Masaryk University
Faculty of Informatics

Automatic Bug-finding Techniques for Linux Kernel

Dissertation Thesis Topic

Jiří Slabý

Brno, spring 2010

Supervisor: prof. RNDr. Antonín Kučera, Ph.D.
Supervisor-specialist: Mgr. Jan Obdržálek, PhD.

Supervisor's Signature: .....

Acknowledgement

I would like to thank supervisors, both Prof. Kučera and Dr. Obdržálek for their patience with my ignorance. I thank my colleagues for their ideas and inspiration. And of course my thanks belong also to my parents for their endless support.

Contents

1 Introduction **1**
1.1 Kernel Code 2
1.2 Intention of Thesis 3

2 Report on Current Results **5**
2.1 Abstract Interpretation 5
2.2 Current Tools Overview 5
2.3 Tools Summary 8
2.4 Algorithms over Abstraction 9
2.5 Current Status of SE 10
2.6 Kernel Checking Status 14
2.7 Other and Related Work 15

3 Dissertation Thesis Intent **17**
3.1 Schedule 18

References **21**

4 Summary of Study Results **22**

1 Introduction

Computer programs are a result of human work. This often means that more complex software contains errors. It is widely traded that the later an error is found in the development cycle the higher cost of the fix is.

Hence to lower the cost of distributed software, software developing companies started to produce tools for pruning errors in early development stages. Today, there also compete many companies focusing solely at finding bugs.

During the years of bug-finding tools development, several techniques emerged. In principle, code can be checked with and without a need of being really executed on a processor. This divides the techniques to *static* (do not run) and *dynamic* (they do). Of course, each of them further contains subcategories covering specific testing needs.

Within a scope of the dynamic technique, programs are usually exposed to attacks of inputs and are expected to sustain the attack, correctly finish their task and exit without a crash (i. e. either successfully or with error reported). Dynamic testing ensures no false positives, since the code is run in real-time with real data. Thanks to this and other factors such as performance of single test and easy setup, it is very common testing approach nowadays.

While standing on the opposite side, static code analysis is currently also considered to be a powerful approach used for finding errors in a source code. Although it is not a new technique, one of the first real applications, capable of finding many errors, was performed by Engler et al. in XGCC [18] even in 2000.

The static analysis might have a form of proving properties of mathematical models known as *Model Checking*, checking of programmer's properties specified in special code constructs, i. e. *checking assertions*, abstract the code and try to verify properties on the abstraction, so-called *Abstract Interpretation* and others. My dissertation thesis will be in the scope of Abstract Interpretation. Obviously, there are some problems connected to it and the thesis will tackle one sort of them, as will be seen later.

For the abstraction, it approximates behaviour of a system for easier checking. For example checking whether there is an assignment of π to a variable no matter where, an analyzer simply looks for assignments and π. Any other code is irrelevant and may be ignored in the abstraction. In the best case, all the unneeded details of the system are thrown away and only relevant parts remain.

However, usually the abstraction is not as good as that, the abstraction turns out to be too coarse often and checking thus becomes unsound. As a response, false positives start to occur. They might be beaten by either revisiting the error, generating an input passed to the system dynamically or finally, narrowing the abstraction.

Opposing to the coarse abstraction, the computation may last too long and seem to never finish. It might mean that the abstraction is too detailed, so it should be widened to lose more details. For these operations, widening and narrowing, a formalization exists and will be mentioned in Section 2.1. The thesis will be based, as many other static analyzers, on the top of that work.

With some level of abstraction settled, an analysis method over the abstraction

is still to be chosen. Some chose to find patterns in the code in some specified manner (similarly to XGCC), others sticked to a test cases generation. The test cases try to cover the most of the code (executing both true and false branches of each branching).

Many tools were developed based on either approach or alternatively by intertwining the methodologies to produce better results. Section 2.2 introduces a reader to the methods and their use in currently known mainstream tools.

Symbolic Execution Besides the mixing of the techniques above, some add another principles such as *Symbolic Execution* (SE). Currently, it seems to be a promising code analysis form. SE uses external theory solvers to decide whether (statically) executed paths in the code are feasible. Thus it eliminates many false positives, because they are often caused by traversing infeasible paths in the code. Static checkers not based on SE usually have minimal chances to catch such bad behaviour except matching patterns in consequent branching. But it does not work well e. g. for operations with switched operands (`if (a < b)` equals to `if (b > a)`) and similar.

Many studies tried to apply the principle on a variety of userspace programs. Early symbolic executors aimed mostly at tests generation that would cover as many program instructions as possible (EXE, KLEE). The tests are then run dynamically, which has already mentioned advantages. Concurrently, some researches started to employ SE in cooperation with concrete execution on a processor to address program coverage as well, but from another point of view (DART, PEX, CUTE). See Section 2.4.2 that is deserved for detailed description of the work done in this area.

Although EXE already experimented with applying SE on parts of an operation system kernel (further only *kernel*), to the best of my knowledge nobody ever executed kernel in a fully-automatic manner without a necessity of configuring a large testbed.

The task of SE to be run on a kernel incorporates not only the execution itself, but some kind of code preparation, compilation and linking, the same as for userspace, but with some specifics. These terms will be discussed further in the text (Section 2.5.1) and also in the thesis intent (Section 3).

1.1 Kernel Code

Deduced from the topic title, the thesis will primarily concentrate on checking the kernel code. However, the code is very specific in few ways and needs special handling. If we do not count virtual machines[1], the very difference is probably that it does not run as a regular task.

It has several consequences, especially when run dynamically: (a) it cannot be stopped and restarted at will, (b) it cannot be forked to run multiple kernels in parallel and (c) whole system crashes if something bad occurs during the tests.

From the source code perspective, the kernel is similar to user applications, but with one advantage and disadvantage. The disadvantage is that almost every function in the kernel may run concurrently with each other unless locking is performed.

[1] Section 2.6.1 mentions their use for kernel checking, but we do not think it is the best way as we will see later.

