S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems

Vitaly Chipounov, Volodymyr Kuznetsov, George Candea

School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
{vitaly.chipounov,vova.kuznetsov,george.candea}@epfl.ch

Abstract

This paper presents S^{2}E, a platform for analyzing the properties and behavior of software systems. We demonstrate S^{2}E's use in developing practical tools for comprehensive performance profiling, reverse engineering of proprietary software, and bug finding for both kernel-mode and user-mode binaries. Building these tools on top of S^{2}E took less than 770 LOC and 40 person-hours each.

S^{2}E's novelty consists of its ability to scale to large real systems, such as a full Windows stack. S^{2}E is based on two new ideas: *selective symbolic execution*, a way to automatically minimize the amount of code that has to be executed symbolically given a target analysis, and relaxed *execution consistency models*, a way to make principled performance/accuracy trade-offs in complex analyses. These techniques give S^{2}E three key abilities: to simultaneously analyze entire families of execution paths, instead of just one execution at a time; to perform the analyses in-vivo within a real software stack—user programs, libraries, kernel, drivers, etc.—instead of using abstract models of these layers; and to operate directly on binaries, thus being able to analyze even proprietary software.

Conceptually, S^{2}E is an automated path explorer with modular path analyzers: the explorer drives the target system down all execution paths of interest, while analyzers check properties of each such path (e.g., to look for bugs) or simply collect information (e.g., count page faults). Desired paths can be specified in multiple ways, and S^{2}E users can either combine existing analyzers to build a custom analysis tool, or write new analyzers using the S^{2}E API.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]

General Terms Reliability, Verification, Performance, Security

1. Introduction

System developers routinely need to analyze the behavior of what they build. One basic analysis is to *understand observed behavior*, such as why a given web server is slow on a SPECweb benchmark. More sophisticated analyses aim to *characterize future behavior* in previously unseen circumstances, such as what will a web server's maximum latency and minimum throughput be, once deployed at a customer site. Ideally, system designers would also like to be able to do quick *what-if analyses*, such as determining whether aligning a certain data structure on a page boundary will avoid all cache misses and thus increase performance. For small programs, experienced developers can often reason through some of these questions based on code alone. The goal of our work is to make it feasible to answer such questions for large, complex, real systems.

We introduce in this paper a platform that enables easy construction of analysis tools (such as oprofile, valgrind, bug finders, or reverse engineering tools) that simultaneously offer the following three properties: (1) they efficiently analyze entire families of execution paths; (2) they maximize realism by running the analyses within a real software stack; and (3) they are able to directly analyze binaries. We explain these properties below.

Why do you
want these?

First, predictive analyses often must reason about entire *families of paths* through the target system, not just one path. For example, security analyses must check that there exist no corner cases that could violate a desired security policy; recent work has employed model checking [29] and symbolic execution [11] to find bugs in real systems—these are all multi-path analyses. One of our case studies demonstrates multi-path analysis of performance properties: instead of profiling solely one execution path, we derive performance envelopes that characterize the performance of entire families of paths. Such analyses can check real-time requirements (e.g., that an interrupt handler will never exceed a given bound on execution time), or can help with capacity planning (e.g., determine how many web servers to provision for a web farm). In the end, properties shown to hold for *all* paths constitute proofs, which are in essence the ultimate prediction of a system's behavior.

Second, an accurate estimate of program behavior often requires taking into account the *whole environment* surrounding the analyzed program: libraries, kernel, drivers, etc.—in other words, it requires in-vivo[1] analysis. Even small programs interact with their environment (e.g., to read/write files or send/receive network packets), so understanding program behavior requires understanding the nature of these interactions. Some tools execute the real environment, but allow calls from different execution paths to interfere inconsistently with each other [12, 18]. Most approaches abstract away the environment behind a model [2, 11], but writing abstract models is labor-intensive (taking in some cases multiple person-years [2]), models are rarely 100% accurate, and they tend to lose

[1] *In vivo* is Latin for "within the living" and refers to experimenting using a whole live system; *in vitro* uses a synthetic or partial system. In life sciences, in vivo testing—animal testing or clinical trials—is often preferred, because, when organisms or tissues are disrupted (as in the case of in vitro settings), results can be substantially less representative. Analogously, in-vivo program analysis captures all interactions of the analyzed code with its surrounding system, not just with a simplified abstraction of that system.

ASPLOS'11, March 5–11, 2011, Newport Beach, California, USA.
Copyright © 2011 ACM 978-1-4503-0266-1/11/03. . . $10.00

accuracy as the modeled system evolves. It is therefore preferable that target programs interact directly with their real environment during analysis in a way that keeps multi-path analysis consistent.

At what cost?

Third, real systems are made up of many components from various vendors; access to all corresponding source code is rarely feasible and, even when source code is available, building the code exactly as in the shipped software product is difficult [5]. Thus, in order to be practical, analyses ought to operate *directly on binaries*.

Scalability is the key challenge of performing analyses that are in-vivo, multi-path, and operate on binaries. Going from single-path analysis to multi-path analysis turns a linear problem into an exponential one, because the number of paths through a program increases exponentially in the number of branches—the "path explosion" problem [7]. It is therefore not feasible today to execute fully symbolically an entire software stack (programs, libraries, OS kernel, drivers, etc.) as would be necessary if we wanted consistent in-vivo multi-path analysis.

We describe in this paper S^{2}E, a general platform for developing multi-path in-vivo analysis tools that are practical even for large, complex systems, such as an entire Windows software stack. First, S^{2}E simultaneously exercises entire families of execution paths in a scalable manner by using *selective* symbolic execution and *relaxed* execution consistency models. Second, S^{2}E employs virtualization to perform the desired analyses in vivo; this removes the need for the stubs or abstract models required by most state-of-the-art symbolic execution engines and model checkers [3, 11, 18, 29, 36]. Third, S^{2}E uses dynamic binary translation to directly interpret x86 machine code, so it can analyze a wide range of software, including proprietary systems, even if self-modifying or JITed, as well as obfuscated and packed/encrypted binaries.

Go over these

The S^{2}E platform offers an automated path exploration mechanism and modular path analyzers. The explorer drives in parallel the target system down all execution paths of interest, while analyzers check properties of each such path (e.g., to look for bugs) or simply collect information (e.g., count page faults). An analysis tool built on top of S^{2}E glues together path selectors with path analyzers. *Selectors* guide S^{2}E's path explorer by specifying the paths of interest: all paths that touch a specific memory object, paths influenced by a specific parameter, paths inside a target code module, etc. *Analyzers* can be pieced together from S^{2}E-provided analyzers, or can be written from scratch using the S^{2}E API.

S^{2}E comes with ready-made selectors and analyzers that provide a wide range of analyses out of the box. The typical S^{2}E user only needs to define in a configuration file the desired selector(s) and analyzer(s) along with the corresponding parameters, start up the desired software stack inside the S^{2}E virtual machine, and run the S^{2}E launcher in the guest OS, which starts the desired application and communicates with the S^{2}E VM underneath. For example, one may want to verify the code that handles license keys in a proprietary program, such as Adobe Photoshop. The user installs the program in the S^{2}E Windows VM and launches the program using *s2e.exe C:\Program Files\Adobe\Photoshop*. From inside the guest OS, the *s2e.exe* launcher communicates with S^{2}E via custom opcodes (described in §4). In the S^{2}E configuration file, the tester may choose a memory-checker analyzer along with a path selector that returns a symbolic string whenever Photoshop reads *HKEY_LOCAL_MACHINE\Software\Photoshop\LicenseKey* from the Windows registry. S^{2}E then automatically explores the code paths in Photoshop that are influenced by the value of the license key and looks for memory safety errors along those paths.

Developing a new analysis tool with S^{2}E takes on the order of 20-40 person-hours and a few hundred LOC. To illustrate S^{2}E's generality, we present here three very different tools built using S^{2}E: a multi-path in-vivo performance profiler, a reverse engineering tool, and a tool for automatically testing proprietary software.

This paper makes the following four contributions:

- ***Selective symbolic execution***, a new technique for automatic bidirectional symbolic–concrete state conversion that enables execution to seamlessly and correctly weave back and forth between symbolic and concrete mode;
- ***Execution consistency models***, a systematic way to reason about the trade-offs involved in over/under-approximation of paths in software system analyses;
- A ***general platform*** for performing diverse in-vivo multi-path analyses in a way that scales to large real systems;
- The first use of ***symbolic execution in performance analysis***.

Pretty cool

In the rest of the paper, we describe selective symbolic execution (§2), execution consistency models (§3), S^{2}E's APIs for developing analysis tools (§4), the S^{2}E prototype (§5), evaluation (§6), related work (§7), and conclusions (§8).

2. Selective Symbolic Execution

In devising a way to efficiently exercise entire families of paths, we were inspired by the successful use of symbolic execution [22] in automated software testing [11, 18]. The idea is to treat a program as a superposition of possible execution paths. For example, a program that is all linear code except for one conditional statement *if* ($x{>}0$) *then ... else ...* can be viewed as a superposition of two possible paths: one for $x{>}0$ and another one for $x{\leq}0$. To exercise all paths, it is not necessary to try all possible values of x, but rather just one value greater than 0 and one value less than 0.

We unfurl this superposition of paths into a *symbolic execution tree*, in which each possible execution corresponds to a path from the root of the tree to a leaf corresponding to a terminal state. The mechanics of doing so consist of marking variables as symbolic at the beginning of the program, i.e., instead of allowing a variable x to take on a concrete value (say, $x{=}5$), it is viewed as a superposition λ of all possible values x could take. Then, any time a branch instruction is conditioned on a predicate p that depends (directly or indirectly) on x, execution is split into two executions E_i and E_k, two copies of the program's state are created, and E_i's path remembers that the variables involved in p must be constrained to make p true, while E_j's path remembers that p must be false.

The process repeats recursively: E_i may further split into E_{i_i} and E_{i_k}, and so on. Every execution of a branch statement creates a new set of children, and thus what would normally be a linear execution (if concrete values were used) now turns into a tree of executions (since symbolic values are used). A node s in the tree represents a program state (a set of variables with formulae constraining the variables' values), and an edge $s_i \rightarrow s_j$ indicates that s_j is s_i's successor on any path satisfying the constraints in s_j. Paths in the tree can be pursued simultaneously, as the tree unfurls; since program state is copied, the paths can be explored independently. Copy-on-write is used to make this process efficient.

S^{2}E is based on the key observation that often *only some* families of paths are of interest. For example, one may want to exhaustively explore all paths through a small program, but not care about all paths through the libraries it uses or the OS kernel. This means that, when entering that program, S^{2}E should split executions to explore the various paths, but whenever it calls into some other part of the system, such as a library, multi-path execution can cease and execution can revert to single-path. Then, when execution returns to the program, multi-path execution must be resumed.

