Practical Automated Bug Finding

Jonas Wagner
School of Computer and Communication Sciences
École polytechnique fédérale de Lausanne (EPFL), Switzerland
jonas.wagner@epfl.ch

***Abstract*— This work presents three state-of-the-art tools that found hundreds of bugs in real-world software, namely the SLAM static driver verifier, the SAGE whitebox fuzzer and the S^{2}E engine for selective symbolic execution. I show that despite their different algorithms and architecture, the tools face similar challenges and solve them in similar ways. The present document describes three ideas that build upon these similarities to make program analysis more flexible, performant and scalable: the combination of different analysis methodologies, the use of compiler techniques, and machine learning for modular analysis.**

***Index Terms*—Testing, Symbolic Execution, Verification, Software Model Checking**

I. Introduction

In a world that relies ever more on software, the quality and dependability of software is of paramount importance. Yet writing high-quality software is challenging. Bugs are frequent and expensive: a 2002 NIST study estimates that they cost 0.6% of the US GDP [10]. The same study suggests that better testing could significantly reduce this number.

In this work I present the state of the art in automated tools that help developers find bugs in their software. I focus on practical tools that scale to real-world programs and whose influence spreads well beyond the research groups that originally developed them.

To be successful, such tools need to be based on solid foundations from formal methods research. At the same time, systems techniques enable them to be distributed and resilient to failures. Bug finding tools rely on components such as constraint solvers. How closely such components are integrated, and how tolerant the system is if they fail, are important design choices that affect the performance and applicability of bug finding tools. Research on bug finding tools is also related to software engineering. These tools often need to handle heterogeneous software, built using a custom process involving multiple programming languages, and running in a complex environment. Ultimately, the tools' output must be understandable and helpful to the developers.

Out of this diversity, the present work presents three points in the design space of automated bug finding tools. The SLAM project [1] analyzes Windows device drivers using predicate abstraction and counter-example guided abstraction refinement (CEGAR). Thanks to its focus on a specific class of programs and its elaborate environment model, it can prove correctness of drivers against many safety properties in a few hours. In contrast, the SAGE whitebox fuzz tester tests large, general-purpose programs for weeks at a time [3]. Finally, the S^{2}E tool is a flexible framework to find bugs in binaries that depend on complex environments [4].

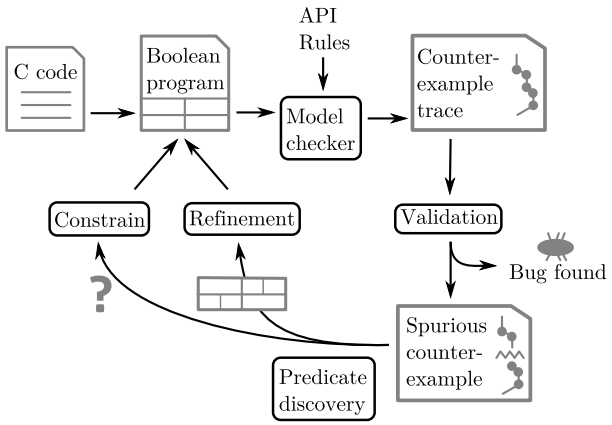

Fig. 1. An overview of the SLAM verification process

II. Background

A. The SLAM project

Device drivers are in many ways an ideal target for software verification. They are relatively small (< 100 kloc) and use a

Proposal submitted to committee: August 20, 2012; Candidacy exam date: August 27, 2012; Candidacy exam committee: Willy Zwaenepoel, George Candea, Viktor Kuncak.

This research plan has been approved:

Date: _____

Doctoral candidate: _____
(name and signature)

Thesis director: _____
(name and signature)

Thesis co-director: _____
(if applicable) (name and signature)

Doct. prog. director: _____
(R. Urbanke) (signature)

well-defined (though complex) API. At the same time, drivers are critical parts of an operating system and are the root of many crashes and hangs. These are the motivations of the SLAM project.

