely on such software. It is characterized by being thoroughly tested and reliable, but as the Internet of Things is becoming pervasive, safety cannot be achieved without considering security, as was shown by the recent example of being able to locate and unlock Tesla Model S electric car by hacking a trivially brute-forceable security code [1].

Because security used to be thought of as completely orthogonal to safety-related software, traditionally used tools for verification of safety-related software are ill-prepared to expose security vulnerabilities. Therefore we decided to try a verification method oriented towards finding bugs in general and evaluate its fitness for testing safety-related software in particular.

The verification method we decided to use is symbolic execution, using KLEE [2] as our executor. Symbolic execution seemed as a good fit for our needs as it is generally sound and its main drawback, exponential runtime complexity in number of executed branches, is mitigated by the fact that safety-related software contains a relatively low number of paths through its code. And while no tool can prevent security vulnerabilities caused by naive authentication scheme, automatic testing can find security vulnerabilities caused by *implementation* bugs which can allow an attacker to bypass security measures that are in place.

This thesis investigates KLEE's fitness for purpose of testing safety-related software, documents its results on two selected pieces of safety-related software, eMotor and MaCAN library, and evaluates whether symbolic execution tools should be used for testing safety-related software in the future.

The rest of this work is organized as follows. Second chapter, Background and Related Technologies, introduces basic information about undefined behaviour, safety-related software, symbolic execution and existing tools using it, the tool I am using in this thesis and the two pieces of safety-related software we will use to evaluate KLEE's fitness for purpose. Third chapter, Toolchain and case study preparation, covers work done on KLEE, eMotor software and the MaCAN library in the course of writing my thesis. Fourth chapter, Evaluation, covers our findings, such as the number of bugs found using KLEE, their severity, time spent on finding them and whether KLEE will be a useful tool for testing real-time safety-related software going forward. The final chapter, Conclusion, recapitulates our findings and recommendations.

Chapter 2
Background and related technologies

Standard C [3] places no constraints upon the result of program invoking undefined behaviour (UB), and this is often interpreted by compilers as allowance to generate arbitrary output for such program. It also defines many causes of undefined behaviour, such as[1]

- Dereferencing uninitialized pointer
- Dereferencing a NULL pointer
- Dereferencing pointer to no longer valid object (i.e. freed object)
- Creating pointers outside of allocated memory
- Converting pointers to objects of incompatible types (e.g. converting `float*` to `int*`)
- Signed integer overflow (unsigned overflow is well defined)
- Shifting values by more than its size (i.e. `int64_t i = 1 << 70`)
- Evaluating arithmetic expression that would be mathematically undefined (e.g. division by zero)

Modern compilers use undefined behaviour to perform optimizations and code in Figure 2.1 will be transformed into code in Figure 2.2 by such compiler. The reasoning is that since a program cannot contain undefined behaviour, and the function dereferences `ptr`, it cannot be null or the function would invoke undefined behaviour. This allows the compiler to remove the conditional branch and error checking, which is seen as worthwhile optimization.

```
int foo(int* ptr){
    int temp = *ptr;
    if (!ptr){
        return ERR;
    }
    ...
}
```

Figure 2.1. Example of undefined behaviour invoking code.

```
int foo(int* ptr){
    int temp = *ptr;
    ...
}
```

Figure 2.2. Code after compiler transformation.

[1] Taken from ISO C99 Appendix J.2 [3]

The willingness of compilers to exploit undefined behaviour in code has already been documented as causing security bugs in real world [4], but defined behaviour can lead to bugs as well. A common example is unsigned overflow, which is well defined, but often is not accounted for and breaks program's logic.

While static analysis can find some of these potential defects (e.g. dereferencing uninitialized pointers and incompatible pointer conversions), it is not sound and gives both false positives and false negatives. Static analysis is also unable to find integer overflows, as these are inherently dynamic and the only viable way to detect them is with tools performing runtime checking.

2.1 Safety-critical software and security

Safety-critical systems are those systems whose malfunction and/or failure might result in death (or grievous injury) of people, severe damage to and/or loss of equipment and environmental damage. These systems are increasingly often implemented in software, as opposed to hardware, or as mechanic, pneumatic, hydraulic, ..., systems and new methods of software defect prevention are needed.

There are various ways to decrease the number and frequency of software malfunctions, such as specific development methodologies (e.g. MISRA C[1], and JSF-AV C++ coding standards[2]) which aim to provide safer subset of the language, thus decreasing possible space for defects and set requirements for testing rigour. Similarly, since compilation is another potential source of software defects, there are formally provable compilers [5].

In general, safety-critical systems often have hard real-time requirements and are required to have their responses fully defined for all possible inputs. This means that software contained within must always respond under a fixed time-limit and have to fully define its failure modes, even if the response is just turning off motor and engaging emergency breaks.

2.2 Symbolic execution

The term symbolic execution has been coined in year 1976 by James C. King in article Symbolic execution and program testing [6]. Since then symbolic execution has been implemented by various tools for various languages, from x86 assembly [7], LLVM's intermediate representation (IR) (from C/C++) [2, 8] to higher level languages such as the .NET framework [9].

The contents of rest of this section are heavily based on the KLEE paper [2] and the S2E paper [7]. Symbolic execution is a testing technique where program is run through an interpreter, which allows for inputs to be symbolic, as opposed to concrete. While run under interpreter, symbolic data (either input or variables) are not actual data, but rather a set of boolean formulae that have been placed upon them by conditions along the currently executing path. This allows the interpreter to go through all possible paths in a program, without having to generate all possible inputs. This is achieved by keeping track of all conditions leading to any given path and using a SAT solver to

[1] `http://www.misra.org.uk/MISRAHome/WhatisMISRA/tabid/66/Default.aspx`
[2] `http://www.stroustrup.com/JSF-AV-rules.pdf`

```
int is_odd(int n){
    if (n % 2) {
        return 1;
    } else {
        return 0;
    }
}
```

Figure 2.3. Example function for symbolic execution.

A) determine whether any given path can be taken and B) generate a sample solution (concretization of symbolic data) that will lead to any given path. As an example, consider the function in Figure 2.3.

It has 2^{32} possible concrete inputs, but as can be easily seen, has only two paths through. Symbolic testing allows us to find both paths without having to explore all 2^{32} inputs. If we mark the input as symbolic and use KLEE to test this function, it outputs both paths and gives concretized values that give full coverage as can be seen in Figure 2.4.

```
KLEE: done: total instructions = 29
KLEE: done: completed paths = 2
KLEE: done: generated tests = 2
...
args       : ['is_odd.o']
num objects: 1
object    0: name: 'n'
object    0: size: 4
object    0: data: 0
...
args       : ['is_odd.o']
num objects: 1
object    0: name: 'n'
object    0: size: 4
object    0: data: 1
```

Figure 2.4. Abbreviated KLEE's output for the example.

