Selective Symbolic Execution

Analysis of user space binaries using the S^{2}E platform

Björn Kirschner <kirschne@in.tum.de>
Technische Universität München (TUM)

May 22, 2015

Abstract

Bla blablablal

1 Introduction

Frequently developers need to understand software systems. In a very simple case they just analyse their own code or test the interaction of own programs with other components or with the surrounding environment in general. Testing self-written programs conceptually permits the application of the whole arsenal of analysis techniques.

Things become interesting when analysis has to be performed without access to source code or documentation. Scenarios for this situation include the need to check proprietary third party software for interoperability on existing servers, performance, unwanted side effects, and much more. Security-critical environments additionally require reliable guarantees of the benignity of all employed software.

One mighty solution for such system analysis is the S^{2}E platform developed at the Swiss Federal Institute of Technology in Lausanne (EPFL) [9]. Its goal is to provide a tool set for rapid development of analysis tools like performance profilers, bug finders, reverse engineering solutions and the like [11]. S^{2}E combines several key characteristics:

1.) The ability to explore entire *families of execution paths* helps to obtain reliable information about the target system. Abstracting from single-path exploration to sets of execution paths which share specific properties is vital for predictive analyses. This technique can for example prove the non-existence of critical corner cases which might be overlooked by other testing strategies.

2.) *In-vivo analysis*, meaning the analysis of a program within its real-world environment (libraries, kernel, drivers, etc.), facilitates extremely realistic and accurate results.

3.) Working directly on *binaries* further increases the degree of realism in system analyses, as it allows to include closed source modules into the investigation.

What I do is bla... ...justify the choice of S^{2}E as platform for system analysis.

Chapter 2 explains..., then bla, then bla

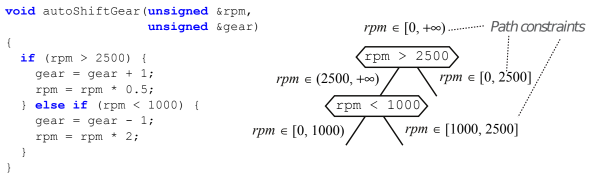
2 Selective Symbolic Execution

Symbolic execution is an advanced analysis technique particularly suited for automated software testing and malware analysis . Instead of concrete input (7, “string”, ...) symbolic execution uses symbolic values (λ, β, ...) when processing code. Assignments in the program path have impacts on these symbolic values. The integer calculation $x = x - 2$, for instance, would update the symbolic expression representing the input x to $\lambda - 2$. Conditional statements (if <condition> then ... else ...) fork program execution into two new paths. Both paths are then constrained by an additional condition, the ‘then’ branch with the if-condition and the ‘else’ branch with the negated if-condition respectively.

ct

ct

Bild > Text



```
void autoShiftGear(unsigned &rpm,
                   unsigned &gear)
{
  if (rpm > 2500) {
    gear = gear + 1;
    rpm = rpm * 0.5;
  } else if (rpm < 1000) {
    gear = gear - 1;
    rpm = rpm * 2;
  }
}
```

Figure 1: Execution tree with path constraints for the symbolic variable *rpm* [7]

Following this procedure results in a tree-like structure of constrained symbolic expressions. A constraint solver can now take all constraints along one execution path as input and find one concrete input (e.g., $\lambda = 5$) which would lead to the program following exactly this path. Such results greatly alleviate writing reproducible test cases [8].

On a technical level, symbolic execution engines save state information (program memory, constraint information, ...) in a custom data structure. Each conditional statement involving symbolic values results in a $fork$ of the program state. The two newly created branches are completely independent and can therefore be processed in parallel.

But the exponential growth of conditionals soon reveals scaling problems of this forking strategy. Despite heavy research on optimisations mitigating this *path explosion* problem only relatively small programs (≅ thousands of lines of code) can be analysed symbolically [8].

Additionally, symbolic execution faces problems when the program under analysis *interacts with its environment*. If it calls a system library like *libc*, in theory the whole system stack including invoked libraries, operating system and drivers would have to be executed symbolically. Considering the path explosion problem mentioned before, the resulting complexity makes such a profound analysis hardly feasible.

One way to solve this problem is to build abstract models of the program's environment . However, due to the complexity of real-world systems, building a model of the entire system is both tedious and unnecessary - the user usually wants to analyse one single program and not the whole system [8].

In order to overcome typical problems of conventional symbolic execution, Chipounov et al. at EPFL developed the concept of **selective symbolic execution** (S^{2}E) [8]. Based on a virtual execution platform S^{2}E gives users the illusion of running the entire system symbolically. By limiting the scope of interest (i.e. which parts of the system should be executed symbolically), users can effectively restrain the path explosion problem. Program code within this defined scope is executed symbolically, whereas out-of-scope parts, which are irrelevant to the analysis, switch to concrete execution.