SLAM uses a form of CEGAR to verify that device drivers comply to a set of API usage rules. Its high-level structure is shown in Figure 1. It starts by converting the driver's source code into a very abstract *Boolean program* that over-approximates the original driver. The Boolean program is tested by a model checker. This step either determines that the program is correct, or produces a trace of an execution that violates an API rule. This trace is validated to determine whether it represents a real bug in the driver, or occurs because the Boolean program is too abstract. In the latter case, the driver is symbolically executed along the counter-example trace to find a set of constraints that precludes the counter-example from occurring. The Boolean program is refined to keep track of these constraints, and the process repeats.

The Boolean programs that SLAM maintains always over-approximate a driver. This means that if no bug is found, SLAM can proof that the driver is correct with respect to the API usage rules. These rules form a specification of the Windows Driver Model (WDM); their completeness determines how many bugs SLAM can find, and their quality influences the number of false positives or useless results (because SLAM can time out if rules are too complex). Because of their importance, the SLAM developers invested significant work in the 470 API usage rules shipped with SLAM [2]. In addition, SLAM contains a handcrafted model of the environment wherein the driver runs.

A number of optimizations make SLAM more efficient. During the refinement phase for example, SLAM uses both forward and backward symbolic execution. The forward phase is run first to compute alias information required by the second step. To make the backward step tractable, symbolic pointer values are concretized and offsets into arrays are discarded.

Because of these and other simplifications, the predicates are not perfectly precise. It sometimes happens that they do not prevent a counter-example from happening again (and again). This lack of progress was a large problem in the first version of SLAM. More recently, the backward symbolic execution step has been enhanced to detect several inconsistencies in a single counter-example. Only if none of these yield good predicates, a last effort is made to increase the precision of the Boolean program (the Constrain step in Figure 1).

These sophisticated techniques enable SLAM to make progress despite imprecision, and obtain a useful result for 96.7% of the WDM drivers in its test suite. The false positive rate is at 0.4%. This number improved compared to past versions mainly because of the increased quality of API rules.

B. Whitebox Fuzzing with SAGE

In contrast to the SLAM project (that uses extensive domain-specific knowledge for quick, high-precision verification of a specific class of programs), SAGE is a general-purpose white-box fuzzer that makes few assumptions about the program under test. It can test any file-reading program (e. g. a media player) and is independent of build processes or closed-source libraries because it works with X86 assembly.

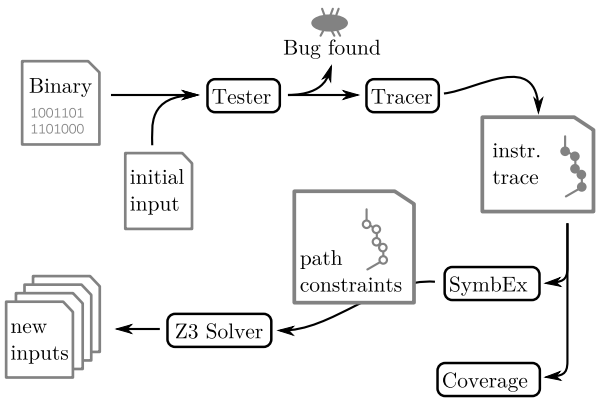

Fig. 2. Overview of the different components of SAGE

Figure 2 shows the different steps of the SAGE framework. SAGE starts by running the program under test with some initial input (e. g. a valid media file). The program is instrumented using Microsoft's AppVerifier tool, so that memory errors and other unwanted behaviors are detected. If this step succeeds, the program is run again in the *Tracer*. The output of this step is an execution trace comprising all X86 instructions executed, with information about memory allocations, system call return values, etc.

This trace is executed again, this time with the program's input replaced by symbolic values. The symbolic execution step keeps track of the symbolic expressions that the program builds from its inputs. It also computes the constraints that the input has to satisfy so that the program follows the given trace. This so-called *path constraint* is a conjunction of clauses, one for each branch in the trace.

One by one, each clause is negated and the resulting path constraint (up to and including the negated clause) passed to the Z3 constraint solver. If it determines the constraint to be satisfiable, Z3 generates a new input that will lead to a different path through the program, potentially uncovering previously untested code. This way, each symbolic execution step leads to a whole set of new inputs that cause the program to follow a prefix of the trace, but take a new path at the point where a clause was negated.