Unlike with fuzzing, time required for symbolic execution of any given program doesn't increase exponentially with size of input, but rather increases with amount of branches (possible paths) through the program. This allows use of symbolic testing on much larger programs than just using random inputs.

On the other hand, certain kinds of conditional branching are more or less unsolvable. Consider comparing cryptographic hash of a symbolic value against predetermined result. To be able to give proceed, a symbolic tool would have to reverse a cryptographic hash which is generally regarded as impossible without brute-force testing the input space, and in specific cases might be impossible completely. Other problematic constructs are unbounded loops (i.e. those waiting for a hardware response) and infinite loops, which prevent symbolic execution from terminating.

Various solutions to increase path coverage exist, including parallelizing independent paths [8], usage of heuristics to guide search [2, 8, 10, 11], usage of annotations [10] and selective symbolic execution [7].

2.2.1 Existing tools

There are other tools for performing symbolic execution, many of which are based on KLEE [7, 8, 10, 11], but we picked KLEE because it is a general purpose tool and would be easiest to extend for our needs in the future.

Cloud9[1] is a symbolic executor that can scale over clusters of machines, and a cloud service for testing software [8]. It's innovation is dynamic partitioning of search space while having a shared-nothing architecture. It is also capable of executing C++ code.

S2E[2] is a platform for analysing large scale programs *in vivo*. It accomplishes this by using *selective symbolic execution* a technique that automatically minimalises amount of symbolically executed code within a binary. Together with relaxed execution consistency, this allows S2E to analyse code as run in its real environment [7].

SymDrive[3] is a tool for testing Linux and FreeBSD drivers without hardware [10]. It uses static-analysis to automatically find driver entry points and loops, together with source to source transformation, which often allows it to test a driver without requiring any modifications by its author and if it cannot create modified driver automatically (that is, it cannot perform necessary transformations for testing), it alerts the developer as to what changes are required [10].

Kite[4] is a KLEE based tool that prunes its search space whenever a path is proven to be infeasible. It is based on recent improvements of Conflict-Driven Clause Learning (CDCL) Boolean Satisfiability Problem (SAT) solvers and uses markedly different exploration strategy from other tools, in that it looks for paths which allow it to "learn" the most, that is, prune the search space the most [11].

2.3 KLEE

KLEE is a symbolic execution tool primarily geared towards performing high-coverage tests on programs, originally created by Cadar et al. [2]. It is built on the LLVM compiler infrastructure, using its Clang front-end to convert C code to LLVM intermediate representation (IR), which it then works on.

Officially KLEE still has LLVM-2.9 and LLVM-GCC as its dependencies, which are currently almost 4 years outdated. Since then there was significant amount of work done to allow KLEE to work with newer version of the LLVM toolchain and Clang instead of LLVM-GCC, but the only officially supported toolchain is LLVM-2.9 and LLVM-GCC. The officially supported version of Simple Theorem Prover (STP) SAT solver is *extremely* outdated as well. Later chapter covers what did I have to do to be able to use it for testing.

KLEE can find and show paths that lead to any of

- Assertion violation
- Access outside of allocated memory (including null pointer dereference and double free)

[1] `http://cloud9.epfl.ch/`
[2] `http://s2e.epfl.ch/`
[3] `http://research.cs.wisc.edu/sonar/projects/symdrive/`
[4] `http://www.cs.ubc.ca/labs/isd/Projects/Kite/`

- Division by zero
- Integer overshift (shifting an integer by more than its size)
- Call to `abort()`

and recently was also extended to handle integer overflow checking, as detailed in later section.

2.3.1 Principle of operation

Text in this subsection is based on the KLEE paper [2].

KLEE takes a file of LLVM IR as its input and works directly on the LLVM IR. LLVM IR is a low-level programming language resembling strongly typed assembly for a very abstracted machine. It uses infinite set of registers, supports floating point, variable width integers, exception handling, atomicity and threads, but also explicit calling conventions. There are two (three) standard forms of LLVM IR, human-readable assembly, serialized bitcode (and C++ objects). KLEE is capable of interpreting most of LLVM IR, but it does not support symbolic floating point, threads, `longjmp`, embedded assembly code and some intrinsics. All memory allocations also have to have concrete size.

At the core of KLEE testing is an interpreter loop, selecting a currently open state. KLEE's state consists of the current values of registers, stack and heap objects as represented by expression trees built from LLVM IR operations. The interpreter then executes current LLVM IR instruction according to the current state. If the instruction does not branch, its execution is straightforward, but if the executed instruction is a conditional branch instruction, KLEE uses a SAT solver to determine whether the condition is provably true or false given current state and constraints created along the path taken to reach it, in which case KLEE takes the appropriate path. If the conditional branch cannot be proved to be either, KLEE creates two new states with added constraints from the branch. These two states are then added to priority queue of all open states.

Branches can also be created implicitly, by potentially dangerous operations. Every division, pointer dereference and similar instructions create an implicit branch that checks for possibility of error. For example, division instruction creates a branch checking whether the divisor can be equal to zero, and if it can, it terminates the state and generates test case leading to to it.

This interpreter loop and state expansion continue while there are open states in its priority queue or while a user defined timeout has not expired.

2.3.2 KLEE integer overflow checking

When I started working with KLEE, it was unable to detect integer overflow. We decided to add support for checking integer overflow, but at the same time Dariz Luca had implemented it as well and since his work has been mainlined [12], I will talk about his implementation.

This capability relies on Clang's support for Undefined Behavior Sanitizer (also known as "ubsan")[1], which in turn is based on work by Regehr et al. on integer overflow checking [13]. KLEE implements overflow checking by performing arithmetic operations

[1] `http://blog.regehr.org/archives/905`

as-if done on double sized types and then checking upper half of the result. If it is not empty, the operation must have overflown.

KLEE can check for overflow of both signed and unsigned integral types, but the checks have to be turned on and/or off during tested program's compilation. This is caused by KLEE's reliance on Clang's intrinsics.

2.3.3 Example of KLEE usage

This subsection explains basic usage of KLEE. We will go through running KLEE on two toy single file programs, `sum.c` and `memcpyassert.c`, whose listings are in Figure 2.5 and Figure 2.6 respectively. We will use Clang-3.3 as our LLVM IR compiler.

```
#include "klee/klee.h"

int read_int(){
    int res;
    klee_make_symbolic(&res, sizeof(res), "summand");
    return res;
}

int main(){
    int total = 0;
    for (int i = 0; i < 5; ++i){
        total += read_int();
    }
    return 0;
}
```

Figure 2.5. `sum.c` listing

```
#include "klee/klee.h"
#include <string.h>
#include <assert.h>

int read_int(){
    int res;
    klee_make_symbolic(&res, sizeof(res), "some int");
    return res;
}

int main(){
        int a, b;
        a = read_int();

        memcpy(&b, &a, sizeof(b));

        assert(b != 0);

        return 0;
}
```

Figure 2.6. `memcpyassert.c` listing

```
clang -emit-llvm -c -g -fsanitize=integer <name>.c
```

Figure 2.7. Compilation command line

```
klee <name>.o
```

Figure 2.8. How to run KLEE

Because KLEE runs on LLVM bitcode files, the first step is to run Clang on the source file as seen in Figure 2.7. The result of this step is a `.o` file containing LLVM bitcode, which can be now run by KLEE as seen in Figure 2.8.