Definition of the scope of interest (what to execute symbolically) is highly flexible. Users may specify whole executables, code regions, or even single variables to be executed symbolically. Everything else will be treated concretely.

But since on a technical level symbolic and concrete execution are handled very differently - concrete code may run natively while symbolic instructions need to be emulated - switching back and forth these two modes is a major challenge. Hence one of the main contributions of the EPFL team around Chipounov is the transparent and consistent management of switching between symbolic and concrete execution modes.

Figure 2 depicts the **interplay of symbolic and concrete execution**. The illustration is based on a scenario where an application *App* is tested. A function *appFn* invokes the method *libFn* in a library *Lib*, which in turn calls a function *sysFn* in the kernel. Since we suspect a bug in *libFn*, we focus our analysis upon this function. Due to the path explosion problem, symbolically executing the entire system stack is not feasible. Hence only execution inside *libFn* follows this technique.

Concrete → symbolic transition: When execution enters the function *libFn* it has to change from concrete into symbolic domain (grey areas). This is done by replacing concrete parameters in the method call with symbolic variables. The call $libFn(10)$ becomes $libFn(\lambda)$, optionally also with constraints: $libFn(\lambda \leq 15)$.
Besides the symbolic multi-path execution, S^{2}E simultaneously also runs the function with its original concrete arguments. This is necessary in order to return a correct calculation result to *appFn* and thus keep the execution of *App* consistent.

Symbolic → concrete transition: Since the operating system is not focus of this analysis example, S^{2}E has to switch from symbolic to concrete domain when *libFn* calls into the kernel. This is

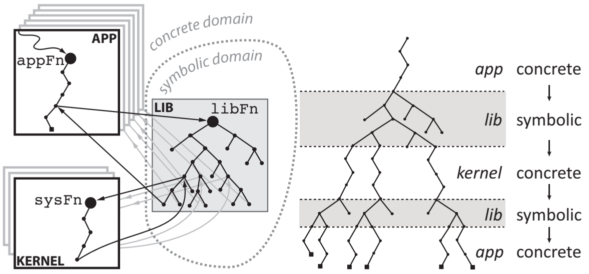


Figure 2: Selective symbolic execution: only paths inside a defined scope of interest (here: a library function *libFn*) are explored symbolically - the rest of the system stack runs concretely [7].

```
void libFn(int x) {
  if (x<5) {
    buf=sysFn(x);
    if (x<0)
      ...
  }
}
```

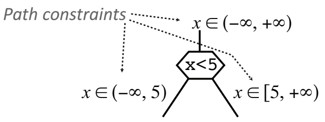


Figure 3: Excerpt from *libFn*'s execution tree [11]

done by randomly picking a concrete value which fulfils all path constraints. If, for instance, the current path is constrained with $x \in]-\infty; 5]$, S^{2}E might choose $x = 4$ and call $sysFn(4)$.
However, when $sysFn(4)$ returns, $libFn$ can no longer make any assumptions about any $x \neq 4$, because the behaviour of $sysFn$ in those cases remains unclear. In order to preserve correctness, a new constraint $x = 4$ has to be added to the path[1]. But imagine $libFn$ being implemented as shown in figure 3; now the 'then' branch of the if-condition $(x < 0)$ can never be reached. Chipounov calls this effect "overconstraining" [7] - it is a result of concretising x when leaving the symbolic domain. S^{2}E tackles the problem by going back in the execution tree and forking an additional sub-tree. The new sub-tree now picks a different concrete value for x which allows to enter the previously unreachable 'then' branch.

bla bla bla

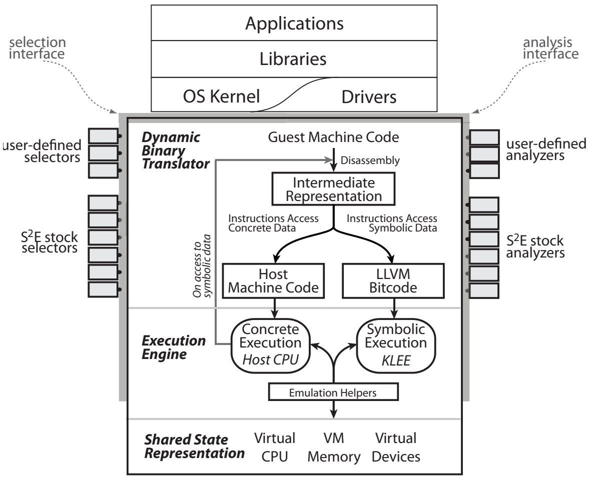


Figure 4: Architecture of the S^{2}E platform [11]

3 The S^{2}E Platform

Based on the concepts described in the previous chapter, Chipounov and his team implemented the S^{2}E platform, an open source framework for writing custom system analysis tools. S^{2}E employs the theoretical concepts of selective symbolic execution by running the system under analysis in a virtual machine and treating code within the scope of interest as symbolic. These symbolic parts are translated into an intermediate representation ( ), while irrelevant instructions are directly passed to the host for native execution.

