Selective Symbolic Execution

Anonymous, *Technische Universität München*

This paper describes the exemplary application of selective symbolic execution techniques for the analysis of a binary file in user mode. Goal of this study is to search the binary for possible privacy issues like unwanted leakage of personal data. Investigation will be done using S^{2}E, a powerful platform for selective symbolic execution of large software systems.

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 [12]. S^{2}E combines several key characteristics:

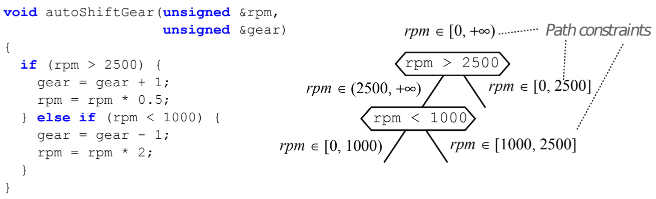
1.) The ability to explore entire *families of execution paths* helps to obtain reliable information about the target system. Abstracting from single-path exploration to sets of execution paths which share specific properties is vital for predictive analyses. This technique can for example prove the 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.

As an exemplary showcase for the power of the S^{2}E platform and its underlying concepts this paper will perform a thorough analysis of a binary file in user mode. The scenario assumes that this program was found somewhere in the internet and claims to be a useful freeware tool. S^{2}E shall help to investigate whether the binary compromises the user's privacy, for instance by leaking private data to the internet.