The better on the kernel is that source codes are self-contained. All functions are present and may be found on demand. This may help the analysis in that there are no black boxes to deal with.

Special care should be taken even on what to check in the kernel. The most interesting properties are to check whether inputs may crash a whole system. Input passed down to the kernel may come from several sites. Major of the paths is untrustworthy and must be properly checked for bogus data. There are many exploits known to abuse *system calls* this way such as in [8, 26, 29][2]. Similarly, *writes to special files* or even sending *packets with special structure* over network remotely may be abused for attacks. The attacks often lead to gaining administrator privileges on the local system or to a system crash – the denial of service.

1.2 Intention of Thesis

The previous paragraphs outlined what can injected user data cause to the kernel. The dissertation should aim at the input from user or hardware (network) and check whether there are proper tests in the code, so that it cannot be used for exploiting or crashing the system rather easy.

In other words, finding security holes should be dissertation's primary goal to disallow the described attacks as much as possible. At that phase, the tool will not be limited to check a user input only, but also well-known errors (such as NULL-pointer dereferences) inside the kernel.

Since SE has shown its power when finding errors in userspace programs in the past decade, it looks like a reasonable approach to be applied also to the kernel. However it is not as easy task as it might seem. The same as for userspace programs and static analysis altogether, for successful SE kernel analysis few basic steps are necessary as well: (a) processing the kernel code, (b) finding the right properties to check, (c) finding them in the processed result and (d) verify if they hold. This process is depicted in Figure 1. The dashed line there shows a dependence of specification on the source code. That is, patterns to find as a part of point (c) usually need to be specified in some representation of the intermediate language.

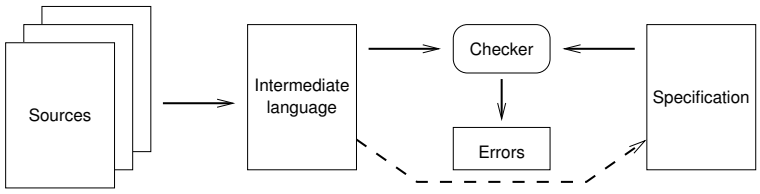

Figure 1: Code checking procedure

There are already components in many tools that can be reused as a basis for each of the drawn step. Exe was successful in all four for picked parts of the kernel, but mostly the parts are not subject to a user input or special instructions. As far as we know, there is still no tool that would run SE on a kernel thoroughly and

[2]See other reports at `http://nvd.nist.gov`.

independently on sybsystem. Also the tools (Exe including) do not use portable intermediate languages or other parts of the system (e. g. a solver).

To fill this gap, the dissertation should lead to a complete implementation of the mentioned points and should find errors with minimum false positives possibly with a help of SE. All of that with minimal input from user (except specification of additional checking properties) and with forward portability to not be blocked by underneath layers (solvers, languages, front-ends) and their versions in the future.

Dissertation Basis There is a wide set of tools for static analysis already out there. By now, we[3] contributed by a free (open-source) tool called Stanse. Some of its parts will be a start-point for a bug-finding tool developed in the scope of the thesis. For example all of batch execution taken over from Sparse, specifying properties from xgcc and graphical interface inspired by FindBugs may be based on Stanse's. Stanse and its functionality is characterized in Section 2.2.

This is not enough, since Stanse is not a symbolic executor. Hence e. g. Klee and its well-portable code processing may be adopted with a limited support of x86 assembly presented in Sage.

Unfortunately some principles were captured in papers, but their implementation is not freely available. It is because they were implemented only in commercial tools. These methods should be either avoided or reimplemented under some permissive license.

[3]at the Institute for Theoretical Computer Science (known as ITI) at the Faculty of Informatics

2 Report on Current Results

Static analysis may be utilized for software testing and/or finding errors. In this section, one of the static mechanism, abstract interpretation, is presented along with tools and principles based on that. It is because of its close connection with our past, present and future doctoral studies, i. e. to reveal also the thesis intent.

2.1 Abstract Interpretation

Abstract interpretation is a relatively old approach, first summarized and formalized by Cousot & Cousot in [9] in 1977. There they specify a formal model used in interpretation (a) to construct a correct abstraction of the code and (b) to be able to compute a solution of some properties defined on the abstracted code (by fixpoints).

Note that checking of properties in programs is in general *undecidable*, hence they also define narrowing and widening techniques if the system of equations does not converge to a solution while computing the fixpoint.

2.2 Current Tools Overview

Cousots laid down the foundation for others and the principle attracted attention again in 2000. It was thanks to work of Engler et al. on a meta compilation based technique described in [12]. System developers there specify properties of the system in METAL [18] language, compile it into internal representation and the tool applies the specification on all executable paths in a particular function. Violation of the specification on a path is then reported as error.

The compiler used for METAL compilation is GNU C compiler (GCC) 3.1.1 with their extensions – XGCC[4]. It redefines GCC's internal code representation, data types and applies the analysis on such modified structure.

By improving the technique[5] they developed a commercial derivation of XGCC named COVERITY PREVENT [36], the current world leader in static analysis, according to developers. Currently it analyses many production systems including the Linux kernel on a daily basis[6].

Programming languages supported by XGCC are C and C++. COVERITY adds support for C# and Java.

Aside from Engler's work, Klocwork started to develop their bug-finding static analysis tools ([38]) in 2003. Currently they, as same as COVERITY, find errors in many production projects. The same as COVERITY, Klocwork products are able to parse C, C++, Java and C# programs. They seem to be world number two in code analysis.

Open-source Apart from XGCC, all the mentioned tools till now are commercial products without an access to their source code. However there are also many open-source tools available. LINT derivatives belong to the *first static analysis tools* used

[4] Homepage at `http://www.cs.stevens.edu/~wbackes/xGCC/index.html`.

[5] According to the press release at http://www.coverity.com/html/press_story04_02_02_05.html, it produces less false positives, can run in parallel on the code and contains a statistical engine.