Selective

Multi-path execution corresponds to *expanding* a family of paths by exploring the various side branches as they appear, while switching to single-path mode corresponds to *corseting* the family of paths. In multi-path mode, the tree grows in width and depth; in single-path mode, the tree only grows in depth. We therefore say S^{2}E's exploration of program paths is *elastic*. S^{2}E turns multi-path mode off whenever possible, to minimize the size of the execution tree and include only paths that are of interest to the target analysis.

S^{2}E's elasticity of multi-path exploration is key in being able to perform in-vivo multi-path exploration of programs inside complex systems, like Windows. By combining elasticity with virtualization, S^{2}E offers the illusion of symbolically executing a full software stack, while actually executing symbolically only select components. For example, by concretely (i.e., non-symbolically) executing libraries and the OS kernel, S^{2}E allows a program's paths to be explored efficiently without having to model its surrounding environment. We refer to this as *selective symbolic execution*.

How do you
make single
syscall when
args are
symbolic?

Interleaving of symbolic execution phases with concrete phases must be done carefully, to preserve the meaningfulness of each explored execution. For example, say we wish to analyze a program P in multi-path (symbolic) mode, but none of its libraries L_i are to be explored symbolically. If P has a symbolic variable n and calls `strncpy(dst,src,n)` in L_k, S^{2}E must convert n to some concrete value and invoke `strncpy` with that value. This is straightforward: solve the current path constraints with a constraint solver and get some legal value for n (say n=5) and call `strncpy`. But what happens to n after `strncpy` returns? Variable *dst* will contain n=5 bytes, whereas n prior to the call was symbolic—can n still be treated symbolically? The answer is yes, if done carefully.

In S^{2}E, when a symbolic value is converted to concrete ($n : \lambda \to 5$), the family of executions is corseted. When a concrete value is converted to symbolic ($n : 5 \to \lambda$), the execution family is allowed to expand. The process of doing this back and forth is governed by the rules of an execution consistency model (§3). For the above example, one might require that n be constrained to value 5 in all executions following the return from `strncpy`. However, doing so may exclude a large number of paths from the analysis. In §3 we describe systematic and safe relaxations of execution consistency.

We now describe the mechanics of switching back and forth between multi-path (symbolic) and single-path (concrete) execution in a way that executions stay consistent. We know of no prior symbolic execution engine that has the machinery to efficiently and flexibly cross the symbolic/concrete boundary both back and forth.

Fig. 1 provides a simplified example of using S^{2}E: an application *app* uses a library *lib* on top of an OS *kernel*. The target analysis requires to symbolically execute *lib*, but not *app* or *kernel*. Function *appFn* in the application calls a library function *libFn*, which eventually invokes a system call *sysFn*. Once *sysFn* returns, *libFn* does some further processing and returns to *appFn*. After the execution crosses into the symbolic domain (shaded) from the concrete domain (white), the execution tree (right side of Fig. 1) expands. After the execution returns to the concrete domain, the execution tree is corseted and does not add any new paths, until execution returns to the symbolic domain. Some paths may terminate earlier than others, e.g., due to hitting a bug in the program.

We now describe the two directions in which execution can cross the concrete/symbolic boundary.

2.1 Concrete → Symbolic Transition

When *appFn* calls *libFn*, it does so by using concrete arguments; the simplest conversion is to use an S^{2}E selector to change the concrete arguments into symbolic ones, e.g., instead of *libFn(10)* call *libFn(λ)*. One can additionally opt to constrain λ, e.g., $\lambda \leq 15$.

Once this transition occurs, S^{2}E executes *libFn* symbolically using the (potentially constrained) argument(s) and simultaneously executes *libFn* with the original concrete argument(s) as well. Once exploration of *libFn* completes, S^{2}E returns to *appFn* the concrete return value resulting from the concrete execution, but *libFn* will have been explored symbolically as well. In this way, the execution of *app* is consistent, while at the same time S^{2}E exposes to the analyzer plugins those paths in *lib* that are rooted at *libFn*'s entry point. The concrete domain is unaware of *libFn* being executed in multi-path mode. All paths execute independently, and it is up to the S^{2}E analyzer plugins to decide whether, besides observing the concrete path, they also wish to look at the symbolic paths.

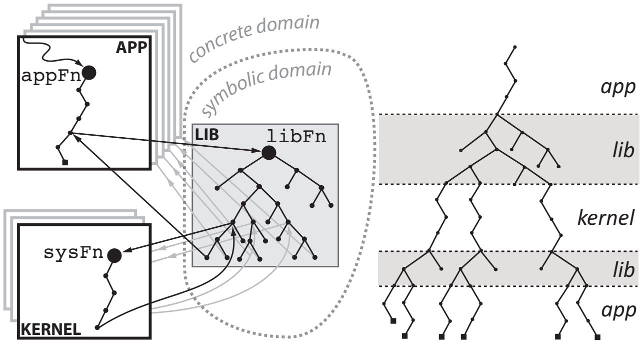


Figure 1: Multi-path/single-path execution: three different modules (left) and the resulting execution tree (right). Shaded areas represent the multi-path (symbolic) execution domain, while the white areas are single-path.

2.2 Symbolic → Concrete Transition

Dealing with the *libFn→sysFn* call is more complicated. Say *libFn* has the code shown in Fig. 2 and was called with an unconstrained symbolic value $x \in (-\infty, +\infty)$. At the first *if* branch instruction, execution forks into one path along which $x \in (-\infty, 5)$ and another path where $x \in [5, +\infty)$. These expressions are referred to as *path constraints*, as they constrain the values that x can take on a path. Along the then-branch, a call to *sysFn(x)* must be made. This requires x to be concretized, since *sysFn* is in the concrete domain. Thus, we choose a value, say x=4, that is consistent with the $x \in (-\infty, 5)$ constraint and perform the *sysFn(4)* call. The path constraints in the symbolic domain are updated to reflect that x=4.

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

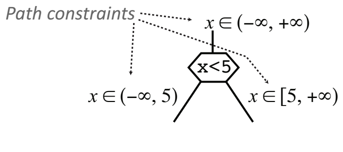


Figure 2: The top level in *libFn*'s execution tree.

Note that S^{2}E actually employs *lazy concretization*: it converts the value of x from symbolic to concrete on-demand, only when concretely running code actually reads the value of x. This is an important optimization when doing in-vivo symbolic execution, because a lot of data can be carried through the layers of the software stack without conversion. For example, when a program writes a buffer of symbolic data to the filesystem, there are usually no branches in the kernel or the disk device driver that depend on this data, so the buffer can pass through unconcretized and be written in symbolic form to the virtual disk, from where it will eventually be read back in its symbolic form. For the sake of clarity, in this section we assume eager (non-lazy) concretization.

Once *sysFn* completes, execution returns to *libFn* in the symbolic domain. When x was concretized prior to calling *sysFn*, the x=4 constraint was automaticaly added to the path constraints—*sysFn*'s return value is correct only under this constraint, because all computation in *sysFn* was done assuming x=4. Furthermore, *sysFn* may also have had side effects that are equally intimately tied to the x=4 constraint. With this constraint, execution of *libFn* can continue, and correctness is fully preserved.

The problem, however, is that this constraint corsets the family of *future* paths that can be explored from this point on: x can no longer take on all values in $(-\infty, 5)$ so, when we subsequently get to the branch *if* $(x{<}0)$ ..., the then-branch will no longer be feasible due to the added $x{=}4$ constraint. This is referred to as "overconstraining": the constraint is not introduced by features of *libFn*'s code, but rather as a result of concretizing x to call into the concrete domain. We think of $x{=}4$ as a soft constraint imposed by the symbolic/concrete boundary, while $x{\in}(-\infty, 5)$ is a hard constraint imposed by *libFn*'s code. Whenever a branch in the symbolic domain is disabled because of a soft constraint, it is possible to go back in the execution tree and pick an additional value for concretization, fork another subtree, and repeat the *sysFn* call in a way that would enable that branch. As explained later, S^{2}E can track branch conditions in the concrete domain, which helps redo the call in a way that re-enables subsequent branches.

Cool, but how costly is this?

The "overconstraining" problem has two components: (a) the loss of paths that results directly from the concretization of x, and (b) the loss of paths that results indirectly via the constrained return value and side effects. Due to the fact that S^{2}E implements VM state in a way that is shared between the concrete and symbolic domain (more details in §5), return values and side effects can be treated using identical mechanisms. We now discuss how the constraints are handled under different consistency models.

3. Execution Consistency Models

The traditional assumption about system execution is that the state at any point in time is consistent, i.e., there exists a feasible path from the start state to the current state. However, there are many analyses for which this assumption is unnecessarily strong, and the cost of providing such consistency during multi-path exploration is often prohibitively high. For example, when doing unit testing, one typically exercises the unit in ways that are consistent with the unit's interface, without regard to whether all those paths are indeed feasible in the integrated system. This is both because testing the entire system in a way that exercises all paths through the unit is too expensive, and because exercising the unit as described above offers higher confidence in its correctness in the face of future use.

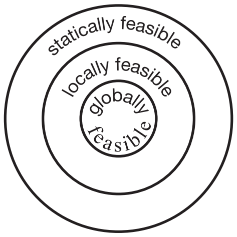
S^{2}E aims to be a general platform for system analyses, so it provides several levels of execution consistency, to enable users to make the best trade-offs. In this section, we take a first step toward systematically defining alternate execution consistency models (§3.1), after which we explain how these different models dictate the symbolic/concrete conversions applied during the back-and-forth transition between the analyzed code and its environment (§3.2). In §3.3 we survey some of the ways in which consistency models are implemented in existing analysis tools.

Doesn't it give part of this up by working on binaries?

3.1 Model Definitions

The key distinction between the various execution consistency models is which execution paths each model admits. Choosing an appropriate consistency model is a trade-off between how "realistic" the admitted paths are vs. the cost of enforcing the model. The appropriateness of the trade-off is determined by the nature of the analysis, i.e., by the way in which feasibility of different paths affects completeness and soundness of the analysis.

In the rest of the paper, we use the term *system* to denote the complete software system under analysis, including the application programs, libraries, and the operating system. We use the term *unit* to denote the part of the system that is to be analyzed. A unit could encompass different parts of multiple programs, libraries, or even parts of the operating system itself. We use the term *environment* to denote everything in the system except the unit. Thus, the system is the sum of the environment and the unit to be analyzed.



When defining a model, we think in terms of which paths it includes vs. excludes. Following the Venn diagram on the right, an execution path can be *statically feasible*, in that there exists a path in the system's inter-procedural control flow graph (CFG) corresponding to the execution in question. A subset of the statically feasible paths are *locally feasible* in the unit, in the sense that the execution is consistent with both the system's CFG and with the restrictions on control flow imposed by the data-related constraints within the unit. Finally, a subset of locally feasible paths is *globally feasible*, in the sense that their execution is additionally consistent with control flow restrictions imposed by data-related constraints in the environment. Observing only the code executing in the unit, with no knowledge of code in the environment, it is impossible to tell apart locally feasible from globally feasible paths.