Chapter 2 explains the theoretical concepts of selective symbolic execution. Chapter 3 then introduces S^{2}E, the platform which builds upon all techniques described before. Coming to the practical part, chapter 4 lays out the concrete analysis scenario and formulates research questions. Chapter 5 describes how S^{2}E can be applied in this scenario, followed by an explanation of S^{2}E results in chapter 6. Possible further research related to this topic is mentioned in chapter 7, together with some selected related work in chapter 8. Finally, chapter 9 summarises and concludes this paper.



```
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]*

2 Selective Symbolic Execution

Symbolic execution is an advanced analysis technique. 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.

Bild > Text

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 ($\cong$ 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 [6, 11]. 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** (S²E) [8]. Based on a virtual execution platform S²E 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, S²E si-

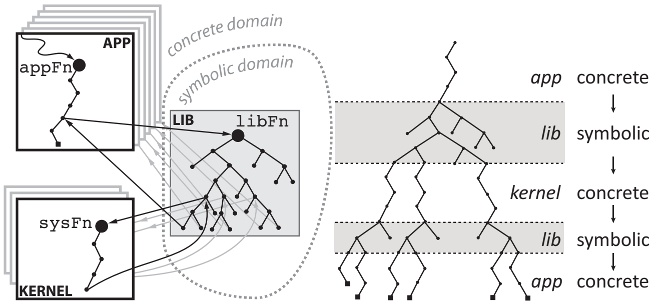
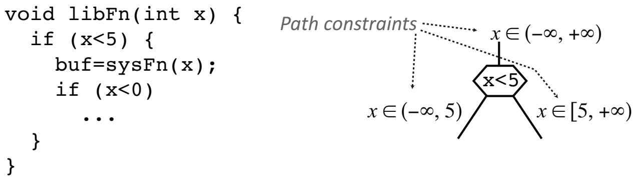


Figure 2: *Selective symbolic execution: only paths inside a defined scope of interest (here: a library function libFn) are explored symbolically - the rest of the system stack runs concretely [7].*



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

Figure 3: *Excerpt from libFn's execution tree [12]*

multaneously 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 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

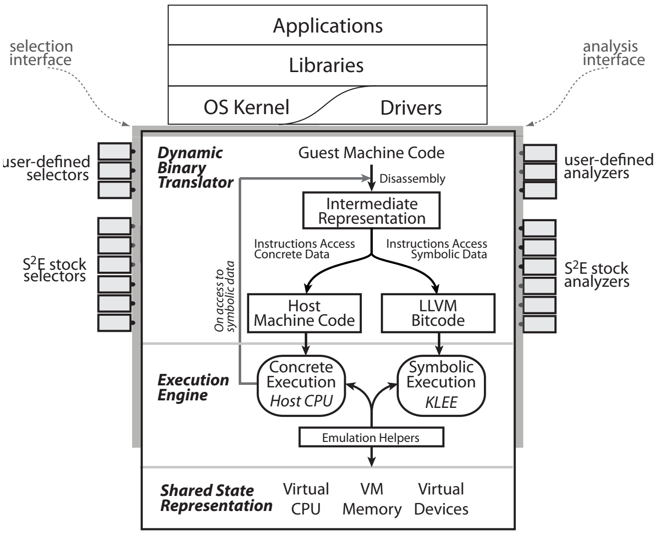


Figure 4: *Architecture of the S^{2}E platform [12]*

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 (LLVM IR), 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 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 representa-

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

tion, called LLVM Bitcode in figure 4.

S²E'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, S²E 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 S²E by writing selection and analysis plugins or by simply configuring S²E'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 S²E'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 [7], p. 50).

m?

4 Project Idea and Research Questions

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 scenario works as follows: The freeware tool *SuperTaxCalcPro* found in the internet claims to be handy for estimating your personal tax load. The tool announces to display a decent advertisement on every startup, which suggests a common and reasonably sound business model. It promises to treat all personal data confidentially, but since the application is closed source, of course this claim cannot be verified easily.

To make sure that *SuperTaxCalcPro* does not compromise the user's privacy, it shall be analysed using selective symbolic execution techniques and the S²E platform.

For a general overview, often a good first step is to have a glance at the program's assembly code. Since static code analysis of large programs is a very time-consuming task obf , that is not what we want to do here. Nevertheless, some useful information can be retrieved from the assembly code quite easily. A look at the list of invoked external libraries shows that *SuperTaxCalcPro* communicates over network sockets, i.e. uses the C library functions connect, write, etc. cmd That is of course not really surprising as the program relies on advertisements as a business model. However, such connections to the internet are always delicate and could potentially leak personal data. Therefore further analysis in this paper will focus on the following concrete questions:

1. When (under which conditions) are connections to the internet established?
2. What data is transferred in these connections?
3. Does a connection to the internet leak personal data?

These questions could of course be tackled via many different analysis techniques. Simply debugging the program should already lead to a rough understanding of the program's behaviour. Applying techniques like fuzzing cit could increase code coverage with the goal of not missing exotic or covert execution paths. This paper, however, will rely on analysis using selective symbolic execution as described in chapter 2. why

5 Implementation

Klar machen, welche Plugins ich verwendet habe! .

Liste. Beschreiben, dass alles in Lua gecodet wird. Note: x86 General notes: before execution in S²E the sample application *SuperTaxCalcPro* was highly optimised. Blabla

As described in chapter 2, working with S²E can be divided into a code selection and the actual analysis part.

In the **code selection phase** S²E is configured to focus on the program *SuperTaxCalcPro* in user space only. The whole environment will be treated as a black box. Symbolic execution will only be applied inside the code of *SuperTaxCalcPro*.

The next step is to precisely specify which variables shall be treated symbolically. In general, as a start we want to make all user inputs symbolic. For a closed source binary this can be done in several ways. A command line tool can be called from a wrapper program which hands over symbolic arguments instead of concrete ones. If user input is entered in a UI, a further look into the assembly code is necessary in order to find the memory locations where the corresponding variables are written.

S²E can intercept execution of the specified memory addresses and replace them with symbolic values (state:writeMemorySymb(...)).

KLEE, the engine for symbolic execution, also requires some configuration. As path search strategy this analysis uses a depth first search.