The output of running KLEE on `sum.o` can be seen in Figure 2.9 and in Figure 2.10 for `memcpyassert.o`. As they show, both of these toy programs contain possible bugs. For `sum.c` it is a possible signed[1] integer overflow, which is undefined behaviour and `memcpyassert.o` contains possible assertion violation.

```
KLEE: output directory is "/.../thesis-setup-stable/examples/klee-out-0"
KLEE: ERROR: /.../thesis-setup-stable/examples/sum.c:12: overflow on
unsigned addition
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 218
KLEE: done: completed paths = 5
KLEE: done: generated tests = 2
```

Figure 2.9. KLEE results for `sum.c`

```
KLEE: output directory is "/.../thesis-setup-stable/examples/klee-out-1"
KLEE: ERROR: /.../thesis-setup-stable/examples/memcpyassert.c:12:
ASSERTION FAIL: b != 0
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 62
KLEE: done: completed paths = 2
KLEE: done: generated tests = 2
```

Figure 2.10. KLEE results for `memcpyassert.c`

For every run KLEE creates a new folder with several files, as can be seen in Figure 2.11. In addition to files created for the execution itself, `assembly.ll`, `info`, `messages.txt`, `run.stats`, `run.istats` and `warnings.txt`, KLEE has created sample inputs for every distinct path through the sample programs and for every error found (`test<n>.ktest`) and if the path generated an error, two more files: `test<n>.pc` with condition guarding the error and `test<n>.<err>` with details about the error (what kind, where in the source code, call stack).

The `test<n>.ktest` files are in a binary format, so KLEE comes with an utility to read them, `ktest-tool`. If used it writes out human readable list of symbolic variables and

[1] Yes, signed. KLEE's overflow reporting currently contains a bug and reports every overflow as unsigned.

```
assembly.ll   run.istats         test000001.overflow.err  warnings.txt
info          run.stats          test000001.pc
messages.txt  test000001.ktest   test000002.ktest
```

Figure 2.11. list of files in KLEE's results directory

```
/.../klee-last$ ktest-tool --write-ints test000001.ktest
ktest file : 'test000001.ktest'
args       : ['sum.o']
num objects: 2
object    0: name: 'summand'
object    0: size: 4
object    0: data: 1073741824
object    1: name: 'summand'
object    1: size: 4
object    1: data: 1073741824
```

Figure 2.12. Sample listing of ktest-tool output

```
/.../examples$ klee-stats klee-last
------------------------------------------------------------------------
|  Path    |  Instrs|  Time(s)|  ICov(%)|  BCov(%)|  ICount|  TSolver(%)|
------------------------------------------------------------------------
|klee-last|     218|     0.23|    29.59|    12.50|     169|       97.19|
------------------------------------------------------------------------
```

Figure 2.13. Sample listing of klee-stats output.

their concrete values that would lead program execution through to its corresponding path. For an example, see Figure 2.12.

KLEE also writes out runtime statistics, including branch coverage, LLVM IR instruction coverage, total time spent running, and so on. Figure 2.13 shows an example of usage and output.

2.4 Real-Time safety-critical applications

This section covers the two pieces of real-time safety-critical software we decided to use to evaluate KLEE's fitness for testing real-time safety-critical software. We chose eMotor because at the time we had a joint project with Infineon and MaCAN library because the Industrial Informatics Group created it.

2.4.1 eMotor

The eMotor driver is Infineon Technologies' proprietary electric motor control software module for AUTomotive Open System ARchitecture (AUTOSAR[1]), standardized software architecture for automotive domain. It has an interrupt driven design (the entire control algorithm is executed inside the interrupt handler) and is designed to run on 32-bit Tricore TC1798 microcontroller. It requires proprietary IDE, compiler, assembler and linker to be compiled.

[1] http://www.autosar.org/

2.4.2 MaCAN

Controller Area Network (CAN) bus is standard communication bus for vehicles, designed by Bosch [14]. The CAN protocol is message based and there are two different message formats, standard frame format and extended frame format. Because it was usually assumed that attackers do not have access to the network inside vehicle, CAN was designed for deterministic real-time communication, reliability and robustness and no consideration was given to security. However modern vehicles often have at least one remotely accessible interfaces and thus there is now a need to have message security as part of the communication protocol.

The MaCAN protocol contains both a key exchange protocol and a means of message authentication, giving a measure of message security and has an advantage in that it builds upon the CAN bus and is backwards compatible with already existing deployments of CAN. Because of these backward compatibility constraints, such as only 8 byte payload in a single CAN frame, MaCAN protocol cannot use more than 32 bits for its Message Authentication codes and thus the cryptographic security is weaker than would be recommended in other domains. However, this is mitigated by other factors, such as short lived keys and the relative slowness of CAN network limiting rate at which an attacker can make guesses. [15]

The MaCAN library[1] implements the MaCAN protocol for Linux, Infineon Tricore TC1798 and STM32 architectures [16]. It was designed to be cross-platform by only having single, platform-independent dependency, cryptographic library Nettle[2] and by separating platform dependent code from bulk of the library.

My investigation was done on a snapshot of the MaCAN library taken at commit 1dac9fed8b9777d32e0c28b2b826b8ac36f146a5[3] with added modification to allow testing with KLEE. I will talk about these in the following chapter.

[1] `https://github.com/CTU-IIG/macan`
[2] `http://www.lysator.liu.se/~nisse/nettle/`
[3] `https://github.com/horenmar/macan/commit/1dac9fed8b9777d32e0c28b2b826b8ac36f146a5`

Chapter 3
Toolchain and case study preparation

This chapter covers changes done to KLEE, eMotor and MaCAN to allow testing these two libraries using KLEE. It will also cover problems I ran into along the way, touching upon some "personal" experience gained during this thesis.

Virtually all my work is also available online, in a GitHub repository[1].

3.1 KLEE preparation

Because KLEE's overflow checking relies on Clang's Undefined Behavior Sanitizer, I had to find a way of compiling and linking together binary from multiple source files using Clang. Because the bitcode linking facilities in LLVM have been deleted between the 2.9 (officially supported by KLEE) version and 3.3 (first version with ubsan support) version, I used the Whole program LLVM[2] (WLLVM) utility created by Tristan Ravitch.

WLLVM works by using a compiler capable of emitting both normal object files and the LLVM bitcode, generating both and saving bitcode into normal object file. During linking the bitcode sections are also linked together and the final bitcode program can then be extracted from resulting binary [17] and run in KLEE.