We say that a model is *complete* if every path through the unit that corresponds to some globally feasible path through the system will eventually be discovered by exploration done under that model. A model is *consistent* if, for every path through the unit admissible by the model, there exists a corresponding globally feasible path through the system (i.e., the system can run concretely in that way).

Why do we care about these?

We now define six points that we consider of particular interest in the space of possible consistency models, progressing from strongest to weakest consistency. They are summarized in Fig. 3 using a representation corresponding to the Venn diagram above. Their completeness and consistency are summarized in Table 1. We invite the reader to follow Fig. 3 while reading this section.

3.1.1 Strict Consistency (SC)

The strongest form of consistency is one that admits only the globally consistent paths. For example, the concrete execution of a program always obeys the strict consistency (SC) model. Moreover, every path admitted under the SC model can be mapped to a certain concrete execution of the system starting with certain concrete inputs. Sound analyses produce no false positives under SC.

We define three subcategories of SC based on what information is taken into account when exploring new paths.

Strictly Consistent Concrete Execution (SC-CE): Under the SC-CE model, the entire system is treated as a black box: no internal information is used to explore new paths. The only explored paths are the paths that the system follows when executed with the sample input provided by the analysis. New paths can only be explored by blindly guessing new inputs. Classic fuzzing (random input testing) falls under this model.

Strictly Consistent Unit-level Execution (SC-UE): Under the SC-UE model, an exploration engine is allowed to gather and use information internal to the unit (e.g., by collecting path constraints while executing the unit). The environment is still treated as a black box, i.e., path constraints generated by environment code are not tracked. Not every globally feasible path can be found with such partial information (e.g., paths that are enabled by branches in the environment can be missed). However, the exploration engine saves time by not having to analyze the environment, which is typically orders of magnitude larger than the unit.

This model is widely used by symbolic and concolic execution tools [11, 12, 18]. Such tools usually instrument the program but not the operating system code (sometimes such tools replace parts of the OS by models, effectively adding a simplified version of it as a part of the program). Whenever such tools see a call to the OS, they execute the call uninstrumented, selecting some concrete arguments for the call. Such "blind" selection of concrete

Whats this?

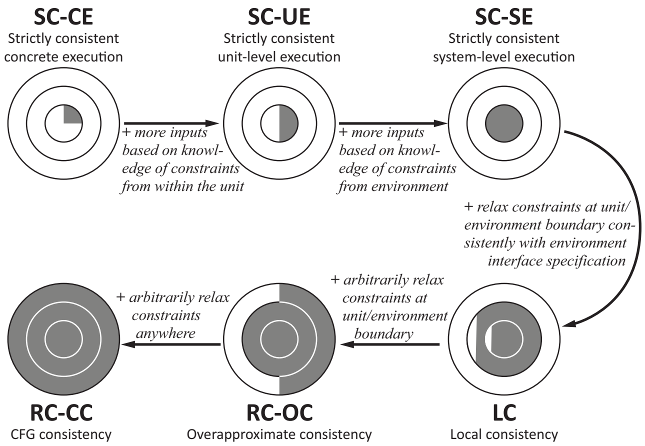

SC-CE
Strictly consistent concrete execution
SC-UE
Strictly consistent unit-level execution
SC-SE
Strictly consistent system-level execution
+ more inputs based on knowl-edge of constraints from within the unit
+ more inputs based on knowl-edge of constraints from environment
+ relax constraints at unit/ environment boundary con-sistently with environment interface specification
+ arbitrarily relax constraints at unit/environment boundary
+ arbitrarily relax constraints anywhere
RC-CC
CFG consistency
RC-OC
Overapproximate consistency
LC
Local consistency

Figure 3: Different execution consistency models cover different sets of feasible paths. The SC-CE model corresponds to the concrete execution. The SC-UE and SC-SE models are obtained from the previous ones by using increasingly more information about the system execution to explore new states. The LC, RC-OC and RC-CC models are obtained through progressive relaxation of constraints.

arguments might cause some paths through the unit to be missed, if they depend on unexplored environment behaviors.

Strictly Consistent System-level Execution (SC-SE): Under the SC-SE model, an exploration engine gathers and uses information about all parts of the system, to explore new paths through the unit. Such exploration is not only sound but also complete, provided that the engine can solve all constraints it encounters. In other words, every path through the unit that is possible under a concrete execution of the system will be eventually found by SC-SE exploration, making SC-SE the only model that is both strict and complete.

However, the implementation of SC-SE is limited by the path explosion problem: the number of globally feasible paths is roughly exponential in the size of the whole system. As the environment is typically orders of magnitude larger than the unit, including its code in the analysis (as would be required under SC-SE) offers an unfavorable trade-off given today's technology.

3.1.2 Local Consistency (LC)

The local consistency (LC) model aims to combine the performance advantages of SC-UE with the completeness advantages of SC-SE. The idea is to avoid exploring all paths through the environment, yet still explore the corresponding path segments in the unit by replacing the results of (some) calls to the environment with symbolic values that represent any possible valid result of the execution.

For example, when a unit (such as a user-mode program) invokes the `write(fd, buf, count)` system call of a POSIX OS, the return value can be any integer between -1 and `count`, depending on the state of the system. The exploration engine can discard the actual concrete value returned by the OS and replace it with a symbolic integer between -1 and `count`. This allows exploring all paths in the unit that are enabled by different return values of `write`, without analyzing the `write` function and having to find concrete inputs to the overall system that would enable those paths. This however introduces global inconsistency—for instance, there exists no concrete execution in which `count` bytes are written to the file and the `write` system call returns 0. However, unless the unit explicitly checks the file (e.g., by reading its content) this does not matter: the inconsistency cannot yield locally infeasible paths.

Does this matter in practice?

In other words, the LC model allows for inconsistencies in the environment, while keeping the state of the unit internally consistent. To preserve LC, an exploration engine must track the propagation of inconsistencies inside the environment and abort an execution path as soon as these inconsistencies influence the internal state of the unit on that path.

This keeps the internal state of the unit internally consistent on all explored paths: for each explored path, there exists some concrete execution of the system that would lead to exactly the same internal state of the unit along that path—except the engine does not need to incur the cost of actually finding that path. Consequently, any sound analysis that takes into account only the internal state of the unit produces no false positives under the LC model. For this reason, we call the LC model "locally consistent."

The set of paths explored under this model corresponds to the set of locally feasible paths, as defined earlier. However, some paths could be aborted before completion, or even be missed completely, due to the propagation of inconsistencies. This means that the LC model is not complete. In practice, the less a unit interacts with its environment, the fewer such paths are aborted or missed.

Technically speaking, the LC model is inconsistent, thus it ought to be a sub-model of the RC model, described next. However, since the LC model is equivalent to a SC model for a large class of analyses, we devoted to it an independent category.

3.1.3 Relaxed Consistency (RC)

Under relaxed consistency (RC), all paths through the unit are admitted, even those that are not allowed by the SC and LC models. The RC model is therefore inconsistent in the general case.

The main advantage of RC is performance: by admitting these additional infeasible paths, one can avoid having to analyze large parts of the system that are not really targets of the analysis, thus allowing path exploration to reach the true target code sooner. However, admitting locally infeasible paths (i.e., allowing the internal state of the unit to become inconsistent) makes most analyses prone to false positives, because some of the paths these analyses are exposed to cannot be produced by any concrete run.

This might be acceptable if the analysis is itself unsound anyway, or if the analysis only relies on a subset of the state that can be easily kept consistent (in some sense, this is like LC, except that the subset of the state to be kept consistent may not be the unit's state). Also note that, even though RC admits more paths, thus producing more analysis work, analyses under RC can abort early those paths that turn out to be infeasible, or the accuracy of the analysis can be decreased, thus preserving the performance advantage.

We distinguish two subcategories of the RC model, both of which we found to be useful in practice.

Overapproximate Consistency (RC-OC): In the RC-OC model, path exploration can follow paths through the unit while completely ignoring the constraints that the environment/unit API contracts impose on return values and side effects. For example, the unit may invoke `write(fd, buf, count)`, and the RC-OC model would permit the return result to be larger than `count`, which violates the specification of the `write` system call. Under the previous model (local consistency), such paths would be disallowed. Even though it is not consistent, RC-OC is complete: every environment behavior is admitted under RC-OC, so every path in the unit corresponding to some real environment behavior is admitted too.

The RC-OC model is useful, for example, for reverse engineering. It enables efficient exploration of all behaviors of the unit that are possible in a valid environment, plus some additional behaviors that are possible only when the environment behaves outside its specification. For instance, when reverse engineering a device driver, the RC-OC model allows symbolic hardware [23] to return unconstrained values; in this way, the resulting reverse engineered paths include some of those that correspond to allegedly impossible hardware behaviors. Such overapproximation improves the quality of the reverse engineering, as explained in [13].

Model	Consistency	Completeness	Use Case
SC-CE	consistent	incomplete	Single-path profiling/testing of units that have a limited number of paths
SC-UE	consistent	incomplete	Analysis of units that generate hard-to-solve constraints (e.g., cryptographic code)
SC-SE	consistent	complete	Sound and complete verification without false positives or negatives; testing of tightly coupled systems with fuzzy unit boundaries.
LC	locally consistent	incomplete	Testing/profiling while avoiding false positives from the unit's perspective
RC-OC	inconsistent	complete	Reverse engineering: extract consistent path segments
RC-CC	inconsistent	complete	Dynamic disassembly of a potentially obfuscated binary

Table 1: S^{2}E consistency models: completeness, consistency and use cases. Each use case is assigned to the weakest model it can be accomplished with.

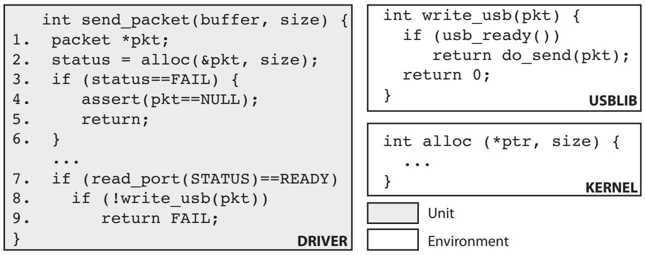
CFG Consistency (RC-CC): In the RC-CC model, the exploration engine is allowed to change any part of the system state, as long as the explored execution paths correspond to paths in the unit's inter-procedural control flow graph. This roughly corresponds to the consistency provided by static program analyzers that are dataflow-insensitive and analyze completely unconstrained paths. Being strictly weaker than the SC-SE model, though using the same information to explore new paths, the RC-CC model is complete.