```
0804940c <_Z9send_dataPc>:
                   .
                   .
 8049448: e8 c3 fa ff ff          call   8048f10 <socket@plt>
                   .
                   .
 80494e9: e8 12 f9 ff ff          call   8048e00 <connect@plt>
 80494ee: c1 e8 1f                shr    $0x1f,%eax
 80494f1: 84 c0                   test   %al,%al
 80494f3: 74 0c                   je     8049501
                                        <_Z9send_dataPc+0xf5>
 80494f5: c7 04 24 83 a0 04 08    movl   $0x804a083,(%esp)
 80494fc: e8 f9 fe ff ff          call   80493fa <_Z5errorPKc>
 8049501: 8d 85 d8 fe ff ff       lea    -0x128(%ebp),%eax
 8049507: 89 04 24                mov    %eax,(%esp)
 804950a: e8 21 fa ff ff          call   8048f30 <strlen@plt>
 804950f: 89 44 24 08             mov    %eax,0x8(%esp)
 8049513: 8d 85 d8 fe ff ff       lea    -0x128(%ebp),%eax
 8049519: 89 44 24 04             mov    %eax,0x4(%esp)
 804951d: 8b 45 f0                mov    -0x10(%ebp),%eax
 8049520: 89 04 24                mov    %eax,(%esp)
 8049523: e8 58 f9 ff ff          call   8048e80 <write@plt>
                   .
                   .
 804955b: 8d 85 d8 fe ff ff       lea    -0x128(%ebp),%eax
 8049561: 89 44 24 04             mov    %eax,0x4(%esp)
 8049565: 8b 45 f0                mov    -0x10(%ebp),%eax
 8049568: 89 04 24                mov    %eax,(%esp)
 804956b: e8 70 f9 ff ff          call   8048ee0 <read@plt>
                   .
                   .
```

Listing 1: *Relevant parts of the function send_data in assembly code. Interesting calls are highlighted.*

Most logic in the **analysis part** relies on S²E's *Annotations* plugin. It allows fine monitoring and even fiddling with the execution state by annotating single instructions or functions in the program binary.

For the scenario in this paper all instructions which handle connections to the internet are of particular interest. The C library call *connect* helps to identify where the program is connecting to, and the corresponding *write* and *read* calls allow to find out what data is sent and received in this connection. Memory addresses of these relevant calls can be retrieved from the assembly code. Since all calls concerned with connecting to the internet seem to take place in a single function named *send_data*, calls to this function as a whole shall be monitored, too. Listing 1 shows relevant parts of the function *send_data* in assembly code.

Verbindung

```
pluginsConfig.Annotation = {
  annotation_write = {
    active=true,
    module="prog",
    address=0x8049523, -- write()
    instructionAnnotation="do_write",
    beforeInstruction=true,
    switchInstructionToSymbolic=false
  },
  annotation_read = {
    active=true,
    module="prog",
    address=0x804956b, -- read()
    instructionAnnotation="do_read",
    beforeInstruction=false,
    switchInstructionToSymbolic=false
  },
  annotation_send_data = {
    active=true,
    module="prog",
    address=0x804940c, -- send_data()
    callAnnotation="call_ann",
    paramcount = 1
  }
}
```

Listing 2: *Configuration of the Annotations plugin (part). Defines the instructions to be monitored and actions to trigger upon execution of these instructions. Note the link to the binary in listing 1 via memory addresses.*

After providing S²E's *Annotations* plugin with all relevant memory addresses plus a little more configuration (shown in listing 2), we can now execute code every time the program runs into one of the interesting instructions. At this point, we will . . .

1.) log that instruction *X* was reached. Together with other S²E output this information shows which execution paths run into the *send_data* method and when they do so.

2.) save information about the executed instruction in the current plugin state. This allows to check whether there have been prior connections to the internet when the program runs into the next annotated instruction.

3.) read out parameters of the called function. The function *send_data*, for instance, is called with one parameter, which appears to be a memory location. Now S^{2}E's analysis interfaces can be employed to dynamically find out what is written in the respective memory.

4.) read (or write) registers (EAX, EBP, ...). Thus S^{2}E users can check and even manipulate the current execution state, similar to debug situations.

5.) switch variables (memory locations) from concrete to symbolic mode (or vice versa). ... Nice for controlling consistency models. Not used currently.

Listing 3, for instance, shows the Lua function which is triggered on every execution of the assembly function *send_data* (see listing 2 for the registration of this function and its coupling to a specific memory address). Because *send_data* was registered as a function call in listing 2, S^{2}E's interfaces facilitate the extraction of all parameters via *state:readParameter()*. A sample test run shows that $arg0$ is clearly a memory address, so the next step must be to examine what data is stored at this location. The method *print_mem* does just that and additionally brings the contents into a human-readable format. Then the Lua script stores information that this function was called in the current execution state. Doing so, the next call will notice that the method has been executed before, increment the counter and write this fact into a log file.

The *else* branch will be called when the function returns. Following common calling conventions, we expect return values to be transmitted via the register EAX (and EDX if bigger than four bytes), hence EAX could also hold interesting information. Unfortunately, this does not seem to be the case as EAX is always 0 except in some error cases. So *send_data* is probably just returning a status integer value.