Furthermore, to enable integer overflow checking I had to change how WLLVM calls Clang, which means that whether WLLVM compiles code with overflow checking enabled or disabled is hardcoded into the utility itself. The patch itself is from the KLEE mailing list [18].

3.2 eMotor modifications

Because eMotor's hardware-dependent code is weakly abstracted, modifying eMotor to support symbolic execution has shown itself to be hard and very time consuming. Because of time issues and the fact that the project for which we worked on eMotor has ended, I talked to my supervisor and we decided to give up on testing eMotor.

3.3 MaCAN modifications

As mentioned before, MaCAN's architecture has decoupled architecture specific code from the general library code. This allowed me to initially make no changes to the library part of MaCAN code, shimming hardware dependencies to return symbolic values when run under KLEE.

[1] `https://github.com/horenmar/thesis-setup-stable`
[2] `https://github.com/travitch/whole-program-llvm`

However, the MaCAN library inner working is deeply reliant on cryptography and validating cryptography using symbolic execution is intractable. Because of this I ended up having to bypass parts of the code, compromising completeness of testing coverage to be able to analyse the rest of MaCAN library codebase.

More specifically, in addition to adding KLEE as a platform to the MaCAN project and adding the necessary platform-specific scaffolding, I only had to perform small modifications to the original library source in the form of conditional compilation and create a dummy MaCAN node application, to create a valid execution entry point for KLEE.

```
 build/klee/Makefile.omk         |   1 +
 build/klee/clear_early.sh       |  12 +
 build/klee/config.target        |  11 +
 build/klee/macan                |   1 +
 build/klee/nettle               |   1 +
 build/klee/node/Makefile.omk    |   4 +
 build/klee/node/node.c          | 138 ++++
 build/klee/run_klee.sh          |   5 +
 macan/include/Makefile.omk      |   3 +
 macan/include/klee.h            |   7 +
 macan/src/Makefile.omk          |   1 +
 macan/src/cryptlib.c            |   9 +
 macan/src/klee/klee_cryptlib.c |  39 ++
 macan/src/klee/klee_macan.c     |  41 ++
 macan/src/klee/macan_ev.c       | 102 +++
 macan/src/klee/macan_ev.h       |  82 +++
 macan/src/macan.c               |   6 +-
```

Figure 3.1. Statistics of my work on MaCAN library.

Figure 3.1 shows overall statistics of changes during my work in the MaCAN repository, minus boilerplate files of MaCAN's build system. Figure 3.3 shows dummy event loop used to invoke the MaCAN library, Figure 3.6 shows my implementation of MaCAN's platform hardware specific functionality and Figure 3.4 shows my implementation of MaCAN's platform specific cryptographic functionality.

```
void
can_rx_cb(macan_ev_loop *loop, macan_ev_can *w, int revents){
    (void)loop; (void)revents; /* suppress warnings */
    struct macan_ctx *ctx = w->data;
    struct can_frame cf;

    while (macan_read(ctx, &cf))
        macan_process_frame(ctx, &cf);
}
```

Figure 3.2. Listing of CAN receive callback.

As can be seen in Figure 3.3, the testing harness consists of a loop called three times, where it each time goes through all registered timers (which for the test program means calling the "housekeeping" timer, that is responsible for renewing expired keys) and then calls the MaCAN library entry point for received messages, `can_rx_cb`. As seen in Figure 3.2, this callback loops while there are current messages and as Figure 3.6 shows, 10 messages can be read in a row. This means that every path through the program found by KLEE has read 30 messages.

Figure 3.5 shows patch that was needed in the general MaCAN implementation of `cryptlib.c` (I also removed static linkage from function serving as entry point for CAN frames in `macan.c`).

All my modifications are publicly available as a fork of the original project[1] as well as on the DVD accompanying this thesis (see Appendix C for more detail).

```
bool
macan_ev_run(macan_ev_loop *loop){
        for (int i = 0; i < 3; ++i){
                uint64_t now = read_time();

                for (macan_ev_timer *t = loop->timers; t; t = t->next) {
                        if (now >= t->expire_us) {
                                t->cb(loop, t, MACAN_EV_TIMER);
                                t->expire_us = now + t->repeat_us;
                        }
                }

                can_rx_cb(NULL /*ignored*/, loop->cans, 0 /*ignored*/);

        }
        return true;
}
```

Figure 3.3. Listing of KLEE's event loop for testing.

```
void macan_aes_cmac(const struct macan_key *key, size_t length,
                    uint8_t *dst, uint8_t *src){
    (void)key, (void)length, (void)dst, (void)src;
    memset(dst, 0, 16);
}

void macan_aes_encrypt(const struct macan_key *key, size_t len,
                       uint8_t *dst, const uint8_t *src){
    (void)key, (void)src;
    klee_make_symbolic(dst, len, "aes encryption");
}

void macan_aes_decrypt(const struct macan_key *key, size_t len,
                       uint8_t *dst, const uint8_t *src){
    (void)key, (void)src;
    klee_make_symbolic(dst, len, "aes decryption");
}
```

Figure 3.4. Overview of MaCAN's KLEE target implementation. `cryptlib.c`

[1] `https://github.com/horenmar/macan/tree/klee`

```
 void macan_unwrap_key(const struct macan_key *key, size_t srclen,
                       uint8_t *dst, uint8_t *src) {
+#ifdef WITH_KLEE
+       klee_make_symbolic(dst+16, 7, "aes unwrap modification");
+#else
        macan_aes_unwrap(key, srclen, dst, src, src);
+#endif
 }
```

Figure 3.5. Necessary patch against `cryptlib.c`.

```
uint64_t read_time(void){
       uint64_t time;
       klee_make_symbolic(&time, sizeof(time), "time");
       return time;
}

bool gen_rand_data(void* dest, size_t len){
       memset(dest, 0, len);
       return true;
}

bool macan_read(struct macan_ctx* ctx, struct can_frame* cf){
       (void)ctx;
       klee_make_symbolic(cf, sizeof(struct can_frame),
                          "incoming can frame");
       static int counter = 0;
       counter++;
       counter %= 10;
       return counter != 0;
}

void macan_target_init(struct macan_ctx* ctx){
       (void)ctx;
}

//Not currently part of testing.
bool macan_send(struct macan_ctx* ctx, const struct can_frame* cf){
       (void)ctx, (void)cf;
       return true;
}
```

Figure 3.6. Overview of MaCAN's KLEE target implementation. `macan.c`

Figure 3.7 shows messages sent in the MaCAN network when it first establishes itself. Because CAN messages received by the node are symbolic in our testing, we are only interested in messages the node receives (they are marked with >> in the figure). There are 16 messages that we need to receive to simulate a node startup, and all messages afterwards are a payload.