The RC-CC model is useful in disassembling obfuscated and/or encrypted code: after letting the unit code decrypt itself under an LC model (thus ensuring the correctness of decryption), a disassembler can switch to the RC-CC model to reach high coverage of the decrypted code and quickly disassemble as much of it as possible.

3.2 Implementing Consistency Models

We now explain how the consistency models can be implemented by a selective symbolic execution engine (SSE), by describing the specifics of symbolic ↔ concrete conversion as execution goes from the unit to the environment and then back again.

We illustrate the different implementations with the example of a kernel-mode device driver (Fig. 4). The driver reads and writes from/to hardware I/O ports and calls the `write_usb` function, which is implemented in a kernel-mode USB library, as well as `alloc`, implemented by the kernel itself.



```
   int send_packet(buffer, size) {
1.  packet *pkt;
2.  status = alloc(&pkt, size);
3.  if (status==FAIL) {
4.     assert(pkt==NULL);
5.     return;
6.  }
    ...
7.  if (read_port(STATUS)==READY)
8.    if (!write_usb(pkt))
9.       return FAIL;
}
```

```
int write_usb(pkt) {
  if (usb_ready())
     return do_send(pkt);
  return 0;
}
```

```
int alloc (*ptr, size) {
  ...
}
```

Figure 4: Example of a "unit" (device driver) interacting with the "environment" (kernel-mode library and OS kernel itself).

3.2.1 Implementing Strict Consistency (SC)

Strictly Consistent Concrete Execution (SC-CE): For this model, an SSE allows only concrete input to enter the system. This leads to executing a single path through the unit and the environment. The SSE can execute the whole system natively without having to track or solve any constraints, because there is no symbolic data.

Strictly Consistent Unit-level Execution (SC-UE): To implement this model, the SSE converts all symbolic data to concrete values when the unit calls the environment. The conversion is consistent with the current set of path constraints in the unit. No other conversion is performed. The environment is treated as a black box, and no symbolic data can flow into it.

In the example of Fig. 4, the SSE concretizes the content of packet `pkt` when calling `write_usb` and, from there on, this soft constraint (see §2.2) is treated as a hard constraint on the content of `pkt`. The resulting paths trough the driver are globally feasible paths, but exploration is not complete, because treating the constraint as hard can curtail globally feasible paths during the exploration of the driver (e.g., paths that depend on the packet type).

Strictly-Consistent System-level Execution (SC-SE): Under SC-SE, the SSE lets symbolic data cross the unit/environment boundary, and the entire system is executed symbolically. This preserves global execution consistency.

Consider the `write_usb` function: This function gets its input from the USB host controller. Under strict consistency, the USB host controller (being "outside the system") can return a symbolic value, which in turn propagates through the USB library, eventually causing `usb_ready` to return a symbolic value as well.

Path explosion due to a large environment can make SC-SE hard to use in practice. The paths that go through the environment can substantially outnumber those that go through the unit, possibly delaying the exploration of interest. An SSE can heuristically prioritize the paths to explore, or employ *incremental symbolic execution* to execute parts of the environment as much as needed to discover interesting paths in the unit quicker. We describe this next.

The execution of `write_usb` proceeds as if it was executed symbolically, but only one globally feasible path is pursued in a depth-first manner, while all other forked paths are stored in a wait list. This simulates a concrete, single-path execution through a symbolically executing environment. After returning to `send_packet`, the path being executed carries the constraints that were accumulated in the environment, and symbolic execution continues in `send_packet` as if `write_usb` had executed symbolically. The return value x of `write_usb` is constrained according to the depth-first path pursued in the USB library, and so are the side effects. If, while executing `send_packet`, a branch that depends on x becomes infeasible due to the constraints imposed by the call to `write_usb`, the SSE returns to the wait list and resumes execution of a wait-listed path that, e.g., is likely to eventually execute line 9.

3.2.2 Implementing Local Consistency (LC)

For LC, an SSE converts, at the unit/environment boundary, the concrete values generated by the environment into symbolic values that satisfy the constraints of the environment's API. This enables multi-path exploration of the unit. In Fig. 4, SSE would turn `alloc`'s return value v into a symbolic value $\lambda_{\text{ret}} \in \{v, \texttt{FAIL}\}$ and `pkt` into a symbolic pointer, while ensuring that $\lambda_{\text{ret}} = \texttt{FAIL} \Rightarrow \texttt{pkt} = \textit{null}$, so that the `alloc` API contract is always upheld.

If symbolic data is written by the unit to the environment, the SSE must track its propagation. If a branch in the environment ever depends on this data, the SSE must abort that execution path, because the unit may have derived that data based on (symbolified) input from the environment that subsumed values the environment could not have produced in its state at the time.

From the driver's perspective, the global state may seem inconsistent, since the driver is exploring a failure path when no failure actually occurred. However, this inconsistency has no effect on the execution, as long as the OS does not make assumptions about whether or not buffers are still allocated after the driver's failure. LC would have been violated had the OS read the symbolic value of `pkt`, e.g., if the driver stored it in an OS data structure.

3.2.3 Implementing Relaxed Consistency (RC)

Overapproximate Consistency (RC-OC): In this model, the SSE converts concrete values at unit/environment interface boundaries into unconstrained symbolic values that disregard interface contracts. For example, when returning from `alloc`, both `pkt` and `status` become completely unconstrained symbolic values.

This model brings completeness at the expense of substantial overapproximation. No feasible paths are ever excluded from the symbolic execution of `send_packet`, but since `pkt` and `status` are completely unconstrained, there could be locally infeasible paths when exploring `send_packet` after the call to `alloc`.

As an example, note that `alloc` is guaranteed to set `pkt` to *null* whenever it returns `FAIL`, so the `assert` on line 4 should normally never fail. Nevertheless, under RC-OC, both `status` on line 3 and `pkt` on line 4 are unconstrained, so both outcomes of the `assert` statement are explored, including the infeasible one. Under stronger consistency models, like LC, `pkt` must be null if `status==FAIL`.

CFG Consistency (RC-CC): An SSE can implement RC-CC by pursuing all outcomes of every branch, regardless of path constraints, thus following all edges in the unit's inter-procedural CFG. Under RC-CC, exploration is fast, because branch feasibility need not be checked with a constraint solver. As mentioned earlier, one use case is a dynamic disassembler, where running with stronger consistency models may leave uncovered (i.e., non-disassembled) code. Implementing RC-CC may require program-specific knowledge, to avoid exploring non-existing edges, as in the case of an indirect jump pointing to an unconstrained memory location.

3.3 Consistency Models in Existing Tools

We now illustrate some of the consistency models by surveying example tools that implement such execution consistency.

Most dynamic analysis tools use the SC-CE model. Examples include Valgrind [38] and Eraser [33]. These tools execute and analyze programs along a single path, generated by user-specified concrete input values. Being significantly faster than multi-path exploration, analyses performed by such tools are, for instance, useful to characterize or explain program behavior on a small set of developer-specified paths (i.e., test cases). However, such tools cannot provide any confidence that results of the analyses extend beyond the concretely explored paths.

Dynamic test case generation tools usually employ either the SC-UE or the SC-SE models. For example, DART [18] uses SC-UE: it executes the program concretely, starting with random inputs, and then instruments the code to collect path constraints on each execution. DART uses these constraints to produce new concrete inputs that would drive the program along a different path on the next run. However, DART does not instrument the environment and hence cannot use information from it when generating new concrete inputs, thus missing feasible paths as indicated by SC-UE.

As another example, KLEE [11] uses either the SC-SE or a form of the SC-UE model, depending on whether the environment is modeled or not. In the former case, both the unit and the model of the environment are executed symbolically. In the latter case, whenever the unit calls the environment, KLEE executes the environment with concrete arguments. However, KLEE does not track the side effects of executing the environment, allowing them to propagate across otherwise independent execution paths, thus making the corresponding program states inconsistent. Due to this limitation, we cannot say KLEE implements precisely SC-UE as we defined it.

Static analysis tools usually implement some forms of the RC model. For example, SDV [2] converts a program into a boolean form, which is an over-approximation of the original program. Consequently, every path that is feasible in the original program would be found by SDV, but it also finds additional infeasible paths.

4. System Analysis with S^{2}E

S^{2}E is a platform for rapid prototyping of custom system analyses. It offers two key interfaces: the *selection* interface, used to guide the exploration of execution paths (and thus implement arbitrary consistency models), and the *analysis* interface, used to collect events or check properties of execution paths. Both interfaces accept modular selection and analysis plugins. Underneath the covers, S^{2}E consists of a customized virtual machine, a dynamic binary translator (DBT), and an embedded symbolic execution engine, as shown in Fig. 5. The DBT decides which guest machine instructions to execute concretely on the physical CPU vs. which ones to execute symbolically using the embedded symbolic execution engine.

S^{2}E provides many plugins out of the box for building custom analysis tools—we describe these plugins in §4.1. One can also extend S^{2}E with new plugins, using S^{2}E's developer API (§4.2).

4.1 User Interface

Path Selection: The first step in using S^{2}E is deciding on a policy for which part of a program to execute in multi-path (symbolic) mode vs. single-path (concrete) mode; this policy is encoded in a selector. S^{2}E provides a default set of selectors for the most common types of selection. They fall into three categories:

Data-based selection provides a way to expand an execution path into a multi-path execution by introducing symbolic values into the system; then, any time S^{2}E encounters a branch predicate involving a symbolic value, it will fork the execution accordingly. Symbolic data can enter the system from various sources, and S^{2}E provides a selector for each: *CommandLine* for symbolic command-line arguments, *Environment* for shell environment variables, *MSWinRegistry* for Microsoft Windows registry entries, etc.

Often it is useful to introduce a symbolic value at an internal interface. For example, say a server program calls a library function `libFn`(x) almost always with $x = 10$, but may call it with $x < 10$ in strange corner cases that are hard to induce via external workloads. The developer might therefore be interested in exploring the behavior of `libFn` for all values $0 \leq x \leq 10$. For such analyses, we provide an *Annotation* plugin, which allows direct injection of custom-constrained symbolic values anywhere they are needed.

Code-based selection enables/disables multi-path execution depending on whether the program counter is or not within a target code area; e.g., one might focus cache profiling on a web browser's SSL code, to see if it is vulnerable to side channel attacks. The *CodeSelector* plugin takes the name of the target program, library, driver, etc. and a list of program counter ranges. Each such range can be an inclusion or an exclusion range, indicating that code within that range should be explored in multi-path mode or single-path mode, respectively. *CodeSelector* is typically used in conjunction with data-based selectors to constrain the data-selected multi-path execution to within only code of interest.