Many auxiliary systems not shown in Figure 2 are important components of SAGE. Most notably, the *JobCenter* system manages the machines on which SAGE runs. It can set up SAGE on a number of machines using a centralized configuration, attribute jobs to SAGE instances, handle software updates, and recover from machine reboots and other errors. This enables SAGE to work on hundreds of machines in parallel, with individual runs often lasting for weeks at a time.

Another important component is the *Sagan* logging platform. It keeps a centralized record of every SAGE run and allows developers to collect statistics, analyze test coverage, investigate errors or re-execute a SAGE run of interest. Data from Sagan led to important insights, e. g. in the nature of constraints generated by SAGE. About 99% of all constraints

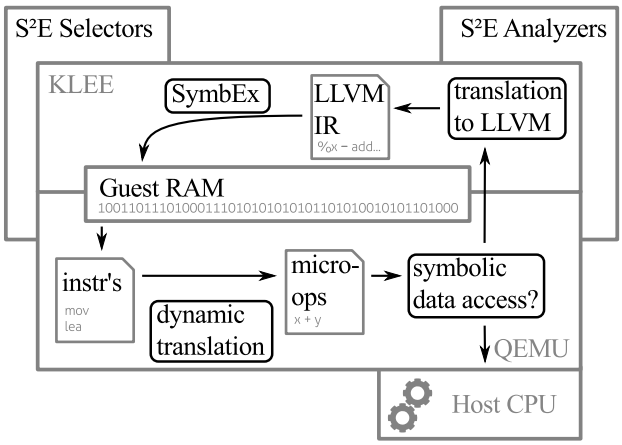

Fig. 3. Schema of the S^{2}E symbolic execution framework

generated by symbolic execution are solved by Z3 in less than a second. Moreover, those constraints that take longer to solve are usually unsatisfiable and hence do not lead to new test inputs. The abundance of such data gives rise to many data-driven optimizations: SAGE now uses short timeouts for solver queries and often concretizes symbolic values, e. g. when non-linear operations, exotic operations such as SIMD instructions, or symbolic pointers would lead to complex constraints.

It is remarkable that SAGE is tolerant to many unsound approximations and simplifications performed during symbolic execution or constraint generation. These imprecisions can lead to *divergences*. That is, the program does not follow the path that was predicted at constraint solving time. Still, each input represents a new test, and SAGE gracefully degrades to a (slow) random fuzz tester if the constraints become too imprecise or difficult to handle.

This shows that unlike SLAM that *verifies* drivers, the focus of SAGE is on test generation to find bugs. It does so very successfully: approximately one third of all file fuzzing bugs in Windows 7 were found using SAGE. Furthermore, those were hard-to find bugs that had evaded other techniques such as unit testing or random testing.

C. Selective Symbolic Execution with S^{2}E

The S^{2}E framework shares many similarities with SAGE. Both are essentially bug-finding tools that enable symbolic execution of binaries. Yet there are unique aspects that set S^{2}E apart from SAGE.

Architecturally, S^{2}E is the composition of the virtual machine QEMU and the symbolic execution tool KLEE (see Figure 3). This combination allows to symbolically execute arbitrary code running inside a virtual machine. Around this core, a number of selector and analyzer plugins are provided. The selectors allow to focus the analysis on the code of interest, whereas analyzers extract a variety of information (e. g. execution traces or cache miss statistics) from the program under test.

This architecture provides a number of benefits. First, it makes S^{2}E even more universal than SLAM; any program running on an X86 or ARM processor can be tested in principle. S^{2}E is fast enough to run a program in a complex environment. The parts of the system that do not touch symbolic data are run concretely and incur approximately 6× overhead with respect to QEMU. Code that manipulates symbolic data undergoes a more costly translation to LLVM and is executed via KLEE, at roughly 78× overhead. For comparison, executing a trace symbolically using SAGE takes about 700× longer than running the program natively.

QEMU uses dynamic binary translation to convert a guest program into code for the host architecture. It uses a relatively small set of micro-operations as intermediate representation. This means that S^{2}E does not have to deal directly with the entire complexity of X86 assembly, but only needs to support a limited number of well-specified operations. This leads to fewer approximations than SAGE and reduced implementation complexity.