Technical backbone of S^{2}E are the virtual machine hypervisor QEMU [3, 5], the symbolic execution engine KLEE [1, 6] and the LLVM compiler infrastructure [2, 10]. Figure 4 gives an overview of how these technologies are integrated into the S^{2}E platform. The top of the picture depicts the software stack of the guest system (=the system under analysis), which is managed by QEMU. S^{2}E is not restricted to user land applications, but also allows inspection on deeper levels (e.g., operating system functions).

For easier emulation, QEMU translates machine code of the guest system into an intermediate representation, called *microoperations*. S^{2}E's dynamic binary translator (DBT) splits the resulting microoperations into those that need to be explored symbolically and those which may run concretely. All concrete microoperations are directly converted

[1] Constraints added because of a symbolic → concrete transition are called 'soft constraints'.

into host instructions. Symbolic expressions, on the other hand, are prepared for being executed on the KLEE engine. This requires microoperations to be translated into the LLVM intermediate representation, called LLVM Bitcode in figure 4.

S^{2}E's execution engine, which is an extension to QEMU's execution engine, now manages the operation of the platform. In an endless loop it asks the DBT for new guest code. Depending on the result, instructions can either be run straight on the host system or are fed into the KLEE symbolic execution engine.

In order to keep the mix of symbolic and concrete execution consistent, S^{2}E stores state (VM CPU, memory, ...) centrally, by consolidating QEMU and KLEE data structures and managing them in a single machine state representation.

Users work with S^{2}E by writing selection and analysis plugins or by simply configuring S^{2}E's standard plugins according to their needs. Plugins subscribe to system-wide events (e.g., *onInstrExecution*) and can perform logging/monitoring tasks or even manipulate the system state.

Configuration usually starts with defining what parts of the system to explore symbolically. This can for example be done with S^{2}E's selection plugin *CodeSelector*, which restricts symbolic execution to a specified module or code region.

Standard analysis plugins allow users to find bugs (*WinBugCheck*), monitor memory (*MemoryChecker*), study performance characteristics (*PerformanceProfiler*) and much more (see m?[7], p. 50).

4 Project Idea

The practical part of this project strives to explore privacy issues in a sample binary.

In order to make life easier, many people use little freeware applications on a regular basis. But most of these programs are proprietary and have to be trusted without any knowledge of their functioning. Real malware (Trojan horses, spyware, ...) is usually detected rather quickly by anti-virus software and can often be blocked effectively. However, between unambiguous malware and thoroughly benign software many shades of grey can be found.

This work will focus on the scenario that an application (intentionally or unintentionally) leaks delicate private data without the user's consent or knowledge.

Due to the difficulty of finding a real-world program which shows exactly this desired behaviour and also in general the complexity of real-world applications, the showcase described here bases on a little self-written program.

The software works as follows:

What we do not want is...

All analysis will be done using the S^{2}E platform.

m?

5 Implementation

6 Analysis of S^{2}E Output

7 Outlook

Other cool things one could do...

Apply to real malware...

8 Related Work

Banabic et al. do bla... [4]

9 Conclusion

References

[1] *KLEE*, https://klee.github.io.

[2] *LLVM*, http://llvm.org.

[3] *QEMU*, http://qemu.org.

[4] Radu Banabic, George Candea, and Rachid Guerraoui, *Finding Trojan Message Vulnerabilities in Distributed Systems*, Proceedings of the 19th international conference on Architectural support for programming languages and operating systems, ACM, 2014, pp. 113–126.

[5] Fabrice Bellard, *QEMU, a Fast and Portable Dynamic Translator*, USENIX Annual Technical Conference, FREENIX Track, 2005, pp. 41–46.

[6] Cristian Cadar, Daniel Dunbar, and Dawson R Engler, *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*, OSDI, vol. 8, 2008, pp. 209–224.

[7] Vitaly Chipounov, *S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems*, Ph.D. thesis, École Polytechnique Fédérale de Lausanne (EPFL), 2014.

[8] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea, *Selective Symbolic Execution*, 5th Workshop on Hot Topics in System Dependability (HotDep), 2009.

[9] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea, *S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems*, 16th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2011.

[10] Chris Lattner and Vikram Adve, *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*, Code Generation and Optimization, 2004. CGO 2004. International Symposium on, IEEE, 2004, pp. 75–86.

[11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea, *The S2E Platform: Design, Implementation, and Applications*, ACM Transactions on Computer Systems (TOCS) **30** (2012).