Priority-based selection is used to define the order in which paths are explored within the family of paths defined with data-based and code-based selectors. S^{2}E includes basic ones, such as *Random*, *DepthFirst*, and *BreadthFirst*, as well as others. The *MaxCoverage* selector works in conjunction with coverage analyzers to heuristically select paths that maximize coverage. The *PathKiller* selector monitors the executed program and deletes paths that are determined to no longer be of interest to the analysis. For example, paths can be killed if a fixed sequence of program counters repeats more than n times; this avoids getting stuck in polling loops.

Path Analysis: Once the selectors define a family of paths, S^{2}E executes these paths and exposes each one of them to the analyzer plugins. One class of analyzers are bug finders, such as *DataRaceDetector* and *MemoryChecker*, which look for the corresponding bug conditions and output an execution path leading to the encountered bug. Another type of analyzer is *ExecutionTracer*, which selectively records the instructions executed along a path, along with the memory accesses, register values, and hardware I/O. Tracing can be used for many purposes, including measuring coverage offline. Finally, the *PerformanceProfile* analyzer counts cache misses, TLB misses, and page faults incurred along each path—this can be used to obtain the performance envelope of an application, and we describe it in more detail in the evaluation section (§6).

While most plugins are OS-agnostic, S^{2}E also includes a set of analyzers that expose Windows-specific events using undocumented interfaces or other hacks. For example, *WinDriverMon* parses and monitors OS-private data structures and notifies other plugins when the Windows kernel loads a driver. The *WinBugCheck* plugin catches "blue screen of death" events and kernel hangs.

4.2 Developer Interface

We now describe the interface that can be used to write new plugins or to extend the default plugins described above. Both selectors and analyzers use the same interface; the only distinction between selectors and analyzers is that selectors influence the execution of the program, whereas analyzers are passive observers. S^{2}E also allows writing of plugins that arbitrarily modify the execution state.

S^{2}E has a modular plugin architecture, in which plugins communicate via events in a publish/subscribe fashion. S^{2}E events are generated either by the S^{2}E platform or by other plugins. To register for a class of events, a plugin invokes *regEventX(callbackPtr)*; the event callback is then invoked every time *EventX* occurs. Callbacks have different parameters, depending on the type of event.

Table 2 shows the *core events* exported by S^{2}E that arise from regular code translation and execution. We chose these core events because they correspond to the lowest possible level of abstraction of execution: instruction translation, execution, memory accesses, and state forking. It is possible to build diverse state manipulation and analyses on top of them, as we will show in the evaluation.

The ExecState object captures the current state of the entire virtual machine *along a specific individual path*. It is the first parameter of every event callback. *ExecState* gives plugins read/write access to the entire VM state, including the virtual CPU, VM physical memory, and virtual devices. Plugins can also toggle multi-path execution and read/write VM memory and registers (see Table 3 for a short list of *ExecState* object methods). A plugin can obtain the PID of the running process from the page directory base register, can read/write page tables and physical memory, can change the control flow by modifying the program counter, and so on.

Super low level. Does this affect analysis?

For each path being explored, there exists a distinct *ExecState* object instance; when execution forks, each child execution receives its own private copy of the parent *ExecState*. Aggressive use of copy-on-write reduces the memory overhead substantially (§5).

Plugins partition their own state into per-path state (e.g., number of cache misses along a path) and global state (e.g., total number of basic blocks touched). The per-path state is stored in a *PluginState* object, which hangs off of the *ExecState* object. *PluginState* must implement a *clone* method, so that it can be cloned together with *ExecState* whenever S^{2}E forks execution. Global plugin state can live in the plugin's own heap.

The dynamic binary translator (DBT) turns blocks of guest code into corresponding host code; for each block of code this is typically done only once. During the translation process, a plugin may be interested in marking certain instructions (e.g., function calls) for subsequent notification. It registers for *onInstrTranslation* and, when notified, it inspects the *ExecState* to see which instruction is about to be translated; if it is of interest (e.g., a `CALL` instruction), the plugin marks it. Whenever the VM executes a marked instruction, it raises the *onInstrExecution* event, which notifies the registered plugin. For example, the *CodeSelector* plugin is implemented as a subscriber to *onInstrTranslation* events; upon receiving an event, it marks the instruction depending on whether it is or not an entry/exit point for a code range of interest. Having the *onInstrTranslation* and *onInstrExecution* events separate leverages the fact that each instruction gets translated once, but may get executed millions of times, as in the body of a loop. For most analyses, *onInstrTranslation* ends up being raised so rarely that using it introduces no runtime overhead (e.g., catching the kernel panic handler requires instrumenting only the first instruction of that handler).

onInstrTranslation	DBT is about to translate a machine instruction
onInstrExecution	VM is about to execute a marked instruction
onExecutionFork	S^{2}E is about to fork execution
onException	The VM interrupt pin has been asserted
onMemoryAccess	VM is about to execute a memory access

Table 2: Core events exported by the S^{2}E platform.

multiPathOn/Off()	Turn on/off multi-path execution
readMem(*addr*)	Read contents of memory at address *addr*
writeReg(*reg, val*)	Write *val* (symbolic or concrete) to *reg*
getCurBlock()	Get currently executing code block from DBT
raiseInterrupt(*irq*)	Assert the interrupt line for *irq*

Table 3: A subset of the *ExecState* object's interface.

S^{2}E opcodes are custom guest machine instructions that are directly interpreted by S^{2}E. These form an extensible set of opcodes for creating symbolic values (`S2SYM`), enabling/disabling multi-path execution (`S2ENA` and `S2DIS`) and logging debug information (`S2OUT`). They give developers the finest grain control over multi-path execution and analysis; they can be injected into the target code manually or using binary instrumentation tools like PIN [27]. In practice, opcodes are the easiest way to mark data symbolic and get started with S^{2}E, without involving any plugins.

The interface presented here was sufficient for all the multi-path analyses we attempted with S^{2}E. Selectors can enable or disable multi-path execution based on arbitrary criteria and can manipulate machine state. Analyzers can collect information about low-level hardware events all the way up to program-level events, they can probe memory to extract any information they need, and so on.

5. S^{2}E Prototype

The S^{2}E platform prototype (Fig. 5) reuses parts of the QEMU virtual machine [4], the KLEE symbolic execution engine [11], and the LLVM tool chain [25]. To these, we added 23 KLOC of C++ code written from scratch, not including third party libraries[2]. We added 1 KLOC of new code to KLEE and modified 1.5 KLOC; in QEMU, we added 1.5 KLOC of new code and modified 3.5 KLOC of existing code. S^{2}E currently runs on Mac OS X, Microsoft Windows, and Linux, it can execute any guest OS that runs on x86, and can be easily extended to other CPU architectures, like ARM or PowerPC. S^{2}E can be downloaded from `http://s2e.epfl.ch`.

S^{2}E explores paths by running the target system in a virtual machine and selectively executing small parts of it symbolically. Depending on which paths are desired, some of the system's machine instructions are dynamically translated within the VM into an intermediate representation suitable for symbolic execution, while the rest are translated to the host instruction set. Underneath the covers, S^{2}E transparently converts data back and forth as execution weaves between the symbolic and concrete domains, so as to offer the illusion that the full system (OS, libraries, applications, etc.) is executing in multi-path mode.

S^{2}E mixes concrete with symbolic execution in the same path by using a representation of machine state that is shared between the VM and the embedded symbolic execution engine. S^{2}E shares the

[2] All reported LOC measurements were obtained with SLOCCount [39].

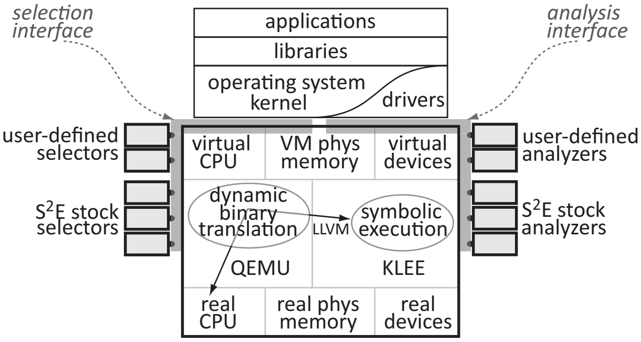


Figure 5: S^{2}E architecture, centered around a custom VM.

state by redirecting reads and writes from QEMU and KLEE to the common machine state—VM physical memory, virtual CPU state, and virtual device state. In this way, S^{2}E can transparently convert data between concrete and symbolic and provide distinct copies of the entire machine state to distinct paths. S^{2}E reduces the memory footprint of all these states using copy-on-write optimizations.

In order to achieve transparent interleaving of symbolic and concrete execution, we modified QEMU's DBT to translate the instructions that depend on symbolic data to LLVM and dispatch them to KLEE. Most instructions, however, run "natively"; this is the case even in the symbolic domain, because most instructions do not operate on symbolic state. We wrote an x86-to-LLVM back-end for QEMU, so neither the guest OS nor KLEE are aware of the x86 to LLVM translation. S^{2}E redirects all guest physical memory accesses, including MMIO devices, to the shared memory state object.

Besides VM physical memory, S^{2}E must also manage the internal state of the virtual devices when switching between execution paths. S^{2}E uses QEMU's snapshot mechanism to automatically save and restore virtual devices and CPU states when switching execution states. The shared representation of memory and device state between the concrete and symbolic domains enables S^{2}E to do on-demand concretization of data that is stored as symbolic. A snapshot can range from hundreds of MBs to GBs; we use aggressive copy-on-write to transparently share common state between snapshots of physical memory and disks. Some state need not be saved—for example, we do not snapshot video memory, so all paths share the same frame buffer. As an aside, this makes for intriguing visual effects on-screen: multiple erratic mouse cursors and BSODs blend chaotically, providing free entertainment to the S^{2}E user.

Interleaved concrete/symbolic execution and copy-on-write are transparent to the guest OS, so all guest OSes can run out of the box. Sharing state between QEMU and KLEE allows the guest to have a view of the system that is consistent with the chosen execution consistency model. It also makes it easy to replay execution paths of interest, e.g., to replay a bug found by a bug-detection analyzer.

Conversion from x86 to LLVM gives rise to complex symbolic expressions. S^{2}E sees a lower level representation of the programs than what would be obtained by compiling source code to LLVM (as done in KLEE): it actually sees the code that simulates the execution of the original program on the target CPU architecture. Such code typically contains many bitfield operations (such as `and`/`or`, `shift`, masking to extract or set bits in the `eflags` register).

We therefore implemented a bitfield-theory expression simplifier to optimize these expressions. We rely on the observation that, if parts of a symbolic variable are masked away by bit operations, removing those bits can simplify the corresponding expressions. First, the simplifier starts from the bottom of the expression (represented as a tree) and propagates information about individual bits whose value is known. If all bits in an expression are known, we replace it with the corresponding constant. Second, the simplifier propagates top-down information about bits that are ignored by the upper parts of the expression—when an operator modifies only bits that are ignored later, the simplifier removes that entire operation.