The main difference in the symbolic execution step is that KLEE (and thus S^{2}E) is a state-based system. Instead of individual concrete+symbolic executions, it builds the entire execution tree of the system in memory. A sophisticated memory management using copy-on-write makes this efficient. This enables adaptive search heuristics and precise selection algorithms that decide which state to explore next. On the other hand, concrete values are no longer automatically available for each symbolic expression. This makes memory accesses more expensive (see Section III-B).

The ability to selectively apply symbolic execution to code of interest is a major novelty of S^{2}E. It makes it possible to perform full multi-path symbolic execution of the program under test, while "fast-forwarding" through the program's environment. This reduces the effective size of the tested program and is an important strategy against path explosion that plagues all symbolic execution tools.

On the boundary between the program and its environment, values need to be converted from the symbolic to the concrete domain and back. The conversion happens according to a consistency model. S^{2}E comes with several predefined models that allow users to reason about the trade-offs between soundness, efficiency and completeness of the analysis.

The flexibility and generality of S^{2}E allowed it to be used successfully for testing and reverse-engineering device drivers, checking file-system utilities, performance testing and other projects.

III. Challenges

A. Testing Programs with Complex Environments

Interesting programs usually interact with large environments in complex ways: applications read data from files or the network, drivers communicate with hardware and the operating system, and multiple programs may solve a problem in a distributed fashion. If the environment is taken into account, the total size of the code being tested increases by orders of magnitude. This section discusses the strategies adopted by SLAM, SAGE and S^{2}E to combat this problem.

The scope of the SLAM project was carefully chosen and limited for reasons discussed in Section II-A. This means that the drivers analyzed by SLAM all interact with their

environment through the same well-defined API, namely the Windows Driver Model (WDM). The large costs incurred by the SLAM project for modelling this API and formally specifying its usage rules can thus be amortized over the many drivers that benefit from these models.

The manually crafted API rules are highly optimized for efficient analysis. They also specify complex properties that catch much more subtle errors than the crashes found by SLAM or S^{2}E. Furthermore, SLAM's API rules are written in close collaboration with experts and designers of the API itself. In the newest version of SLAM, this influence was even mutual: the Kernel-mode Driver Framework (KMDF, a new driver model for Windows 7) was developed in parallel with the corresponding SLAM rules. If some function was hard to formalize, it indicated that the API was too complex, and caused the function to be replaced by simpler equivalents.

The SAGE tool does not have the luxury of a well-defined and limited testing target. Instead, it solves the environment problem by working on binaries and scaling to really large programs. It scales so well that it can handle the additional complexity incurred by program instrumentation (with program checkers such as AppVerifier). This strategy enables SAGE to still detect some higher-level errors such as illegal memory accesses or incorrect use of some OS functions.

The environment problem is handled in a principled way in S^{2}E thanks to its consistency models. These specify exactly what happens when data crosses the boundary between the tested program and its environment. For example, when a program makes a system call, the call's parameters may be concretized. Upon return, the return value of the call can be replaced again by a symbolic expression.

Concretizing values means losing completeness (some bugs might not be found because of the particular values chosen). On the other hand, introducing symbolic data may be unsound in that it introduces execution paths which would otherwise be infeasible. Depending on the consistency model, S^{2}E relies on lightweight annotations to know where it can safely introduce symbolic data (e. g. the set of possible return values of a system call). Being able to reason about completeness and soundness in a principled way is also beneficial for applications (such as reverse engineering) that do not require soundness.

B. Memory models

Most reasonably complex programs use dynamically allocated heap memory. This is very challenging for software analysis. Data structures on the heap may be unbounded. Pointers can alias each other. If pointer values are symbolic, any memory location can in principle be accessed and modified through them. The three presented tools tackle these challenges in a variety of ways.

As discussed in Section II-A, SLAM uses forward symbolic execution to compute alias information required by its predicate discovery step. Specifically, this analysis determines for every pointer which memory location is pointed to. To my knowledge, SLAM does not support symbolic pointers targeting multiple locations. In this case, an arbitrary location is chosen. Also, all memory regions are mapped to their base address; in other words, any array access is treated as if it modified the first array element.