As I already mentioned, my test harness forces the MaCAN library to receive a total of 30 messages, meaning that our node receives enough messages to perform initial initialization and then 14 messages extra. This means that our testing harness could exercise all paths through the MaCAN library, except those that make use of cryptographic primitives or those that rely on specific node configuration (as the node configuration for our test harness was concrete.)

```
  0.007 40029DCA42EE41EA crypt TS->KS (1->0): challenge fwd_id=S
  0.007 4003653631FB679F crypt TS->KS (1->0): challenge fwd_id=R
  0.007 00000000         time 0
  0.007 4001FB8D4CB204DD crypt S->KS (2->0): challenge fwd_id=TS
  0.007 4003CBFC5E431EEB crypt S->KS (2->0): challenge fwd_id=R
  0.007 4001ED728E635600 crypt R->KS (3->0): challenge fwd_id=TS
  0.007 4002A90562E93EA7 crypt R->KS (3->0): challenge fwd_id=S
  0.007 81068C7BB634DCB8 crypt KS->TS (0->1): sess_key seq=0 len=6
  0.007 8116527F367675FE crypt KS->TS (0->1): sess_key seq=1 len=6
  0.007 8126F2E868371D44 crypt KS->TS (0->1): sess_key seq=2 len=6
  0.007 81365DA64E2999FE crypt KS->TS (0->1): sess_key seq=3 len=6
  0.007 814680FEB6BA8926 crypt KS->TS (0->1): sess_key seq=4 len=6
  0.007 815236F1B6BA8926 crypt KS->TS (0->1): sess_key seq=5 len=2
  0.007 0201             crypt KS->S (0->2): req challenge fwd_id=TS
  0.008 810633B3D3CDFC50 crypt KS->TS (0->1): sess_key seq=0 len=6
  0.008 8116C2AD9E7A9B20 crypt KS->TS (0->1): sess_key seq=1 len=6
  0.008 81262D798AF157C3 crypt KS->TS (0->1): sess_key seq=2 len=6
  0.008 8136EAD1B8033C06 crypt KS->TS (0->1): sess_key seq=3 len=6
  0.008 8146E89D3BC0985A crypt KS->TS (0->1): sess_key seq=4 len=6
  0.008 815229A33BC0985A crypt KS->TS (0->1): sess_key seq=5 len=2
>>0.008 0301             crypt KS->R (0->3): req challenge fwd_id=TS
  0.009 4100F14494135731 crypt S->TS (2->1): challenge fwd_id=KS
>>0.010 0302             crypt KS->R (0->3): req challenge fwd_id=S
  0.010 0000000077F739C5 authenticated time 0
  0.010 81040000BD50C3F9 crypt S->TS (2->1): ack group=[2]
>>0.010 83040000252D298E crypt S->R (2->3): ack group=[2]
>>0.011 83064C12C76D9612 crypt KS->R (0->3): sess_key seq=0 len=6
>>0.011 831695FAED34F1B6 crypt KS->R (0->3): sess_key seq=1 len=6
>>0.011 83268DEFEE67C20E crypt KS->R (0->3): sess_key seq=2 len=6
>>0.011 833628066B8A5CB9 crypt KS->R (0->3): sess_key seq=3 len=6
>>0.011 83463C529952A7A9 crypt KS->R (0->3): sess_key seq=4 len=6
>>0.011 8352ABB29952A7A9 crypt KS->R (0->3): sess_key seq=5 len=2
  0.011 41000CC006A7A332 crypt R->TS (3->1): challenge fwd_id=KS
>>0.011 8306F18851EF18A6 crypt KS->R (0->3): sess_key seq=0 len=6
>>0.011 8316853B6B3D173A crypt KS->R (0->3): sess_key seq=1 len=6
>>0.011 8326B5867DB711E5 crypt KS->R (0->3): sess_key seq=2 len=6
>>0.011 833640EE35D796E6 crypt KS->R (0->3): sess_key seq=3 len=6
>>0.011 834652C89235C20A crypt KS->R (0->3): sess_key seq=4 len=6
>>0.011 83520BEC9235C20A crypt KS->R (0->3): sess_key seq=5 len=2
  0.011 0000000054D0F079 authenticated time 0
  0.011 81080000EFC1FEAA crypt R->TS (3->1): ack group=[3]
  0.011 820800008987F55D crypt R->S (3->2): ack group=[3]
>>0.011 830C00005F99C3D4 crypt S->R (2->3): ack group=[2 3]
  0.011 C20000FE2B6430   crypt R->S (3->2): auth req + MAC signal=#0
presc=0
```

Figure 3.7. List of initialization messages on MaCAN network.

Chapter 4
Evaluation

This chapter presents results I obtained by running KLEE on the MaCAN library to evaluate its viability as testing tool of real-time safety-critical software. Because of the already mentioned difficulties with modifying eMotor to run under KLEE, results from testing eMotor are not included.

4.1 Findings

When I ran KLEE on the modified MaCAN library, KLEE has reported some potential errors. After fixing Assertion and access violation caused by my misconfiguration of the test client, I ran KLEE again in two configurations. One using the default heuristics as its search strategy and one using DFS as its search strategy. Total numbers of bugs found during both runs are in Table 4.1.

Type of bug	Count
Assertion violation	0
Access outside of allocated memory	0
Division by zero	0
Integer overshift	0
Calls to abort	0
Integer overfow	8

Table 4.1. Overview of bugs found.

As can be seen, the only reports were for integer overflows. The numbers for these are further broken down in Table 4.2. All of them were found in arithmetic expressions containing time. To determine whether the potential overflow in any given report is dangerous or not, I used simple rules: any and all signed overflows invoke UB and as such are always considered dangerous. For unsigned overflows, I used personal inspection and assistance from my supervisor, Michal Sojka[1].

Type of overflow	Count
benign	5
causing bugs	2
causing undefined behaviour	1

Table 4.2. Overview of overflows found.

[1] Michal Sojka is one of the MaCAN library authors and current maintainer.

```
uint64_t time;
int i;
...
time = macan_get_time(ctx);

for (i = -1; i <= 1; i++) {
    *ftime = htole32((int)time + i);
    macan_aes_cmac(skey, len, cmac, plain);

    if (memcmp(cmac4, cmac, 4) == 0) {
        return 1;
    }
}
```

Figure 4.1. Listing of the UB causing bug.

```
int delta_t;
uint32_t time = (uint32_t)macan_get_time(ctx);
for (delta_t = -1; delta_t <= 1; delta_t++) {
    *ftime = htole32(time + (uint32_t)delta_t);
}
```

Figure 4.2. Listing of the fix for UB causing bug in mainline MaCAN.

The one case of undefined-behaviour-causing overflow was in function checking authenticity of received message. The programmer wanted to test whether the message was authentic, assuming it was sent either slightly before "now" (that is, current MaCAN time), now, or after now and wrote the code in Figure 4.1, where the expression `(int)time + i` performed signed arithmetic that could easily overflow.