Symbolic expressions can also appear in pointers (e.g., as array indices or jump tables generated by compilers for switch statements). When a memory access with a symbolic pointer occurs, S^{2}E determines the pages referenced by the pointer and passes their contents to the constraint solver. Alas, large page sizes can bottleneck the solver, so S^{2}E splits the memory into small pages of configurable size (e.g., 128 bytes), so that the constraint solver need not reason about large areas of symbolic memory. In §6.2 we show how much this helps in practice.

Finally, S^{2}E must carefully handle time. Each system state has its own virtual time, which freezes when that state is not being run (i.e., is not in an actively explored path). Since running code symbolically is slower than native, S^{2}E slows down the virtual clock when symbolically executing a state. If it didn't do this, the (relatively) frequent VM timer interrupts would overwhelm execution and prevent progress. S^{2}E also offers an opcode to completely disable interrupts for a section of code, to further reduce the overhead.

6. Evaluation

S^{2}E's main goal is to enable rapid prototyping of useful, deep system analysis tools. In this vein, our evaluation of S^{2}E aims to answer three key questions: Is S^{2}E truly a general platform for building diverse analysis tools (§6.1)? Does S^{2}E perform these analyses with reasonable performance (§6.2)? What are the measured tradeoffs involved in choosing different execution consistency models on both kernel-mode and user-mode binaries (§6.3)? All reported results were obtained on a 2 × 4-core Intel Xeon E5405 2GHz machine with 20 GB of RAM, unless otherwise noted.

Can it actually be used for rapid prototyping'

6.1 Three Use Cases

We used S^{2}E to build three vastly different tools: an automated tester for proprietary device drivers (§6.1.1), a reverse engineering tool for binary drivers (§6.1.2), and a multi-path in-vivo performance profiler (§6.1.3). The first two use cases are complete rewrites of two systems that we built previously in an ad-hoc manner: RevNIC [13] and DDT [23]. The third tool is brand new.

Table 4 summarizes the productivity advantage we experienced by using S^{2}E compared to writing these tools from scratch. For these use cases, S^{2}E engendered two orders of magnitude improvement in both development time and resulting code volume. This justifies our efforts to create general abstractions for multi-path in-vivo analyses, and to centralize them into one platform.

Use Case	**Development Time** [person-hours]		**Tool Complexity** [lines of code]	
	from scratch	*with S^{2}E*	*from scratch*	*with S^{2}E*
Testing of proprietary device drivers	2,400	38	47,000	720
Reverse engineering of closed-source drivers	3,000	40	57,000	580
Multi-path in-vivo performance profiling	n/a	20	n/a	767

Table 4: Comparative productivity when building analysis tools from scratch (i.e., without S^{2}E) vs. using S^{2}E. Reported LOC include only new code written or modified; any code that was reused from QEMU, KLEE, or other sources is not included. For reverse engineering, 10 KLOC of offline analysis code is reused in the new version. For performance profiling, we do not know of any equivalent non-S^{2}E tool, hence the lack of comparison.

6.1.1 Automated Testing of Proprietary Device Drivers

We used S^{2}E to build DDT$^+$, a tool for testing closed-source Windows device drivers. This is a reimplementation of DDT [23], an ad-hoc combination of changes to QEMU and KLEE, along with handwritten interface annotations: 35 KLOC added to QEMU, 3 KLOC added to KLEE, 2 KLOC modified in KLEE, and 7 KLOC modified in QEMU. By contrast, DDT$^+$ has 720 LOC of C++ code, which glues together several exploration and analysis plugins, and provides the necessary kernel/driver interface annotations to implement LC.

DDT$^+$ combines several plugins: the *CodeSelector* plugin restricts multi-path exploration to the target driver, while the *MemoryCheck*, *DataRaceDetector*, and *WinBugCheck* analyzers look for bugs. To collect additional information about the quality of testing (e.g., coverage), we use the *ExecutionTracer* analyzer plugin. Additional checkers can be easily added. DDT$^+$ implements local consistency (LC) via interface annotations that specify where to inject symbolic values while respecting local consistency—examples of annotations appear in [23]. None of the reported bugs are false positives, indicating the appropriateness of local consistency for bug finding. In the absence of annotations, DDT$^+$ reverts to strict consistency (SC-SE), where the only symbolic input comes from hardware.

We run DDT$^+$ on two Windows network drivers, RTL8029 and AMD PCnet. DDT$^+$ finds the same 7 bugs reported in [23], including memory leaks, segmentation faults, race conditions, and memory corruption. Of these bugs, 2 can be found when operating under SC-SE consistency; relaxation to local consistency (via annotations) helps find 5 additional bugs. DDT$^+$ takes <20 minutes to complete testing of each driver and explores thousands of paths in each one.

For each bug found, DDT$^+$ outputs a crash dump, an instruction trace, a memory trace, a set of concrete inputs (e.g., registry values and hardware input) and values that where injected according to the LC model that trigger the buggy execution path.

While it is always possible to produce concrete inputs that would lead the system to the desired local state of the unit (i.e., the state in which the bug is reproduced) along a globally feasible path, the exploration engine does not actually do that while operating under LC. Consequently, replaying execution traces provided by DDT$^+$ usually requires replaying the symbolic values injected into the system during testing. Such replaying can be done in S^{2}E itself. Despite being only locally consistent, the replay is still effective for debugging: the execution of the driver during replay is valid and appears consistent, and injected values correspond to the values that the kernel could have passed to the driver under real, feasible (but not exercised) conditions.

S^{2}E generates crash dumps readable by Microsoft WinDbg. Developers can thus inspect the crashes using their existing tools, scripts, and extensions for WinDbg. They can also compare crash dumps from different execution paths to better understand the bugs.

6.1.2 Reverse Engineering of Closed-Source Drivers

We also built REV$^+$, a tool for reverse engineering binary Windows device drivers; it is a reimplementation of RevNIC [13]. REV$^+$ takes a closed-source binary driver, traces its execution, and then feeds the traces to an offline component that reverse engineers the driver's logic and produces new device driver code that implements the exact same hardware protocol as the original driver. In principle, REV$^+$ can synthesize drivers for any OS, making it easy to port device drivers without any vendor documentation or source code.

Adopting the S^{2}E perspective, we cast reverse engineering as a type of behavior analysis. As in DDT$^+$, the *CodeSelector* plugin restricts the symbolic domain to the driver's code segment. The *ExecutionTracer* plugin is configured to log to a file the driver's executed instructions, memory and register accesses, and hardware I/O. The already existing offline analysis tool from RevNIC then processes these traces to synthesize a new driver.

REV$^+$ uses overapproximate consistency (RC-OC). The goal of the tracer is to see each basic block execute, in order to extract its logic—full path consistency is not necessary. The offline trace analyzer only needs fragments of paths in order to reconstruct the original control flow graph—details appear in [13]. By using RC-OC, REV$^+$ sacrifices consistency in favor of obtaining coverage fast.

	RevNIC	REV$^+$	Improvement
PCnet	59%	66%	+7%
RTL8029	82%	87%	+5%
91C111	84%	87%	+3%
RTL8139	84%	86%	+2%

Table 5: Basic block coverage obtained by RevNIC and REV$^+$ in 1 hour.

We run REV$^+$ on the same drivers reported in [13], and REV$^+$ reverse engineers them with better coverage than RevNIC (see Table 5). Fig. 6 shows how coverage evolves over time during reverse engineering. Manual inspection of the reverse engineered code blocks reveals that the resulting drivers are equivalent to those generated by RevNIC, and thus to the originals too [13].

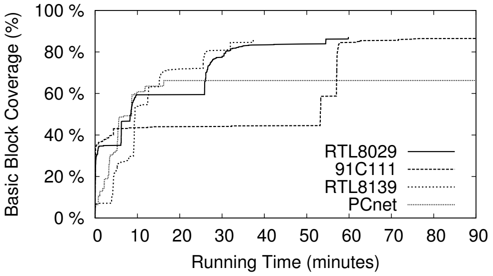


Figure 6: Basic block coverage over time for REV$^+$.

6.1.3 Multi-Path In-Vivo Performance Profiling

To further illustrate S^{2}E's generality, we used it to develop PROF$_S$, a multi-path in-vivo performance profiler and debugger. To our knowledge, such a tool did not exist previously, and this use case is the first in the literature to employ symbolic execution for performance analysis. In this section, we show through several examples how PROF$_S$ can be used to predict performance for certain classes of inputs. To obtain realistic profiles, performance analysis can be done under local consistency or any stricter consistency model.

PROF$_S$ allows users to measure instruction count, cache misses, TLB misses, and page faults for arbitrary memory hierarchies, with flexibility to combine any number of cache levels, size, associativity, line sizes, etc. This is a superset of the cache profiling functionality found in Valgrind [38], which can only simulate L1 and L2 caches, and can only measure cache misses.

For PROF$_S$, we developed the *PerformanceProfile* plugin. It counts the number of instructions along each path and, for memory reads/writes, it simulates the behavior of a desired cache hierarchy and counts hits and misses. For our measurements, we configured PROF$_S$ with 64-KB I1 and D1 caches with 64-byte cache lines and associativity 2, plus a 1-MB L2 cache that has 64-byte cache lines and associativity 4. The path exploration in PROF$_S$ is tunable, allowing the user to choose any execution consistency model.

The first PROF$_S$ experiment analyzes the distribution of instruction counts and cache misses for Apache's URL parser. In particular, we were interested to see whether there is any opportunity for a denial-of-service attack on the Apache web server via carefully constructed URLs. The analysis ran under local consistency for 9.5 hours and explored 5,515 different paths through the code. Of the 9.5 hours, 2.5 hours were spent in the constraint solver and 6 hours were spent running concrete code. In this experiment, the analysis carries high overhead, because it simulates a TLB and three caches.

We found each path involved in parsing a URL to take on the order of 4.3×10^6 instructions, with one interesting feature: for every additional "/" character present in the URL, there are 10 extra instructions being executed. We found no upper bound on the execution of URL parsing: a URL containing $n + k$ "/" characters will take $10 \times k$ more instructions to parse than a URL with n "/" characters. The total number of cache misses on each path was predictable at $15,984 \pm 20$. These are examples of behavioral insights one can obtain with a multi-path performance profiler. Such insights can help developers fine-tune their code or make it more secure (e.g., by ensuring that password processing time does not depend on the password content, to avoid side channel attacks).

We also set out to measure the page fault rate experienced by the Microsoft IIS web server inside its SSL modules while serving a static page workload over HTTPS. Our goal was to check the distribution of page faults in the cryptographic algorithms, to see if there is any opportunity for side channel attacks. We found no page faults in the SSL code along any of the paths, and only a constant number of them in gzip.dll. This suggests that counting page faults should not be the first choice if trying to break IIS's SSL encryption.