```lua
function call_ann (state, curPlgState)
  if curPlgState:isCall() then
    arg0 = state:readParameter(0);
    print ("Calling function with with arg0=" .. ("%x"):format(arg0));
    -- print_mem dereferences the pointer arg0 and converts the memory behind that address into a string.
    print_mem(state, curPlgState, arg0);
    -- saving info about the no. of connections in this state so far
    local no =curPlgState:getValue("no");
    if no == nil then
      no = 0
    end
    no = no + 1;
    curPlgState:setValue("no", no);
    print ("\n\tNo. of connections on this path so far: " .. no);
  else
    -- Called when the function send_data returns. The return value can be read from the EAX register.
    local eax =state:readRegister("eax");
    print("EAX: " .. ("%x"):format(eax));
  end
end
```

Listing 3: *Lua function executed upon every call of the function send_data(). See lines 18 - 24 in listing 2 for the registration of call_ann.*

In addition to the *Annotations* plugin, analysis in this paper also employs the *TestCaseGenerator*. For each execution path, it finds concrete inputs for all symbolic variables. Those will be printed upon termination of the path and, apart from helping to understand the program, may serve as input for further testing.

Naturally, an important step for understanding symbolic execution of *SuperTaxCalcPro* is to log and later interpret output of KLEE, the symbolic execution engine used in S^{2}E. The behaviour of KLEE can also be controlled via S^{2}E's Lua configuration file.

6 Analysis of S^{2}E Output

Each execution of S^{2}E creates a folder for analysis output. Apart from the general log files, where all plugins write important messages, some plugins also create separate files. One file for memory ..., one with LLVM bytecode, one ...

The most important information about S^{2}E's general execution is accumulated in *messages.txt*. Backbone of this file is information about all execution paths, at which memory addresses they were forked, how they are constrained, when and why they were terminated, and much more.

Thanks to the clear structure of *messages.txt*, it can be used as input for a custom Python script. This script was written in order to visualise the tree of

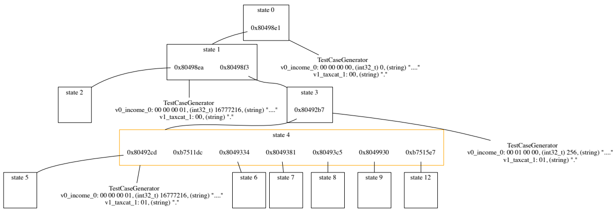

state 0
0x80498e1
TestCaseGenerator
v0_income_0: 00 00 00 00, (int32_t) 0, (string) "...."
v1_taxcat_1: 00, (string) "."
state 1
0x80498ea
0x80498f3
state 2
TestCaseGenerator
v0_income_0: 00 00 00 01, (int32_t) 16777216, (string) "...."
v1_taxcat_1: 00, (string) "."
state 3
0x80492b7
state 4
0x80492cd
0xb7511dc
0x8049334
0x8049381
0x80493c5
0x8049930
0xb7515e7
TestCaseGenerator
v0_income_0: 00 01 00 00, (int32_t) 256, (string) "...."
v1_taxcat_1: 01, (string) "."
state 5
TestCaseGenerator
v0_income_0: 00 00 00 01, (int32_t) 16777216, (string) "...."
v1_taxcat_1: 01, (string) "."
state 6
state 7
state 8
state 9
state 12

Figure 5: *Graph*

execution paths graphically, which serves as a great help with understanding path forking behaviour. Figure **??** shows the root of the tree of execution paths of *SuperTaxCalcPro*. Each box represents an execution state, lines pointing to another state symbolise a fork of the execution state. The hex number at the origin of each edge is the memory address in which the state was forked. Edges are labelled with all constraints that need to be respected in the corresponding execution state. Blue state boxes indicate one connection to the internet so far, yellow two and red three. The text below each leaf shows the output of the *TestCaseGenerator* plugin for this execution path.

It is only the root because....

The whole graph (without constraints) looks as folllows:...

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] Corina Pasareanu, Peter C Mehlitz, David Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape, *Combining Unit-Level Symbolic Execution and System-Level Concrete Execution for Testing NASA Software*, Proceedings of the 2008 international symposium on Software testing and analysis, ACM, 2008, pp. 15–26.

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