[6] `http://marc.info/?l=linux-kernel&m=125856033627134&w=2`

whatsoever. LINT used to find suspicious and non-portable code constructs. Mostly, the checks are currently superseded by compilers that perform the checking during standard compilation phase, so that there is no reason to use LINT anymore.

There was also a popular successor LCLINT [13] later renamed to SPLINT. The development stalls for more than five years[7] now though. Again LCLINT performs many checks that are done by compiler nowadays. Besides that, it can catch buffer overflow vulnerabilities. It also interprets *annotated code* and tries to avoid false positives that way (it allows turning on more checks which would, without the annotations, generate many false positives).

Later, tools dedicated for a *single purpose or project* emerged. Specifically for the Linux kernel purposes, SPARSE [40] was developed. For example it reports uses of number as a pointer (`0` instead of proper `NULL`), casts that truncate constants and other, kernel specific, properties (elaborated in Section 2.6). Many of the checks have no sense other than in the kernel. SPARSE is widely used by kernel developers and many SPARSE errors are fixed in each kernel release cycle[8]. Note that since this tool is able to check whole kernel, it is capable to parse most of GNU C extensions described in [30]. It is distributed with a system that intercepts build process (`make`) and is able to eavesdrop compiler command-line. It is then used to preprocess sources and check them as a next step.

From a different point of view, static checkers need not to be only *dedicated programs*. CLANG [35] compiler contains a built-in non-configurable static analyzer. With a use of internal compiler structures, it looks for typical programming errors like buffer overruns, use of deprecated (insecure) functions, a sort of memory leaks etc.

All previous open-source tools are configurable only to some extent. There is no way of specifying a new check except changing the sources. UNO [19] is a *fully configurable* tool. It allows user to specify for what type of bugs should the checker look for. The tool is distributed with predefined ones (for illustration a use of uninitialized variables), but others can be easily added. UNO is the only open-source tool from these with a support of *inter-procedural analysis*. Unfortunately its parser does not even support full ANSI C99 standard[9].

Still, some tools are hard to use. There is one exception called FINDBUGS [37]. To the best of my knowledge it is the most known code-checking tool for Java sources. FINDBUGS is capable to find hundreds types[10] of errors in whole Java project. Its advantage is a sophisticated *graphical user interface* (GUI) which turns FINDBUGS in a single button tool.

Stanse In the Introduction, STANSE was quickly mentioned as a static analyzer developed at ITI by J. Obdržálek, J. Slabý and M. Trtík (alphabetical order, I will narrow down my role later in Section 4) with a help of master's students. For the ease of programming, the tool is written mostly in Java. Further, some parts are

[7] `http://www.cs.virginia.edu/pipermail/splint-discuss/2007-July/000994.html`

[8] For instance, over 20 of SPARSE errors were fixed in 2.6.32 (since 2.6.31).

[9] Minimal input to demonstrate that is `_Bool b;`.

[10] `http://www.ibm.com/developerworks/java/library/j-findbug1/`

pure C and the rest (scripts) needs Perl. All the components are released under the GPLv2 license and hence are open-source.

Stanse tries to pick all the pros from previously named tools and also adds some novelty. It consists of three checkers:

- **AutomatonChecker** – technique based on finite automata. More detailed description is later in Section 2.4.1.
- **ThreadChecker** – Jan Kučera developed that in the scope of his master's thesis. The checker looks for thread-creation functions and statically follows the threads. Namely, it watches lock operations and builds *locksets* [27], represented as resource allocation graphs (RAG). The checker combines RAGs of all threads and tries to find cycles there. If it is successful with the search, an error is reported[11].
- **ReachabilityChecker** – It is a short checker (below 200 lines of code) that performs reachability analysis of statements to reveal unintentionally dead code. It catches errors like omitted semicolons after the if condition, so that the branching is not done properly[12] (`if (error); return;`). The true branch here would be an empty statement and `return` executed every time rendering the following code unreachable.

For simplicity, the last checker needs no configuration. The former two are fully configurable though. A user can specify properties to check, functions to watch and importance of found errors. All expressed by XML structures.

But before the checking itself is started, files to check need to be known. Stanse supports several methods of specifying input files. The most convenient way is to use batch files generated by a build system of code to be checked. That is after specifying project's `Makefile`, the checking will be performed automatically on all its files.

Then, on a file basis, C language with most GNU extensions is supported, thus it can parse whole Linux kernel. A speedup still was needed to support such big code bases. Each preprocessed file is split and one part which originates from external files (`#includes`) is thrown away after remembering `typedefs`. They are the only information needed for the second part, which is the original file itself after preprocessing.

Sometimes few of the preprocessor expansions of code constructs are unwanted. For example `spin_lock` call in the kernel may be expanded to many different variants depending on the kernel configuration. Stanse avoids that by renaming these functions to disable the expansion.

Another problem arises after parsing hundreds of files. For comparison, in the kernel, typically about 7000 files are compiled for one architecture. It is obvious, that they cannot fit in the operation memory. To tackle down this issue, streaming was added. UnitManager class contains a watermark with count of parsed files and throws away those, which are unnecessary at that time point. If they are needed

[11] For example one error report is available at `http://lkml.org/lkml/2009/4/14/527`.

[12] Such bad behaviour was fixed e. g. by `http://patchwork.kernel.org/patch/56386/`.

later again, they are automatically streamed in. The current implementation purges the ones touched the earliest, i. e. the LRU algorithm.

While having internal structures parsed from a source file, inter-procedural analysis is performed. It is implemented as connecting *Control Flow Graph* (CFG) node of a call with CFG of called function. A summaries computation similar to [18] is yet to be implemented.

For language support of Stanse, the source codes are not strictly bound to a single language, not even to a single compiler version (cf. xgcc). The front-end may be easily extended or changed thanks to modular structure of the tool.

Modular approach also allows multiple forms of user's input. Inspired by FindBugs, everything what can be done on command-line is also available in the Stanse GUI. It renders the tool into one-button checker. Sometimes it is easier to dump results via command-line switch into XML for later inspection either in GUI or web interface. Or simply to generate statistics about bugs and performance of the tool.