Next, we aimed to establish a performance envelope in terms of instructions executed, cache misses, and page faults for the ubiquitous `ping` program. This program has on the order of 1.3 KLOC. The performance analysis ran under local consistency, explored 1,250 different paths, and ran for 5.9 hours. Unlike the URL parsing case, almost 5.8 hours of the analysis were spent in the constraint solver—the first 1,000 paths were explored during the first 3 hours, after which the exploration rate slowed down.

The analysis does not find a bound on execution time, and it points to a path that could hit an infinite loop. This happens when the reply packet to `ping`'s initial packet has the record route (RR) flag set and the option length is 3 bytes, leaving no room to store the IP address list. While parsing the header, `ping` finds that the list of addresses is empty and, instead of `break`-ing out of the loop, it does `continue` without updating the loop counter. This is an example where performance analysis can identify a dual performance and security bug: malicious hosts could hang `ping` clients. Once we patched `ping`, we found the performance envelope to be 1,645 to 129,086 executed instructions. With the bug, the maximum during analysis had reached 1.5×10^6 instructions and kept growing.

PROF$_S$ can find "best case performance" inputs without having to enumerate the input space. For this, we modify slightly the *PerformanceProfile* plugin to track, for all paths being explored, the common lower bound on instructions, page faults, etc. Any time a path exceeds this minimum, the plugin automatically abandons exploration of that path, using the *PathKiller* selector described in §4. This type of functionality can be used to efficiently and automatically determine workloads that make a system perform at its best. This use case is another example of performance profiling that can only be done using multi-path analysis.

We wanted to compare our results to what a combination of existing tools could achieve: run KLEE to obtain inputs for paths through the program, then run each such test case in Valgrind (for multi-path analysis) and with Oprofile (for in-vivo analysis). This is not possible for `ping`, because KLEE's networking model does not support yet ICMP packets. It is not possible for binary drivers either, because KLEE cannot fork kernel state and requires source code. These difficulties illustrate the benefits of having a platform like S^{2}E that does not require models and can automatically cross back and forth the boundary between symbolic and concrete domains.

To conclude, we used S^{2}E to build a thorough multi-path in-vivo performance profiler that improves upon classic profilers. Valgrind [38] is thorough, but only single-path and not in-vivo. Unlike Valgrind-type tools, PROF$_S$ performs its analyses along multiple paths at a time, not just one, and can measure the effects of the OS kernel on the program's cache behavior and vice versa, not just the program in isolation. Although tools like Oprofile [30] can perform in-vivo measurements, but not multi-path, they are based on sampling, so they lack the accuracy of PROF$_S$—it is impossible, for instance, to count the exact number of cache misses in an execution. Such improvements over state-of-the-art tools come easily when using S^{2}E to build new tools.

Did they actually validate their result?

6.1.4 Other Uses of S^{2}E

S^{2}E can be used for pretty much any type of system-wide analysis. We describe here four additional ideas: energy profiling, hardware validation, certification of binaries, and privacy analysis.

First, S^{2}E could be used to profile energy use of embedded applications: given a power consumption model, S^{2}E could find energy-hogging paths and help the developer optimize them. Second, S^{2}E could serve as a hardware model validator: S^{2}E can symbolically execute a SystemC-based model [20] together with the real driver and OS; when there is enough confidence in the correctness of the hardware model, the modeled chip can be produced for real. Third, S^{2}E could perform end-to-end certification of binaries, e.g., verify that memory safety holds along all critical paths. Finally, S^{2}E could be used to analyze binaries for privacy leaks: by monitoring the flow of symbolic input values (e.g., credit card numbers) through the software stack, S^{2}E could tell whether any of the data leaks outside the system. S^{2}E alleviates the need to trust a compiler, since it performs all analysis on the final binary.

6.2 Implementation Overhead

S2E introduces ~6× runtime overhead over vanilla QEMU when running in concrete mode, and ~78× in symbolic mode. Concrete-mode overhead is mainly due to checks for accesses to symbolic memory, while the overhead in symbolic mode is due to LLVM interpretation and symbolic constraint solving.

The overhead of symbolic execution is mitigated in practice by the fact that the symbolic domain is much smaller than the concrete domain. For instance, in the `ping` experiments, S^{2}E executed 3×10^4 times more x86 instructions concretely than it did symbolically. All the OS code (e.g., page fault handler, timer interrupt, system calls) that is called frequently, as well as all the software that is running on top (e.g., services and daemons) are in concrete mode. Furthermore, S^{2}E can distinguish inside the symbolic domain instructions that can execute concretely (e.g., that do not touch symbolic data) and run them "natively." `ping`'s 4 orders of magnitude difference is a *lower* bound on the amount of savings selective symbolic execution brings over classic symbolic execution: by executing concretely those paths that would otherwise run symbolically, S^{2}E *also* saves the overhead of further forking (e.g., on branches inside the concrete domain) paths that are ultimately not of interest.

Another source of overhead are symbolic pointers. We compared the performance of symbolically executing the `unlink` utility's x86 binary in S^{2}E vs. symbolically executing its LLVM version in KLEE. Since KLEE recognizes all memory allocations performed by the program, it can pass to the constraint solver memory arrays of exactly the right size; in contrast, S^{2}E must pass entire memory pages. In 1 hour, with a 256-byte page size, S^{2}E explores 7,082 paths, compared to 7,886 paths in KLEE. Average constraint solving time is 0.06 sec for both. With 4 KB pages, though, S^{2}E explores only 2,000 states and averages 0.15 sec per constraint.

We plan to reduce this overhead in two ways: First, we can instrument the LLVM bitcode generated by S^{2}E with calls to the symbolic execution engine, before JITing it into native machine code, to avoid the overhead of interpreting each instruction in KLEE. This is similar in spirit to the difference between QEMU and the Bochs [6] emulator: the latter interprets instructions in one giant switch statement, whereas the former JITs them to native code and obtains a major speedup. Second, we plan to add support for directly executing native LLVM binaries inside S^{2}E, which would reduce significantly the blowup resulting from x86-to-LLVM translation and would reduce the overhead of symbolic pointers.

6.3 Execution Consistency Model Trade-Offs

Having seen the ability of S^{2}E to serve as a platform for building powerful analysis tools, we now experimentally evaluate the tradeoffs involved in the use of different execution consistency models. In particular, we measure how total running time, memory usage, and path coverage efficiency are influenced by the choice of models. We illustrate the tradeoffs using both kernel-mode binaries—the SMSC 91C111 and AMD PCnet network drivers—and a user-mode binary—the interpreter for the Lua embedded scripting language [26]. The 91C111 closed-source driver binary has 19 KB, PCnet has 35 KB; the symbolic domain consists of the driver, and the concrete domain is everything else. Lua has 12.7 KLOC; the concrete domain consists of the lexer+parser (2 KLOC) and the environment, while the symbolic domain is the remaining code (e.g., the interpreter). Parsers are the bane of symbolic execution engines, because they have many possible execution paths, of which only a small fraction are paths that pass the parsing/lexing stage [19]. The ease of separating the Lua interpreter from its parser in S^{2}E illustrates the benefit of selective symbolic execution.

We use a script in the guest OS to call the entry points of the drivers. Execution proceeds until all paths have reached the driver's `unload` entry point. We configure a selector plugin to exercise the entry points one by one. If S^{2}E has not discovered any new basic block for some time (60 sec), this plugin kills all paths but one. The plugin chooses the remaining path so that execution can proceed to the driver's next entry point.

Without path killing, drivers could get stuck in the early initialization phase, because of path explosion (e.g., the tree rooted at the initialization entry point may have several thousand paths when its exploration completes). The selector plugin also kills redundant subtrees when entry points return, because calling the next entry point in the context of *each* of these execution states (subtree leaves) would mostly exercise the same paths over again.

For Lua, we provide a symbolic string as the input program, under SC-SE consistency. Under local consistency, the input is concrete, and we insert suitably constrained symbolic Lua opcodes *after* the parser stage. Finally, in RC-OC mode, we make the opcodes completely unconstrained. We average results over 10 runs for each consistency model on a 4×6-core AMD Opteron 8435 machine, 2.6 GHz, 96GB of RAM. Table 6 shows running times for different execution consistencies.

Weaker (more relaxed) consistency models help achieve higher basic block coverage in that time—Fig. 7 shows results for the running times from Table 6. For PCnet, coverage varies between 14%–66%, while 91C111 ranges from 10%–88%. The stricter the model, the fewer sources of symbolic values, hence the fewer explorable paths and discoverable basic blocks in a given amount of time. In the case of our Windows drivers, system-level strict consistency (SC-SE) keeps all registry inputs concrete, which prevents several configuration-dependent blocks from being explored. In SC-UE, concretizing symbolic inputs to arbitrary values prevents the driver from loading, thus yielding poor coverage.

Consistency	91C111 Driver	PCnet Driver	Lua
RC-OC	1,400	3,300	1,103
LC	1,600	3,200	1,114
SC-SE	1,700	1,300	1,148
SC-UE	5	7	-

Table 6: Time (in seconds) to finish the exploration experiment for two device drivers and the Lua interpreter under different consistency models.

In the case of Lua, the local consistency model allows bypassing the lexer component, which is especially difficult to symbolically execute due to its loops and complex string manipulations. RC-OC exceptionally yielded less coverage because execution got stuck in complex crash paths reached due to incorrect Lua opcodes.

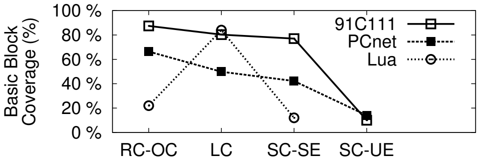


Figure 7: Effect of consistency models on coverage.

Path selection together with adequate consistency models improve memory usage (Fig. 8). Under LC, the PCnet driver spends 4 minutes in the initialization method, exploring ~7,000 paths and using 8 GB of memory. In contrast, it spends only 2 minutes (~2,500 paths) and 4 GB under RC-OC consistency. Under LC consistency, the `CardType` registry setting is symbolic, causing the initialization entry point to call in parallel several functions that look for different card types. Under LC consistency, S^{2}E explores these functions slower than under RC-OC consistency, where we liberally inject symbolic values to help these functions finish quicker. Slower exploration leads to less frequent timeout-based path kills, hence more paths, more memory consumption, and longer exploration times. Under SC-SE and SC-UE consistency, registry settings are concrete, thus exploring only functions for one card type.

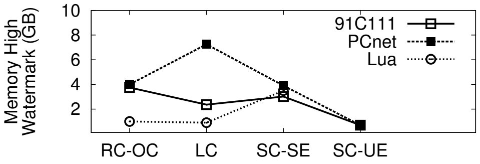


Figure 8: Effect of consistency models on memory usage.