These simplifications make the predicate discovery tractable. The prize to pay is that they sometimes impede progress. SLAM has other means to recover precision in some of these cases (see Section II-A).

The first versions of SAGE shared SLAM's strategy of concretizing symbolic pointer values whenever memory is accessed through them. This works well because (1) the concrete value is already available and (2) SAGE is not harmed much by imprecision and divergences.

As of 2009, SAGE has a memory model that is fully precise, yet incurs only a small overhead compared to concrete addresses [6]. For every memory access through a symbolic pointer, SAGE uses the corresponding concrete address to find the memory region which is being accessed[1]. It then adds a constraint that forces the pointer to lie inside the bounds of that region. This constraint is part of the path constraint. Like any other clause, it can be negated during input generation. Hence SAGE will eventually find inputs that trigger out-of-bound memory accesses (if they are feasible).

The process is more complicated if the program under test contains arrays of arrays. In this case, a symbolic pointer can be dereferenced multiple times. Such an operation may access a set of memory regions. Thus SAGE adds a disjunctive constraint that forces the pointer to point to a valid address inside *any* of the regions in consideration.

The memory model in KLEE is very similar, except that multiple dereferences are not handled separately. This means that KLEE needs more solver queries to resolve a memory access: A first query checks whether the pointer *must* lie inside the bounds of the target memory region. In the case of multiple dereferences, this query fails, and KLEE needs another query per possible target region. In contrast, SAGE replaces these multiple queries by a single disjunctive query. In addition, KLEE requires one additional query for every concrete address needed in this computation, because concrete values are not readily available for a state-based symbolic execution engine.

S^{2}E, unlike SAGE, does not have the capability to infer the memory layout of the programs under test. Instead, S^{2}E partitions the virtual machine memory in fixed-sized segments. The size of these segments (which defaults to 128 bytes) introduces an interesting trade-off. Large segments lead to many accesses per segment. The constraint solver considers all possible aliasing situations between those accesses; its task is very complex. If memory segments are smaller, accesses that otherwise would touch a single region could now fall into multiple segments. S^{2}E creates a new state for every potential segment. Hence, smaller segments shift work from the constraint solver to the symbolic execution engine. Such trade-offs arise frequently and are described in more detail in Section III-C.

[1]The range of memory allocated in a single operation (e. g. using `malloc`) is called a memory region or memory object

C. Efficient Use of Constraint Solvers

The authors of SAGE claim that "for whitebox fuzzing, *the art of constraint generation* is as important as *the art of constraint solving*." [3] They employ a number of strategies and optimizations to use the constraint solver as efficiently as possible.

Among these are several sound optimizations such as caching to identify common subexpressions, or pruning to remove irrelevant clauses before the query is sent to the solver. However, some of SAGE's optimizations are unsound and may influence the result of the analysis. For example, constraint subsumption eliminates constraints that are implied by other constraints from the same program location. For a loop of the form `for (int i = 0; i < λ; ++i)`, the constraints $\lambda > 1$, $\lambda > 2$, ... would be eliminated and only the last constraint kept. This means that on every SAGE run, only one input with a different number of loop iterations is generated (instead of inputs for every possible number of iterations).

A characteristic of SAGE is that symbolic execution has a large overhead. Therefore, much has been done to fully exploit each execution. For example, SAGE generates a whole range of new inputs from a single trace (one input for every feasible alternative branch). Another example is SAGE's use of disjunctive constraints to handle multiple pointer dereferences. Also, experiments have shown that function summaries can create considerable speedups because they replace all constraints from a function body by a single disjunctive constraint, and remove the need to symbolically execute the function body [7].

These approaches move work from the symbolic execution engine to the constraint solver in the form of more complex queries. These trade-offs have been explored for KLEE [9]. The authors were able to explore several orders of magnitude more states in a given time compared to vanilla KLEE. They provide heuristics to estimate under what conditions more complex disjunctive queries will lead to faster program exploration.