Stanse as a Framework Stanse is not only a checker. It provides easily extensible framework for checking. All code structures such as AST and CFG are available to the programmer. For the convenience, Stanse offers traversal functions over these structures as well. A unified error reporting interface is exposed too, so checker programmers need to implement only relevant parts of their new checker. A good example of simplicity is ReachabilityChecker.

2.3 Tools Summary

To sum up what was written about the tools, Table 1 depicts their overview. *Configurable* there means, again, that new checks may be added to the particular tool without a need of source code update.

Tool name	Open-source	Configurable	Batch support	Inter-proc.	Languages
Coverity	no	yes	yes	yes	GNU C, C++, C#, Java
Klocwork	no	yes	yes	yes	GNU C, C++, C#, Java
Splint	yes	no	no	no	C
Sparse	yes	no	yes	no	GNU C
Clang	yes	no	no	no	GNU C*, C++, Obj-C
Uno	yes	yes	no	yes	C†
Stanse	yes	yes	yes	yes	GNU C
FindBugs	yes	no	yes	no	Java

*Clang is still under heavy development, its GNU C support is incomplete.

†C support in Uno does not implement full C99.

Table 1: Overview and comparison of described tools

2.4 Algorithms over Abstraction

The previous section introduced solely to the tools currently known and used. This section will try to cover the main algorithms implemented over abstract interpretation in some of them.

2.4.1 Automata

XGCC uses state machines (SM) over abstraction. The machine specifies some rules that should ever hold for the code. See Figure 2 for example of SM representing states of pointers in C. Transitions are performed by a code matched against patterns like `alloc` and `free`.

For illustration, if the automaton is in the initial/freed (`I/F`) state and there is a pointer dereference in the code, the error state (`Err`) is reached and the incident reported. From `I/F`, there are also two allocation arcs. It is because the allocation may fail, to cover both cases.

Aside from dereferences, the automaton reports an error when memory is leaked. In this case a valid (`V`) memory is lost by another allocation (the bottom-down arrow in the right). Note that for simplicity the automaton does not depict the case of (correct) allocation from null (`N`) to `V`.

Much code behaviour may be described by automata. STANSE contains thorough automata descriptions for memory, locking (lock, unlock, try_lock, nested locks up to 5 levels), operation pairing (mostly for reference counting) and atomic functions (ones make atomic contexts, others de-atomize and no atomic-unsafe operations, e. g. sleeps, may be in-between).

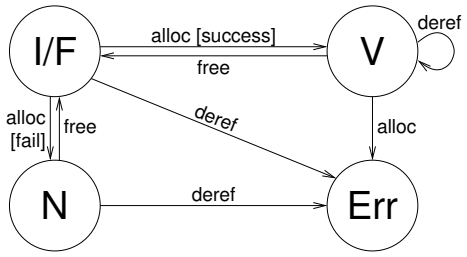

Figure 2: Automaton representation of memory pointers

For complete report on how automata internals are implemented in XGCC, see [18]. STANSE's AutomatonChecker is largely influenced by XGCC's technique except for few details: (a) we use XML for patterns notation (excludes meta-compilation and dependency on target language) and (b) the inter-procedural analysis is performed only per file, but, thanks to that, in a path-sensitive way.

2.4.2 Symbolic Execution

So far I have not described in detail one special code interpretation method called symbolic execution. As outlined in the Introduction, it stems from [5]. The principle is that chosen variables are tracked symbolically rather than concretely.

```
1 x = read();
2 if (x < 2)
3   lock();
4 compute();
5 if (x < 2)
6   unlock();
```

Figure 3: Symbolic execution example

Consider the imaginary code in Figure 3. `x` is initialized by user to some integer value and tested at lines 2 and 5 for the same condition (`x < 2`). If `x` is not handled symbolically, the execution may run through the true branch (`lock`) first and then through the false one (no `unlock`). This would generate a locking imbalance error in the code even though it is a clear false positive.

Dynamic testing will likely fail to cover both branches here in many cases too. Random testing may take ages to successfully hit range $[0 \ldots 1]$. Unit tests might be better, but if there are many such conditions in the code, the tests are hard to write comprehensively.

Now, consider `x` being handled symbolically. In such case, a path condition `x < 2` at line 2 is remembered. An executor is not then allowed to traverse the code via the false branch at line 5 since the previous condition would be violated. It has two outcomes, the code is properly executed such that either the lock is both locked and unlocked or not touched at all. Second, the other path may be traversed on the next run easily by inverting the condition and feeding corresponding input.

The next section will cover evolution of SE and its current state of the art.

2.5 Current Status of SE

Symbolic execution was already successfully deployed in software testing. To the very first tools with SE inside belong Exe [7] and Dart [16]. They were improved, forked or rewritten from scratch and potentially renamed. Also many tools appeared based on their combinations or just with few principles picked from them. To sort out these connections, this section will dive deeper into their internals.

Dart and Pex Dart was developed by Godefroid et al. at Microsoft Research and is designed to use SE in cooperation with standard (concrete) execution on a processor. This principle, first mentioned in [21], is used here to generate tests. The tests are intended to cover as much code as possible. In fact, the method leads to almost 100% code coverage[13] and hence it generates tests for almost all possible code paths. Dart works on the top of C language.

In detail, Dart first scans the source code by a simple static analysis tool to find a so-called *interface*. It is a description of external program inputs such as input specified by user on command-line (parameters of top-level functions, e. g. of `main`), external functions etc. This interface is then used as a potential symbolic input.

[13]The problem is undecidable in general, approximations may lead to imprecision and lower coverage.

Once the interface is known, a *test driver* is created. It is a small code hunk written in C that represents the interface. It initializes top-level function parameters with random values and also implements external functions found in the previous step. These stubs return random values as well.