After I found this bug, it was fixed in mainline MaCAN library, by changing the code to the one in Figure 4.2 [19]. The new version can still overflow (in fact it has to overflow to work properly), but the new overflow is well defined from the language point of view and is intended (and thus benign).

One of the two non-benign overflows was found in function that receives time from the MaCAN network and attempts to adjust local time to keep it in sync. As can be seen in Figure 4.3 it calculates time in microseconds and places the result into 64 bit unsigned integer which is large enough, but the arithmetic itself is done with 32 bits of precision and leads to erroneous results.

```
uint32_t time_ts;
uint64_t time_ts_us;
...
time_ts_us = time_ts * ctx->config->time_div;
```

Figure 4.3. Unsigned overflow causing bug.

This bug was also found independently by Michal Sojka when he was presenting MaCAN at an exhibition and Figure 4.4 shows the fix already implemented in mainline. Casting `time_ts`'s type to 64 bit unsigned integer leads to the multiplication being performed with 64 bits of precision, which means that overflow can still potentially happen, but only if the MaCAN network in question is online for more than 500000 years.

```
uint32_t time_ts;
uint64_t time_ts_us;
...
time_ts_us = (uint64_t)time_ts * ctx->config->time_div;
```

Figure 4.4. The fix for unsigned overflow bug.

The other non-benign overflow is in the event-loop mechanism, in timer expiry tracking. It is currently dormant because MaCAN system time, stored in 64 bit unsigned integer `now`, is platform specific and could theoretically overflow. All current platform implementation start system time at 0, but if a future platform started system time at different value, it could trigger the overflow in Figure 4.5 and the overflow could cause errors in the future, unless fixed. The bug lies in the expression `t->expire_us = now + t->repeat_us` whose right-hand side can overflow and the timer then will remain expired until MaCAN system time (`now`) overflows as well.

```
for (macan_ev_timer *t = loop->timers; t; t = t->next) {
    if (now >= t->expire_us) {
        t->cb(loop, t, MACAN_EV_TIMER);
        t->expire_us = now + t->repeat_us;
    }
}
```

Figure 4.5. Dormant bug in the event loop.

The rest of the potential overflows KLEE found were classified as benign. This either meant that they only occurred within 64 bit arithmetic and thus could happen only theoretically, or they were intended. Figure 4.6 contains the flagged expressions.

```
return (read_time() + (uint64_t)ctx->time.offs) / ctx->config->time_div;

t->offs = (time_ts_us - read_time());

t->offs = (time_ts_us - t->nonauth_loc);

loc_us = now + t->offs;

w->expire_us = read_time() + w->after_us;
```

Figure 4.6. Overview of benign overflows.

Table 4.3 shows an overview of bugs found, where were they found and their classification. I added a "dormant" classification to label overflow that currently doesn't cause bugs, but under different hardware could.

4.2 Complexity, limitations, execution time

As trying to test eMotor has shown, modifying software so it can be tested can be difficult if its hardware-dependent code is weakly abstracted. This limits the kinds of

File	Line	Classification
cryptlib.c	165	causing undefined behaviour
macan.c	409	benign
macan.c	468	causing bugs
macan.c	473	benign
macan.c	475	benign
macan.c	864	benign
macan_ev.c	68	benign
macan_ev.c	94	dormant

Table 4.3. Overview of found overflows.

software that can be tested using KLEE, to those with well abstracted inputs, which usually means software that was written to be cross-platform.

Another difficulty in using KLEE to test software is that the tested software requires modifications by programmer, which means that the validity of results depends on programmer. This has shown itself when I created a test program for the MaCAN library. Due to an error, made while configuring the library in my test program, I ended up with a false positive results and had to determine it was configuration error via manual inspection.

Apart from relying on programmer to prepare the test program correctly, another potential pitfall of using KLEE to test software is the time it needs to test a program. The MaCAN test setup described previously has been running on modestly fast CPU[1] for 5 days without finishing. This is caused by KLEE's complexity being exponential in both memory (because the number of possible path generally increases exponentially with encountered conditional branches) and in CPU time (because it attempts to solve SAT problem at every branching).

The memory complexity can be mitigated at the cost of further increasing runtime, by running KLEE with search strategy set to either Depth First Search (DFS) or Iteratively Deepening Depth First Search (IDDFS) (which is only experimentally supported), but there is no way to reduce the CPU time complexity. Even though KLEE does not have to finish testing a program completely to find bugs, safety critical software demands complete verification (or at least as complete as possible).

Figure 4.7 shows runtime statistics from KLEE for test using the DFS search strategy, where `Instrs` is number of executed LLVM IR instructions, `Time` is total time spent running, `ICov` is percentage of total instructions covered during testing, `BCov` is percentage of branches covered, `ICount` is the total number of instructions in the bitcode file and `TSolver` is percentage of time spent in the SAT solver. As the figure shows, KLEE has achieved approx. 77% instruction coverage and approx. 60% branch coverage.

```
--------------------------------------------------------------------------
|  Path    |   Instrs|  Time(s)|  ICov(%)|  BCov(%)|  ICount|  TSolver(%)|
--------------------------------------------------------------------------
|klee-last|52135650|302651.30|    77.61|    60.76|    1697|       99.39|
--------------------------------------------------------------------------
```

Figure 4.7. KLEE's runtime statistics.

[1] 2GHz AMD Opteron 6128

4.3 KLEE fitness for purpose

Overall we found KLEE fit for purpose, but with reservation. It has helped to find two previously unknown bugs in the MaCAN library and another one that has been independently found and fixed after I forked MaCAN for testing.

Because most safety-critical programs are hardware dependent and their execution is expected to never terminate, the software given to KLEE has to be modified and these modifications can potentially obscure existing bugs. Assuming no bugs were obscured by testing-enabling modifications, KLEE's analysis is sound (does not provide false negatives), which for safety-critical software is very valuable property.

The cost of KLEE's analysis is exponential time and space complexity and specific requirements on the design of tested software. This also means that for software that resembles eMotor, having weakly abstracted hardware-dependent code, the high time investment for testing said software with KLEE might not be worth it, depending on how critical said software is.

For software that does not use all of KLEE's capabilities, e.g. it does not use assertions, but rather uses it to find undefined behaviour invoking code, there also exist tools that sacrifice soundness of analysis for speed. An example of such tool is the STACK [20].

Chapter 5
Conclusion

In this work I demonstrated how to use KLEE to test safety-critical software. I described how to compile KLEE and what modifications were needed in the tested software. Because of KLEE's limitations and requirements, I could only proceed with testing of the MaCAN library and had to abandon testing eMotor.