In the context of SLAM, a technique called *coarse-grained abstraction* leads to better utilization of the constraint solver while increasing precision. SLAM works with Boolean programs that keep track of boolean predicates. Early versions of SLAM computed how each of these predicates is affected by individual instructions in the original program. More recently, SLAM computes these predicate transformers for entire basic blocks. That is, it computes the joint effect of a group of instructions on a predicate. This makes the evaluation of predicate transformers more costly, but also keeps track of more relationships between predicates.

Finally, all three bug finding systems use strategies to minimize the number of queries to the constraint solver. For example, SLAM employs specialized algorithms using binary decision diagrams for predicate abstraction. SAGE keeps concrete values in parallel to symbolic values. S^2E caches solver queries and counter-examples to eliminate duplicated queries.

IV. Research Proposal

In this section, I use insights from SLAM, S^2E and SAGE to develop three ideas that improve the scalability and practicality of bug finding tools.

A. Combining Multiple Analysis Techniques

Each of the tools presented in this work is unique and has its individual approach to program verification or testing. The artifacts that they create and manipulate are rather different: SLAM builds Boolean programs that over-approximate the original program, S^2E builds an execution tree, and SAGE handles individual execution traces. Yet, all of these are code fragments that correspond to partial unrollings of the original program.

I argue that a unified intermediate representation could allow these tools (and many others) to work together. In current work with Vova Kuznetsov, Johannes Kinder and George Candea, we develop the Klein program analyzer that reasons about program fragments in the LLVM intermediate representation.

Klein is first of all a bounded model checker working on a loop-free, finite unrolling of the program under test. It generates expressions for arbitrary program values (e. g. values used in `assert` statements, or values expressing reachability of a location) and dispatches them to a constraint solver. However, when such queries become too complex, Klein can apply a technique called *state splitting* to analyze smaller groups of paths at a time, akin to symbolic execution.

In the context of a project for the *Synthesis, Analysis and Verification* course at EPFL, I extended Klein and argued using a prototype that Klein's intermediate representation can encode abstract reachability trees. In other words, predicate abstraction engines such as SLAM could be implemented in the Klein framework.

This allows a framework where multiple tools analyze the same program and collaboratively manipulate an intermediate, partially unrolled and abstracted version of the program. The framework could choose the best tool for each small task, e. g. use predicates from predicate abstraction to prove the safety of an unbounded loop, while using whitebox fuzzing to find corner-case bugs quickly. Since the tools collaboratively manipulate LLVM, each intermediate result is a valid program that can be transformed, compiled to native code, fuzz-tested and so on. Hence, the distinction between computer programs and mathematical objects becomes smaller. Such a framework extends ideas such as Godefroid et al's may-must analysis [8].

The LLVM intermediate representation is in many ways an ideal choice to exchange results between multiple tools. The language is designed to be easy to analyze. It is much less complex than X86 assembly, yet provides some generality because many source languages can be compiled to LLVM. The LLVM IR preserves much high-level information (such as types) present in the original source. Many LLVM operations are directly supported by constraint solvers for bit-vectors. The language can also be extended with metadata and custom functions to express information such as loop bounds, `assume` statements, etc.

B. Leveraging Compiler Techniques

Klein's choice of LLVM as an intermediate representation opens up a range of optimization opportunities. Efficient implementations exist for techniques such as constant propagation or global value numbering in LLVM code. These

can be used to significantly simplify (or even avoid) solver queries. Many of these are transformations of some code into equivalent (but more efficient) code. Hence they can be applied without hesitation in a variety of settings.

We believe that these ideas could be taken even further to create an optimizing compiler for program analysis. Assigning parts of the program analysis to the compiler means that the results are available for all subsequent steps. The compiler has a very detailed high-level view of the program's source (as opposed to, say, tools working with binaries) and can factor all this information into the analysis.