With all the code known, DART starts SE, initially with random inputs. Since it wants to cover both true and false branches of every branching, it executes the code repeatedly with the input corresponding to the opposite of each branch every time. To demonstrate it on our example in Figure 3, consider DART goes through the true branch first time, i. e. with path condition `x < 2`. Next time it will add `x >= 2` to a path condition and generates the program input accordingly to cover even the false branch.

Sometimes it may be infeasible for several reasons[14] to obtain some values symbolically. In such a situation DART falls back to the value from dynamic execution. In this case, testing completeness is of course lost.

PEX [31] is a successor of DART working with .NET applications. The same as DART, PEX executes the code dynamically and statically in parallel. Its difference lays in parametrization of tests specified by a user here. It might seem similar to the standard unit testing known from *extreme programming*, but exactly the parametrization makes the difference.

Each method (test itself) has several parameters, which specify inputs. Take a look at slightly modified example taken from [31]:

```
1 public void TestAdd(int capacity, object element) {
2   ArrayList a = new ArrayList(capacity);
3   a.Add(element);
4   Assert.IsTrue(a[0] == element);
5 }
```

With standard unit testing, one would need to specify several `element` objects and `capacities` to test `ArrayList` implementation. However, given `TestAdd`, PEX will exercise all `capacity` and `element` combinations which branches `ArrayList` implementation differently. In turn it covers more code in shorter time without a programmer's help.

PEX, with one of its later extensions ([1]), may look at the source code differently. Unlike generating tests for all possible paths, a location to explore may be known in advance. One example for all is `assert`, because its location can be easily obtained from the code. In such situations a *demand-driven analysis* may be incorporated. This approach reduces time spent by analysis significantly, because only paths that really lead to the target are taken into considerations.

The very same paper also mentions *compositional* analysis. It uses already mentioned summaries to represent functions' behaviour. Functions with the same environment (same constraints on parameters and heap) need not be executed twice. A value/constraint returned in previous run is used instead.

Developers may try PEX to check their .NET applications, because it is distributed as an extension for Microsoft Visual Studio.

[14] Non-linearity in expressions, dereferences of user-provided pointers etc.

Cute Also CUTE [28] is based on parallel execution principle. However it tries to address pointer parameters of test functions. A structure behind the pointer is represented as a *memory graph* and its contents is changed according to branches in the code containing pointer dereferences.

For example if the code contains `if (ptr->member == 1)`, for the first time, CUTE will construct random memory map for structure behind `ptr`. Second, the map will contain 1 on the `ptr->member` position (or some integer other than 1 in case the random input was 1 already).

Exe and Klee Contrary to the previous work, a tool from Engler et al. called EXE [7] executes the code solely symbolically. EXE works as follows.

A user *specifies variables* (or arrays, structures) that should be handled symbolically by the `make_symbolic` function. This is further compiled by `exe-cc` compiler and later compiled along with the code by standard compiler.

Before the compilation, each branch, assignment and expression from the original code must be *instrumented*. It has two reasons. First, operands of these operations are checked whether they should be handled symbolically (at least one operand is symbolic) or the same as in the uninstrumented code (all are concrete). Second, each branch contains a `fork` call to (natively, on the OS level) split the execution.

Each cloned child contains negated last element from path condition and hence walks the true or false branch respectively.

With the instrumentation described, resulting binary is normally *run*. It may finish either by standard way (`main` returned, `exit` called etc.) or with an error (crash, `assert` parameter is false, EXE detected an error such as zero division, ...). In all cases, inputs corresponding to each path constraint are dumped. When dumped from the error case, it is an "input of death" in their language.

They designed also their own tailor-made solver. This allows EXE to perform certain operations and optimizations that cannot be done by other tools when using external solvers. EXE's key improvements are constraint caching, independent constraint optimization, bitvector arithmetic and tracking indirect memory accesses symbolically. This allows them to speedup checking significantly, since solvers take huge amount of time from the checking procedure. See [7] for technical details.

With this tool they checked three filesystems from the Linux kernel by writing a special block device driver. It injects a "corrupted" filesystem structure. They were also successful in finding bugs in firewall implementations/interpreters of BSD and Linux. This problematics is also touched in Section 2.6.1.

However there is still one notable bottleneck in the algorithm: `fork`. It does not belong to the fastest system calls[15]. Hence KLEE [6], next Engler's work, tries to tackle this disadvantage.

Instead of `fork`, they wrote their own *process scheduler*. There is one immediate result – scheduling is in their hands and can thus be parametrized and tuned. Created children are scheduled by heuristics to not unroll infinite loops and to cover as much code as possible early.

[15] Whole process space needs to be copied, all per process structures allocated in the kernel etc.

Another improvements in KLEE are further simplification of path conditions before solving, the solving itself and also lower memory consumption targeted by condensed representation of created children. Intermediate representation was changed too. *Low Level Virtual Machine* (LLVM) is used for symbolic execution[16]. Section 2.5.1 contains more details about LLVM.

KLEE is distributed with *C library implementation* so that external C functions are not black boxes anymore. Those that still cannot be resolved (including system calls) are tried run dynamically with reconstructed environment and parameters back from symbolic path constraints. If that fails too, KLEE refuses to check the program and forces user to provide sources of that unknown function.

Similar to EXE, it dumps an input for programs for every possible code branch, allowing near 100% code coverage.

With a use of these methods, they were able to find three severe bugs in CORE-UTILS, the package of tools used daily on most Linux machines.

Sage All previous work was based on some intermediate language representation. SAGE [17] is unique in that it symbolically executes native x86 instructions. Immediate advantage is that it needs no source codes. The program is linked in a standard way and all code is available. This allows even close-source code to be exercised.

What is worse are the instructions themselves. According to the Intel processor manual[17], there are almost 400 basic instructions. Most of them has to be supported by the tool[18]. And with new processors, this count grows.