The results from testing the MaCAN library are mixed. After running the tests for more than 5 days, KLEE had found three severe bugs. The first one was a signed integer overflow in message time validation, second was an independently found and fixed unsigned overflow bug in calculating current time and the third was a currently dormant bug in manipulation with the MaCAN system time. However, the best run of KLEE had only covered approx. 60% of all branches and 77% of all the LLVM IR instructions in the test program. Because the progress of KLEE's test coverage is non-linear and can generally be expected to be logarithmic, I predict that even after another 5 days KLEE would not have finished running the MaCAN test program.

Because of its results, KLEE will be added amongst tools used to continuously test the MaCAN library.

Thus we conclude that KLEE can be a valuable tool for verification of real-time safety-critical software, but the software has to be designed from the start with KLEE in mind. This means liberal use of assertions, minimizing complexity of branches, having strongly abstracted hardware-dependent code and documented parts that are intractable for symbolic execution, such as cryptographic primitives.

In closing, I think there is a niche for a symbolic execution tools targeted specifically towards real-time safety-critical software systems, niche that has not yet been filled with an already existing symbolic execution tool. It could provide a more comprehensive set of checks (e.g. no currently existing tool checks against aliasing violations) and have specific annotations to help with managing runtime complexity (e.g. to guide the symbolic execution tool towards failure paths).

References

[1] Nitesh Dhanjani. *Cursory Evaluation of the Tesla Model S: We Can't Protect Our Cars Like We Protect Our Workstations*. Nitesh Dhanjani's blog.
`http://www.dhanjani.com/blog/2014/03/curosry-evaluation-of-the-tesla-model-s-we-cant-protect-our-cars-like-we-protect-our-workstations.html`.

[2] Cristian Cadar, Daniel Dunbar, and Dawson Engler. *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.* In: *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation.* Hollywood, CA: USENIX, 2008. ISBN 978-1-931971-65-2.

[3] JTC 1/SC 22/WG 14. *ISO/IEC 9899:1999: Programming languages – C*. .

[4] *CVE of Linux kernel vulnerability caused by GCC's optimization of UB invoking code*. MITRE Corporation's CVE system.
`http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1897`.

[5] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. *A Formally-Verified C Compiler Supporting Floating-Point Arithmetic.* In: *ARITH, 21st IEEE International Symposium on Computer Arithmetic.* Austin, TX, USA: IEEE Computer Society Press, 2013. 107-115. ISBN 978-1-4673-5644-2.
`http://hal.inria.fr/hal-00743090`.

[6] James C King. Symbolic execution and program testing. *Communications of the ACM.* 1976, Volume 19 (Issue 7), 385 - 394.

[7] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. *S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems.* In: *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems.* California, USA: ACM New York, 2011. ISBN 978-1-4503-0266-1.

[8] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: A Software Testing Service. *ACM SIGOPS Operating Systems Review.* 2010, Volume 43 (Issue 4), 5 - 10.

[9] Brett Daniel, Tihomir Gvero, and Darko Marinov. *On Test Repair Using Symbolic Execution.* In: *ISSTA 2010: 2010 International Symposium on Software Testing and Analysis.* Trento, Italy: ACM New York, 2010. 207—218. ISBN 978-1-60558-823-0.

[10] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. *SymDrive: Testing Drivers without Devices.* In: *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation.* San Diego: USENIX, 2012. ISBN 978-1-931971-96-6.

[11] Celina G. Val. *Conflict-Driven Symbolic Execution: How to Learn to Get Better*. Master's Thesis, University of British Columbia. 2014.

[12] *Merging commit*. KLEE's github page.
`https://github.com/klee/klee/commit/a743d7072d9ccf11f96e3df45f25ad07da6ad9d6`.

[13] Will Dietz, Li Peng, John Regehr, and Vikram Adve. *Understanding Integer Overflow in C/C++.* In: *Proceedings of the 34th International Conference on Software Engineering*. Zurich, Switzerland: IEEE Press Piscataway, NJ, USA, 2012. ISBN 978-1-4673-1067-3.

[14] Uwe Kiencke, Siegfired Dais, and Martin Litschel. *Automotive Serial Controller Area Network*. . SAE.

[15] Oliver Hartkopp, Cornel Reuber, and Roland Schilling. *MaCAN - Message Authenticated CAN*. In: *ESCAR 2012*. Berlin, Germany: 2012.

[16] Ondřej Kulatý. *Message authentication for CAN bus and AUTOSAR software architecture*. Master's Thesis, Czech Technical University in Prague. 2015.

[17] Tristan Ravitch. *How WLLVM works*. WLLVM's github page.
`https://github.com/travitch/whole-program-llvm/#introduction`.

[18] Dingbao Xie. *Email in klee-dev mailing list*. klee-dev mailing list archive.
`http://mailman.ic.ac.uk/pipermail/klee-dev/2015-March/001010.html`.

[19] Michal Sojka. *MaCAN library UB fix*. MaCAN library's github page.
`https://github.com/CTU-IIG/macan/commit/38dd6a22cb1808ca3fd6386a498967932092d520`.

[20] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. *Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior.* In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. Farmington, PA, USA: ACM New York, 2013. 260-275. ISBN 978-1-4503-2388-8.

Appendix A Specification

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student:	Martin H o ř e ň o v s k ý
Study programme:	Open Informatics
Specialisation:	Computer and Information Science
Title of Bachelor Project:	The Use of Symbolic Execution for Testing of Real-Time Safety-Related Software

Guidelines:

1. Familiarise yourself with "symbolic execution" and the LLVM framework-based tool KLEE.
2. Explore possibilities of using KLEE for verification of safety-critical, real-time applications and analyse what guarantees using KLEE gives us.
3. Using KLEE, first analyse a simple library for motor control and then a complex software module provided by Infineon company for controlling electrical motors in cars.
4. Analyse the results and propose generalised methodology for verification of safety-critical, real-time applications.
5. Document the results thoroughly.

Bibliography/Sources:

[1] Cristian Cadar, Daniel Dunbar, Dawson Engler: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs - USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008).
[2] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift SymDrive: Testing Drivers without Devices. – USENIX Symposium on Operating Systems Design and Implementation (OSDI 2012).

Bachelor Project Supervisor: Ing. Michal Sojka, Ph.D.

Valid until: the end of the summer semester of academic year 2014/2015

L.S.

doc. Dr. Ing. Jan Kybic	prof. Ing. Pavel Ripka, CSc.
Head of Department	**Dean**

Prague, January 10, 2014

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student:	Martin H o ř e ň o v s k ý
Studijní program:	Otevřená informatika (bakalářský)
Obor:	Informatika a počítačové vědy
Název tématu:	Využití symbolické exekuce pro testování real-time, bezpečnostně kritického softwaru

Pokyny pro vypracování:

1. Seznamte se s technikou "symbolické exekuce" a nástrojem KLEE založeným na frameworku LLVM.
2. Prozkoumejte možnosti použití nástroje KLEE pro bezpečnostně kritické real-time aplikace a analyzujte jaké záruky nám jeho použití dává.
3. Analyzujte nástrojem KLEE nejprve jednoduchou knihovnu pro řízení motorů a poté komplexni softwarový modul pro řízení elektrických motorů v automobilech vyvinutý firmou Infineon.
4. Analyzujte výsledky a zkuste navrhnout obecnou metodologii pro verifikaci real-time bezpečnostně kritických aplikací.
5. Výsledky pečlivě zdokumentujte.