The following analyses and transformations are examples of what such a compiler could do:

- *Aggressive loop unswitching, loop unrolling or function inlining*. Whereas traditional compilers are limited in terms of the size of the resulting programs, a compiler optimizing for program analysis can focus solely on the performance of the analysis.
- *Emitting simplified code*. Fast code is not always easiest to analyze. For instance, `switch` statements are often efficiently implemented using jump tables. The symbolic memory accesses due to those tables are challenging for analysis tools; it would be better to use a series of nested `if-then-else` statements.
- *Search heuristics*. Using its complete view of the call-flow graph, a compiler could annotate branch instructions with information about how much code is statically reachable for a given condition value. This could make exploration more guided, especially for tools working on binaries where the call-flow graph is not readily available.
- *Annotations*. A compiler could embed various data into a program for the analysis engine's perusal. Examples are value ranges for program variables, array sizes and alias information for pointers, or bounds on the number of loop iterations.

C. Modular Analysis

I have shown that all three systems presented in this work have to balance soundness, completeness and performance of the analysis against each other. Despite these compromises, the maximum size of the programs that can be analyzed is still limited because program complexity increases exponentially with a program's size. We believe that there is no fundamental way around this. Instead, we would like to build tools that can analyze parts of programs in isolation. The challenge is how to abstract away the rest while retaining enough precision for the results to be useful.

One idea that is being explored in the Klein framework is the use of machine learning to find function preconditions. Data collected from multiple program runs is mined for likely invariants. These are then used as preconditions to set up an environment in which the function can be tested. Similarly, external functions called by the function under test can be replaced by their postconditions (which are obtained in a similarly way to preconditions).

Modular analysis also opens avenues for systems techniques. Many functions could for example be analyzed in parallel on distributed clusters.

Modular systems such as SATURN have been successful in the past [5]. However, they relied on hand-crafted specialized abstractions instead of automatically generated pre- and post-conditions. Our ideas are still in early stage of development, and only experiments will show whether bugs can be found this way. In the end, the cost of a bug report is the only quality metric of practical automated bug-finding tools.

V. Conclusions

This work describes three practical tools that find bugs in software. These tools combine theoretical algorithms such as counter-example guided abstraction refinement and symbolic execution with systems techniques to create a scalable, practical analysis.

I have shown how the common aspects of these systems could be leveraged to build a framework that combines different program analysis techniques. I argue that a compiler intermediate representation is an ideal common language to exchange results between multiple tools. I give examples of other aspects of program analysis that can benefit from compiler technology. Finally, I outline an idea for modular analysis that promises to make the analysis of large programs tractable.

Acknowledgements

I would like to thank my colleagues at DSLab and the Klein team in particular for providing a work environment in which the ideas presented in this paper can thrive.

References

[1] Ball, T., Bounimova, E., Kumar, R., and Levin, V. SLAM2: static driver verification with under 4% false alarms. In *10th Intl. Conf. on Formal Methods in Computer-Aided Design* (2010).

[2] Ball, T., Levin, V., and Rajamani, S. K. A decade of software model checking with SLAM. *Communications of the ACM* (2011).

[3] Bounimova, E., Godefroid, P., and Molnar, D. Billions and billions of constraints: Whitebox fuzz testing in production. Tech. rep., Microsoft Research, 2012.

[4] Chipounov, V., Kuznetsov, V., and Candea, G. S2E: A platform for in-vivo multi-path analysis of software systems. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2011).

[5] Dillig, I., Dillig, T., and Aiken, A. Sound, complete and scalable path-sensitive analysis. In *Intl. Conf. on Programming Language Design and Implem.* (2008).

[6] Elkarablieh, B., Godefroid, P., and Levin, M. Y. Precise pointer reasoning for dynamic test generation. In *Intl. Symp. on Software Testing and Analysis* (2009).

[7] Godefroid, P. Compositional dynamic test generation. In *Symp. on Principles of Programming Languages* (2007).

[8] Godefroid, P., Nori, A., Rajamani, S., and Tetali, S. Compositional may-must program analysis: unleashing the power of alternation. In *Symp. on Principles of Programming Languages* (2010).

[9] Kuznetsov, V., Kinder, J., Bucur, S., and Candea, G. Efficient state merging in symbolic execution. In *Intl. Conf. on Programming Language Design and Implem.* (2012).

[10] National Institute of Standards and Technology, May 2002. The economic impacts of inadequate infrastructure for software testing.