One more SAGE novelty is *generational search* that maximizes number of generated inputs on each execution. It creates inputs for each negated component of a path constraint at once. I. e. if the path condition was $a \wedge b \wedge c$ and first run satisfies all of them (true branches), next run will generate inputs with $!a$, $a \wedge !b$ and $a \wedge b \wedge !c$ path conditions at once (if satisfiable indeed).

2.5.1 LLVM

LLVM [39] seems to be a future of compilers. The support of languages in it is already big. Currently, C, C++, Obj-C, Ocaml[19] and Java are supported. Others are planned, it is a matter of implementing a front-end.

There is a specially designed instruction set that every front-end should compile into. The set is reduced, with only tens of instructions. As it is language-independent, it allows several language-independent optimizations, separately from front-ends.

The language may be also emitted to a file (as a plain text or bytecode), so that it can be used arbitrarily e. g. for symbolic execution. A big advantage is that LLVM framework provides also libraries for LLVM code traversing. This renders

[16] Hence the tool is likely extensible to arbitrary language supported by LLVM framework.

[17] Available at `intel.com`.

[18] Some are often omitted in the implementations. For instance floating point operations have no support, since checkers do not track floating point variables either.

[19] Obj-C is a language with added objects functionality to the C language. The same for Ocaml, a functional language based on ML with objects, see `http://pauillac.inria.fr/ocaml/`.

"parser" for symbolic execution quite easy and fast, because only few instructions have to be supported.

Although this may seem perfect, inline assembly thrashes the whole idea of few instructions. The assembly is hard-copied from source code to the LLVM code and hence tools have to take care of it properly. A simple solution is to ignore it completely, however for the kernel, it may lead to inconsistencies due to high inline assembly occurrence rate.

What is worse, none of contemporary C front-ends (`llvm-gcc` and `clang`) is capable of parsing the kernel code seamlessly. GCC is still a superset of both of them in what language constructs are supported. Personally, I already reported two bugs from this area in a hope of fixing. Both of them are a matter of unsupported inline assembly operations found in the kernel.

Still, LLVM turns out to be eligible choice nowadays because it is under heavy development with release schedules. Hence it looks like missing features will be added in the near future.

In addition, this framework was used in KLEE. Hence, due to licensing, KLEE may be freely obtained from LLVM site.

Toolkit We have decided to use LLVM as an intermediate representation in my dissertation unless some blocker appears. Hence we will briefly introduce the toolkit here. Current stable version is 2.6, all facts drawn here are with regard to this release.

To obtain a LLVM code, compilers understand an `--emit-llvm` parameter. Although the LLVM code is generated to a separate file for each source file, it is still possible to have the code all-in-one. For that purpose `llvm-link` and `llvm-ld` exist and differ only in details. When all files are linked together it is one single LLVM code which external tools (checkers included) accept as input.

The input is handled through LLVM framework written and exported in C, C++ and Ocaml languages. It designates the tool to be written in one of those languages.

2.5.2 Summary

Similar to static analyzers earlier, Table 2 depicts a comparison of described SE tools. *Mixed analysis* column shows whether tools are capable of simultaneous concrete and dynamic analysis. *Base* enumerates intermediate languages which the tools are based upon.

2.6 Kernel Checking Status

It was already stated in Section 2.2, that the Linux kernel is already being checked by static tools, namely by SPARSE. Few simple examples of checks were given in that section too. Above that the tool can watch pointer usage on user- and kernel-space interface, since they must not be interweaved, or uses of remapped I/O memory which has to be dereferenced via special helpers. A complete list is available in SPARSE's manual (`man`) page.

Tool name	Mixed analysis	Open-source	Free to use	Base	Languages
DART	yes	no	no	CIL [25]	C
PEX	yes	no	yes	.NET	.NET
CUTE	yes	yes	yes	CIL	C
EXE	no	no	no	n/a*	C
KLEE	no	yes	yes	LLVM	C
SAGE	no	no	no	x86	n/a†

*Not applicable. The code is instrumented and then run dynamically.

†Not applicable. Everything compiled to a x86 binary may be checked.

Table 2: Overview and comparison of symbolic tools

2.6.1 Exe and Kernel

Even EXE was used for testing picked parts of the Linux kernel. Specifically, three filesystems were checked in [33]. A disadvantage of EXE is that it has to run as a standard system process to be clone-able by `fork` (compare to the description earlier). They tried to exempt pieces of code relevant to the filesystem and use them in userspace first, but they failed. It is because the code was still dependent on the rest of the kernel in some way[20].

Finally they felt back to *User mode Linux* (UML)[21] adaptation. It virtually runs kernel as a userspace process which exactly matches EXE's needs. The adaptation is a reuse of their past work on a kernel Model Checking (see Section 2.7).

However, the use of virtual machine has a severe limitation. Many virtualizations do not allow direct access to the hardware. Hence, when performing a check similar to EXE, drivers for the hardware cannot be exercised.

On the other hand, when focused to hardware independent parts, EXE showed that it may work. But there was still a problem they had to cope with. Minimal input for checked filesystems is 16 MiB. The solution is to handle the filesystem image symbolically only on demand. I. e. when the kernel asks for a block, it is made symbolic. This allows the tool to run quickly, as relevant metadata are stored only in few blocks.

2.7 Other and Related Work

Static analysis is indeed a wide area. This section provides quick overview of other approaches and techniques. Although the list is still not complete, it tries to add clues for further reading.

Dynamic Testing Dynamic testing was here since the first programs were written. They are fed by random inputs, a kind of black-box testing in [4, 14, 11] or

[20] Either environment, assembly, allocators or other peculiar issues arose.

[21] `user-mode-linux.sourceforge.net`.

unit-tested. Unit testing is a clear opposite. A programmer tries to use the knowledge of what the code does and writes tests for his implementation to test as many paths as possible and for each unit/module.

Sometimes, programmers instrument their code with conditional (debug-only) fault injections. For example allocators return an error instead of correctly allocated memory even if there is enough of memory. It helps to cover corner cases in the code.

Some code have specification created during the development cycle. They may reveal many bugs when converted to grammars and utilized to generate inputs like in [10, 22, 15].