Seznam odborné literatury:

[1] Cristian Cadar, Daniel Dunbar, Dawson Engler: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs - USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008).
[2] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift SymDrive: Testing Drivers without Devices. – USENIX Symposium on Operating Systems Design and Implementation (OSDI 2012).

Vedoucí bakalářské práce: Ing. Michal Sojka, Ph.D.

Platnost zadání: do konce letního semestru 2014/2015

L.S.

doc. Dr. Ing. Jan Kybic
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 10. 1. 2014

Appendix B
Glossary

AUTOSAR	AUTomotive Open System ARchitecture — standardized open automotive software architecture
AUTOSAR MCAL	MicroController Abstraction Layer
CAN	Controller Area Network
CDCL	Conflict Driven Clause Learning
Clang	LLVM's C, C++ and Objective-C frontend
compiler-rt	LLVM's library for low-level, target-specific runtime components
CVE	Common Vulnerabilities and Exposures — A reference of publicly known information security vulnerabilities
GitHub	Free hosting for Git repositories
glibc	GNU C library — Most used C standard library implementation on Linux
IDDFS	Iterative Deepening Depth First Search — A Depth First Search with increasing depth limit
KLEE	Symbolic virtual machine built on top of the LLVM compiler infrastructure
LLVM	Formerly Low Level Virtual Machine, but the acronym is considered obsolete and not used anymore
LLVM bitcode	A binary stream encoding of LLVM IR
LLVM IR	LLVM's Intermediate Representation
MaCAN	Message authenticated CAN
MISRA	Motor Industry Software Reliability Association
SAT	Abbreviation of Boolean Satisfiability Problem — Determining whether given boolean formula can be satisfied
STP	Simple Theorem Solver — constraint solver
S2E	Selective Symbolic Execution
UB	Undefined Behaviour — A behaviour for which the C standard gives no guarantees
ubsan	Undefined Behaviour Sanitizer — A runtime instrumentation library for detection of undefined behavior in C and C++
uClibc	C standard library implementation aimed at embedded Linux
WLLVM	Whole program LLVM, python based utility script over LLVM used as drop-in replacement for compiler

Appendix C
Contents of enclosed CD

The CD enclosed with this thesis contains digital copy of thesis itself in the `thesis-pdf` folder, all code needed to repeat my tests, together with patches I made while working my thesis, is in the `thesis-setup` folder and all raw results from KLEE runs I used to make the final report in the `thesis-data` folder.

The `thesis-pdf` folder contains:

- TEX document to recreate this thesis
- TEX dependencies for said document
- Cited papers in `.pdf` format

for obvious reasons, resources cited from the web are not included

The `thesis-setup` folder contains:

- LLVM-3.3, Clang and compiler-rt tarballs
- `MaCAN-diff` a diff of all changes I made to MaCAN
- `thesis-setup-stable` folder with the rest of dependencies
- KLEE, STP, WLLVM and MaCAN, checked out at version used in this thesis
- `HOW_TO_INSTALL` and `HOW_TO_USE` files — exactly what it says

eMotor is not there because of its proprietary nature and the NDA I had to sign to work with it.

The `thesis-data` folder contains:

- `readme.txt` file to explain the folder organization
- A folder for each KLEE run used in the thesis
- Calltrace for each KLEE run
- All files created by KLEE during a run

Appendix D
Test details

This appendix contains list of all flags given to KLEE when testing the MaCAN library, an overview of results and statistics of all runs.

I used three independent instances of KLEE to test the MaCAN library. I shall refer to them as `covnew`, `dfs10` and `dfs15`. Abbreviated contents of their respective `info` files are in Figure D.1, Figure D.2 and Figure D.3 respectively.

```
klee --optimize --max-memory=0 --check-div-zero --check-overshift \
--search=dfs --search=dfs --search=dfs --search=nurs:covnew
Started: 2015-05-14 15:43:38
```

Figure D.1. Contents of `info` file for `covnew`.

```
klee --optimize --max-memory=0 --search=dfs --check-overshift \
--check-div-zero
Started: 2015-05-14 15:49:56
```

Figure D.2. Contents of `info` file for `dfs10`.

```
klee --only-output-states-covering-new --optimize --search=dfs \
--max-memory=0 --check-div-zero --check-overshift
Started: 2015-05-15 23:26:36
```

Figure D.3. Contents of `info` file for `dfs15`.

The difference between `dfs` and `covnew` is that KLEE is allowed to use its own heuristics for guiding the search in the `covnew` invocation. The difference between `dfs10` and `dfs15` is whether the MaCAN hardware implementation for KLEE serves 10 received messages in row, or 15.

Type	Potential problems found
10dfs	8
15dfs	5
covnew	3

Table D.1. Overview of KLEE runs results.

```
----------------------------------------------------------------------
|  Path | Instrs |  Time(s)|  ICov(%)|  BCov(%)|  ICount| TSolver(%) |
----------------------------------------------------------------------
|15dfs  | 4547644|416400.30|    75.37|    57.85|    1697|       99.93|
----------------------------------------------------------------------
|10dfs  |52135650|302651.30|    77.61|    60.76|    1697|       99.39|
----------------------------------------------------------------------
|covnew | 4834942| 51827.84|    54.10|    38.66|    1697|       99.59|
----------------------------------------------------------------------
```

Figure D.4. Runtime statistics across used tests.

Table D.1 shows an overview of KLEE runs and number of potential bugs they reported, and Figure D.4 shows their runtime statistics.

Interestingly, even though `covnew` and `15dfs` have been started only 6 minutes apart and stopped inside couple of minutes of each other as well, for the `covnew` run KLEE reports that it has ran for only about tenth of the time `15dfs` has ran for. The exact reason for this is unknown, but I speculate that it is because the `covnew`'s memory consumption quickly ballooned up to approx. 3.2 GBs and then it hit a KLEE bug with memory consumption. This is supported by two facts. First, an empirical observation: When trying to run KLEE with given command line options again, it quickly hit 3.2 GBs of memory consumption and then its memory consumption doesn't change for hours. Secondly, there are currently known bugs in KLEE's memory management and it might be related. This is supported by fact that while using the DFS search strategy, KLEE's memory consumption is a lot smaller and KLEE executes normally.

Taking these facts into account, together with overview in Table D.1 and test results from Chapter 4, I have to recommend using KLEE command line invocation from Figure D.5.

```
klee --only-output-states-covering-new --optimize --search=dfs \
     --max-memory=0 --check-div-zero --check-overshift <bitcode file>
```

Figure D.5. KLEE command line invocation.