Model Checking Opposing to checkers, model checking (MC) tries to *prove*, that checked programs satisfy a particular property according to specification. We are not trying to achieve such goals, because MC (a) suffers from state space explosion and hence lasts long often; (b) need precise models. On the other hand, it reports no false positive errors if the model is precise enough.

A comprehensive overview of Microsoft's MC tools with further links and technical details about SLAM and SDV is in [2]. It has found many errors in Windows Device Drivers. A SLAM-based tool that is supplied with lazy abstraction is BLAST [3]. A model there is very coarse at the beginning and is narrowed even after it is necessary and only at places which needs that. It lowers time and space spent on run.

There is a plenty of abstraction-based model checkers, the two were picked because of summary papers provided. A reader will be directed for further reading there.

Instead of direct MC, some tools generate a source code from temporal model and original sources provided. The generated code representing a part of the model is compiled and run. As a representative of this category, SPIN [20] is chosen, presently a widely used formal verification tool with many MC improvements and optimizations.

Kernel and Model Checking Similarly to Section 2.6.1, Engler et al. checked three filesystem implementations again, this time with a use of MC in [34]. Linux TCP/IP stack was under MC algorithms inspection in [23]. Both works are based upon a [24].

There is also a thesis [32], in which Thomas Witkowski uses *Calculus of communicating systems* (CCS) for modeling a kernel behaviour in some special cases. There are then properties specified in LTL logic proved by a solver on the top of CCSs. He implemented it in the DDVERIFY tool and was successful in checking whether resources such as interrupts and queued works are properly initialized before their use.

3 Dissertation Thesis Intent

Since I plan to finish my doctoral studies in the *Computer Systems and Technologies* programme, the main output of the thesis will be a fully working open-source tool performing the bug-finding on the Linux kernel. As far as I know, there is no such tool that would perform automatically, without a need of long establishing of a testbed like with EXE, the following steps: (a) take kernel source code with build system as input, (b) compile and link all the sources into portable and machine independent intermediate language (LLVM) without user intervention and (c) apply analyses that have low false positive rate.

In addition, there are problems yet to be solved. In spite of the generated kernel image contains all functions called from the inside, some of them are in assembly, inherited from their sources. The thesis will solve this problem, at least for selected functions. Optimized functions like `memcpy` and `strcpy` must be replaced by non-optimized ones (written in C) for easier analysis – the more code is in the LLVM code the less assembly has to be specially handled by the algorithm.

Further, some primitive inlined assembly should be handled as well. As a result, calls like `put_user`, `get_user` should be recognized either on an assembly level or handled even in the preprocessor. They make sure that user's pointer is correct. That means the tool cannot ignore them and instead should check if they are used on a user- and kernel-space boundary properly and every time.

Some of the problems may be tackled down by known solutions already. STANSE contains build system support, it need to be adjusted to generate LLVM though. As well as compilation, the configuration present in STANSE may be adopted. Additionally utilities to support asserted code, instrumentation (if needed), specification of specially handled functions like `put_user` and definition of what to check in them all need to be added. Some of well-known to-check patterns will be already predefined for the ease of use and demonstration of what the tool can do.

Code Analysis We have not considered how to handle the intermediate LLVM code further. If the SE is the technique actually used in the tool, KLEE or better the Marek Trtík's dissertation may be used as executor and error miner. Marek plans to come up with a faster algorithm than in KLEE. KLEE here is considered only as a fallback due to its speed, as it is not primarily designed for finding specific bugs, but rather for a high code coverage (resulting in bug-finding in the end anyway).

No matter what approach on the top of LLVM is used, based on the fact, that the functions in the kernel might be called from different contexts and as a response to different kind of events, majority of the functions might run concurrently. I am going to investigate possibilities of handling concurrent threads and find properties that may be checked there.

In principle, even with a use of SE, there is still a big gap in solving false positives. The thesis will aim at pruning them and consider and extend techniques used, for example, in XGCC, because they allow lowering false positive rate in the order of magnitude.

A possible approach to reduce false positives, that the thesis will take into account, may be also to compare two consequent results from two different versions

of the kernel. I. e. when the initial version is checked and false positives marked by user, it is highly likely, that the later version contains the same false positives. Ability for matching errors between two versions (lines and function names may change) may prune more false positives and would not bother user with the same reports repeatedly. I think this was never implemented yet.

Collaboration The tool development will not be a task for singles. An additional help from bachelor/master's students will be needed. Hence particular parts may lead to theses under my leadership. A good example is investigation whether machine learning can be used for recognition of false positives in checker results.

Further, because of our limited knowledge of the Linux kernel (it is a wide area), the tool will be discussed with kernel developers and possibly improved by their ideas. Specifically, it may help even in the enterprise kernel development. It means a cooperation with commercial subjects on improving the tool and checking their products before they are released.

And last but not least, if there appear groups dealing with a similar problem, the cooperation will be helpful reciprocally.

Summary The result of the thesis will be a tool based on portable parts that contains algorithms for finding kernel-specific properties such as watching concurrent threads (after studying their behaviour and patterns of their creation) for possible deadlocks while reducing false positives. Despite some algorithms for that are already known, due to kernel specifics (e. g. much code in assembly), they cannot be all applied and hence the thesis must come up with new ones.

3.1 Schedule

I plan to finish my Ph.D. studies by dissertation defence in the term spring 2012, i. e. after 8 terms of doctoral studies. Here comes the expected schedule of my work during the following terms.

- spring 2010 – STANSE papers submission, dissertation topic defense, examination; for dissertation, working code processing, decision of algorithms to use. Specifically decide whether the known algorithms may be used on kernel at all and think about replacements if not.
- autumn 2010 – Implementation of new and adopted algorithms on the top of the intermediate code walkers/executors provided. It should be clear now whether the algorithms are appropriate or not and replace them eventually. It includes that e. g. the behaviour of threads is known already.
- spring 2011 – The tool is able to work with patterns and find simple violations. I. e. it is proven that the chosen algorithms may work. False positives start to occur, they are studied and contemplated algorithms directed towards them.
- autumn 2011, spring 2012 – Fully working automatic bug-finding tool executable on the kernel, capable of finding errors. Dissertation write-up and defense.

References

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. *Lecture Notes in Computer Science*, 4963:367, 2008.

[2] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. *Lecture notes in computer science*, 2999:1–20, 2004.

[3] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker B last. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5):505–525, 2007.

[4] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Journal of Research and Development*, 22(3):229, 1983.

[5] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT—a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the international conference on Reliable software table of contents*, pages 234–245. ACM New York, NY, USA, 1975.

[6] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. 2006.

[7] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. Engler. EXE: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[8] J. Corbet. vmsplice(): the making of a local root exploit. *LWN.net*, 2008. `http://lwn.net/Articles/268783/`.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM New York, NY, USA, 1977.

[10] A. G. Duncan and J. S. Hutchison. Using attribute grammars to test designs and implementations. In *5th Int. Conf. on Software Engineering*, pages 170–178. San Diego, CA, March 1981.

[11] J. W. Duran and S. C. Ntafos. Evaluation of random testing. *IEEE transactions on Software Engineering*, 10(4):438–444, 1984.

[12] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, pages 1–1. USENIX Association Berkeley, CA, USA, 2000.

[13] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. *ACM SIGSOFT Software Engineering Notes*, 19(5):87–96, 1994.

[14] J. E. Forrester and B. P. Miller. An empirical study of the robustness of Windows NT applications using random testing.

[15] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. *ACM SIGPLAN Notices*, 43(6):206–215, 2008.

[16] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, page 223. ACM, 2005.

[17] P. Godefroid, M. Y. Levin, D. Molnar, et al. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*. Citeseer, 2008.

[18] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses, 2002.

[19] G. Holzmann. UNO: static source code checking for user-defined properties. Citeseer.

[20] G. Holzmann et al. The model checker SPIN. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[21] E. Larson and T. Austin. High coverage detection of input-related security faults. *Ann Arbor*, 1001:48105.

[22] P. M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55, 1990.

[23] M. Musuvathi and D. Engler. Model checking large network protocol implementations. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation-Volume 1*, page 12. USENIX Association, 2004.

[24] M. Musuvathi, D. Y. W. Park, A. Chou, D. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code.

[25] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. *Lecture Notes in Computer Science*, pages 213–228, 2002.

[26] T. Ormandy. Another kernel null pointer vulnerability. *LWN.net*, 2009. `http://lwn.net/Articles/347006/`.

[27] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[28] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, page 272. ACM, 2005.

[29] B. Spengler. Linux 2.6.30 exploit posted. *LWN.net*, 2009. `http://lwn.net/Articles/341773/`.

[30] R. M. Stallman and the Developer Community. *Using the GNU Compiler Collection*. GNU Press, 2008. For gcc version 4.4.2.

[31] N. Tillmann and J. de Halleux. Pex–white box test generation for. net. *Lecture Notes in Computer Science*, 4966:134–153, 2008.

[32] T. Witkowski. Formal verification of Linux device drivers. *Master's thesis, Dresden University of Technology*, 2007.

[33] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 243–257. Citeseer, 2006.

[34] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems*, 24(4):393–423, 2006.

[35] CLANG. `http://clang.llvm.org/`.

[36] COVERITY PREVENT. `http://www.coverity.com/products/`.

[37] FINDBUGS. `http://findbugs.sourceforge.net/`.

[38] KLOCWORK. `http://www.klocwork.com/products/`.

[39] LLVM. `http://llvm.org/`.

[40] SPARSE. `http://www.kernel.org/pub/software/devel/sparse/`.

4 Summary of Study Results

During the first year of doctoral study I studied papers with results of past researches in the area of code analysis and testing. Also I read through some literature about static analysis and compilers.

At ITI, we developed Stanse, a static analysis tool written in Java. We learned what sort of bugs programmers do, and overall problems bound to the static analysis. It helped out to my dissertation, because some of the principles may be reused and some implemented from scratch, but better, with the knowledge we gained.

We are currently in the process of submission of two papers about Stanse.

Stanse Contribution Now let me summarize my contributions to Stanse as was noted in Section 2.2. I wrote major part of code processing from preprocessor written in Perl to UnitManager which throws unneeded parsed structures away from memory. Further I moved debugging code for reachability away from parser to ReachabilityChecker to demonstrate how easy is to write a checker.

I prepared the system for being run on the Linux kernel. It includes specifying properties, lower false positives (by tuning properties and involving false positive detectors), implementing a workaround to disable macro expansion, added some timeouts to algorithms to limit the analysis and finally I consulted the found errors with kernel developers and sent fixes.

I am also the author of a web interface, which serves for an easy access to errors for developers. It turns out to be an eligible interface when referring to an error in emails.

In sum, the value of all above will be appreciated when implementing the tool in the scope of dissertation, because we more-or-less know where the analysis can lead.

Error Types Namely, the analysis revealed us many kind of errors both in kernel and userspace. We have found omitted unlocks in fail paths and also in standard paths under certain circumstances like writing to a file twice in a second. There were many missing pointer checks against `NULL` when allocating memory, forgotten resource releases for TTY, PCI and IRQs. Potential deadlocks when the process tried to sleep in atomic contexts were found too. Finally, we found out that compilers are not as good in finding unreachable statements, Stanse does better in some cases. See Section 2.2 to compare implemented features.

A list of bugs found by Stanse is at `http://stanse.fi.muni.cz/bugs.html`. The list is incomplete, we still find bugs in latest kernels, because there is a constant income of them.

SE Investigation I already read through LLVM manuals and tested it on a real code. I compiled major parts of the Linux kernel into LLVM code and linked together. However, as I expressed earlier in this text (in Section 2.5.1), some problems bound to the kernel parsing need to be solved yet.