Finally, consistency models affect constraint solving time (Fig. 9). The relationship between consistency model and constraint solving time often depends on the structure of the system being analyzed—generally, the deeper a path, the more complex the corresponding path constraints. For our targets, solving time decreases with stricter consistency, because stricter models restrict the amount of symbolic data. For 91C111, switching from local to overapproximate consistency increases solving time by $10\times$. This is mostly due to the unconstrained symbolic inputs passed to the `QueryInformationHandler` and `SetInformationHandler` entry points, which results in complex expressions being generated by switch statements. In Lua, the structure of the constraints causes S^{2}E to spend most of its time in the constraint solver.

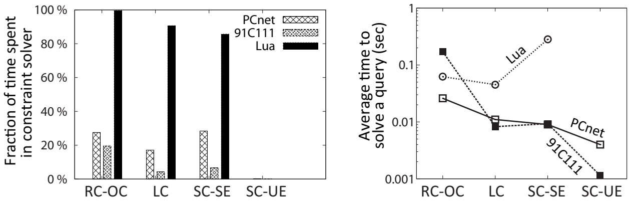


Figure 9: Impact of consistency models on constraint solving.

As in §6.1.3, we attempted a comparison to vanilla KLEE. We expected that the Lua interpreter, being completely in user-mode and not having any complex interactions with the environment, could be handled by KLEE. However, KLEE does not model some of its operations. For example, the Lua interpreter makes use of `setjmp` and `longjmp`, which turn into `libc` calls that manipulate the program counter and other registers in a way that confuses KLEE. Unlike S^{2}E, other engines do not have a unified representation of the hardware, so all these details must be explicitly coded for (e.g., detect that `setjmp` / `longjmp` is used and ensure that KLEE's view of the execution state is appropriately adjusted). In S^{2}E, this comes "for free," because the CPU registers, memory, I/O devices, etc. are shared between the concrete and symbolic domain.

Our evaluation shows that S^{2}E is a general platform that can be used to write diverse and interesting system analyses—we illustrated this by building, with little effort, tools for bug finding, reverse engineering, and comprehensive performance profiling. Consistency models offer flexible trade-offs between the performance, completeness, and soundness of analyses. By employing selective symbolic execution and relaxed execution consistency models, S^{2}E is able to scale these analyses to large systems, such as an entire Windows stack—analyzing real-world programs like Apache httpd, Microsoft IIS, and `ping` takes a few minutes up to a few hours, in which S^{2}E explores thousands of paths through the binaries.

7. Related Work

We are not aware of any platform that can offer the level of generality in terms of dynamic analyses and execution consistency models that S^{2}E offers. Nevertheless, a subset of the ideas behind S^{2}E did appear in various forms in earlier work.

BitBlaze [37] is the closest dynamic analysis framework to S^{2}E. It combines virtualization and symbolic execution for malware analysis and offers a form of local consistency to introduce symbolic values into API calls. In contrast, S^{2}E has several additional consistency models and various generic path selectors that trade accuracy for exponentially improved performance in more flexible ways. To our knowledge, S^{2}E is the first to handle all aspects of hardware communication, which consists of I/O, MMIO, DMA, and interrupts. This enables symbolic execution across the entire software stack, down to hardware, resulting in richer analyses.

One way to tackle the path explosion problem is to use models and/or relax execution consistency. File system models have allowed, for instance, KLEE to test UNIX utilities without involving the real filesystem [11]. However, based on our own experience, writing models is a labor-intensive and error-prone undertaking. Other researchers report that writing a model for the kernel/driver interface of a modern OS took several person-years [2].

Other bodies of work have chosen to execute the environment concretely, with various levels of consistency that were appropriate for the specific analysis in question, most commonly bug finding. For instance, CUTE [36] can run concrete code consistently without modeling, but it is limited to strict consistency and code-based selection. SJPF [32] can switch from concrete to symbolic execution, but does not track constraints when switching back, so it cannot preserve consistency in the general case.

Another approach to tackling path explosion is compositional symbolic execution [17]. This approach saves the results of exploration of parts of the program and reuses them when those parts are called again in a different context. We are investigating how to implement this approach in S^{2}E, to further improve scalability.

Non-VM based approaches cannot control the environment outside the analyzed program. For instance, both KLEE and EXE allow a symbolically executing program to call into the concrete domain (e.g., perform a system call), but they cannot fork the global system state. As a result, different paths clobber each other's concrete domain, with unpredictable consequences. Concolic execution [35] runs everything concretely and scales to full systems (and is not affected by state clobbering), but may result in lost paths when execution crosses program boundaries. Likewise, CUTE, KLEE, and other similar tools cannot track the branch conditions in the concrete code (unlike S^{2}E), and thus cannot determine how to redo calls in order to enable overconstrained but feasible paths.

In-situ model checkers [16, 21, 29, 40, 41] can directly check programs written in a common programming language, usually with some simplifications, such as data-range reduction, without requiring the creation of a model. Since S^{2}E directly executes the target binary, one could say it is an in-situ tool. However, S^{2}E goes further and provides a consistent separation between the environment (whose symbolic execution is not necessary) and the target code to be tested (which is typically orders of magnitude smaller than the rest). This is what we call in-vivo in S^{2}E: analyzing the target code in-situ, while facilitating its consistent interaction with that code's unmodified, real environment. Note that other work uses the "in vivo" term to mean something different from S^{2}E's meaning—e.g., Murphy et al. propose a technique for testing where "in vivo" stands for executing tests in production environments [28].

Several static analysis frameworks have been used to build analysis tools. Saturn [14] and `bddbddb` [24] prove the presence or absence of bugs using a path-sensitive analysis engine to decrease the number of false positives. Saturn uses function summaries to scale to larger programs and looks for bugs described in a logic programming language. `bddbddb` stores programs in a database as relations that can be searched for buggy patterns using Datalog. Besides detecting bugs, `bddbddb` helped optimizing locks in multithreaded programs. Static analysis tools rely on source code for accurate type information and cannot easily verify run-time properties or reason about the entire system. Both `bddbddb` and Saturn require learning a new language.

Dynamic analysis frameworks alleviate the limitations of static analysis tools. In particular, they allow the analysis of binary software. Theoretically, one could statically convert an x86 binary to, say, LLVM and run it in a system like KLEE, but this faces the classic undecidable problems of disassembly and decompilation [34]: disambiguating code from data, determining the targets of indirect jumps, unpacking code, etc.

S^{2}E adds multi-path analysis abilities to all single-path dynamic tools, while not limiting the types of analysis. PTLsim [42] is a VM-based cycle-accurate x86 simulator that selectively limits profiling to user-specified code ranges to improve scalability. Valgrind [38] is a framework best known for cache profiling tools, memory leak detectors, and call graph generators. PinOS [9] can instrument operating systems and unify user/kernel-mode tracers. However, PinOS relies on Xen and a paravirtualized guest OS, unlike S^{2}E. PTLsim, PinOS, and Valgrind implement cache simulators that model multi-level data and code cache hierarchies. S^{2}E allowed us to implement an equivalent *multi-path* simulator with little effort.

S^{2}E complements classic single-path, non VM-based profiling and tracing tools. For instance, DTrace [15] is a framework for troubleshooting kernels and applications on production systems in real time. DTrace and other techniques for efficient profiling, such as continuous profiling [1], sampling-based profiling [10], and data type profiling [31], trade accuracy for low overhead. They are useful in settings where the overhead of precise instrumentation is prohibitive. Other projects have also leveraged virtualization to achieve goals that were previously prohibitively expensive. These tools could be improved with S^{2}E by allowing the analyses to be exposed to multi-path executions.

S^{2}E uses mixed-mode execution as an optimization, to increase efficiency. This idea first appeared in DART [18], CUTE [36], and EXE [12], and later in Bitscope [8]. However, automatic bidirectional data conversions across the symbolic-concrete boundary did not exist previously, and they are key to S^{2}E's scalability.

To summarize, S^{2}E embodies numerous ideas that were fully or partially explored in earlier work. What is unique in S^{2}E is its generality for writing various analyses, the availability of multiple user-selectable (as well as definable) consistency models, automatic bidirectional conversion of data between the symbolic and concrete domains, and its ability to operate without any modeling or modification of the (concretely running) environment.

8. Conclusions

This paper described S^{2}E, a new platform for *in-vivo multi-path analysis* of systems, which scales even to large, proprietary, real-world software stacks, like Microsoft Windows. It is the first time virtualization, dynamic binary translation, and symbolic execution are combined for the purpose of generic behavior analysis. S^{2}E simultaneously analyzes entire *families of paths*, operates directly on *binaries*, and operates *in vivo*, i.e., includes in its analyses the entire software stack: user programs, libraries, kernel, drivers, and hardware. S^{2}E uses automatic bidirectional symbolic–concrete data conversions and relaxed execution consistency models to achieve scalability. We showed that S^{2}E enables rapid prototyping of a variety of system behavior analysis tools with little effort. S^{2}E can be downloaded from `http://s2e.epfl.ch/`.

Acknowledgments

We thank Jim Larus, our shepherd, and Andrea Arpaci-Dusseau, Herbert Bos, Johannes Kinder, Miguel Castro, Byung-Gon Chun, Petros Maniatis, Raimondas Sasnauskas, Willy Zwaenepoel, the S^{2}E user community, and the anonymous reviewers for their help in improving our paper. We are grateful to Microsoft Research for supporting our work through a PhD Fellowship starting in 2011.

References

[1] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S.-T. Leung, D. Sites, M. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *Symp. on Operating Systems Principles*, 1997.

[2] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *ACM SIGOPS/EuroSys European Conf. on Computer Systems*, 2006.

[3] T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg. The static driver verifier research platform. In *Intl. Conf. on Computer Aided Verification*, 2010.

[4] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conf.*, 2005.

[5] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2), 2010.

[6] Bochs IA-32 Emulator. http://bochs.sourceforge.net/.

[7] P. Boonstoppel, C. Cadar, and D. R. Engler. RWset: Attacking path explosion in constraint-based test generation. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[8] D. Brumley, C. Hartwig, M. G. Kang, Z. L. J. Newsome, P. Poosankam, D. Song, and H. Yin. BitScope: Automatically dissecting malicious binaries. Technical Report CMU-CS-07-133, Carnegie Mellon University, 2007.

[9] P. P. Bungale and C.-K. Luk. PinOS: a programmable framework for whole-system dynamic instrumentation. In *Intl. Conf. on Virtual Execution Environments*, 2007.

[10] M. Burrows, U. Erlingson, S.-T. Leung, M. T. Vandevoorde, C. A. Waldspurger, K. Walker, and W. E. Weihl. Efficient and flexible value sampling. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2000.

[11] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Systems Design and Implementation*, 2008.

[12] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Conf. on Computer and Communication Security*, 2006.

[13] V. Chipounov and G. Candea. Reverse engineering of binary device drivers with RevNIC. In *ACM SIGOPS/EuroSys European Conf. on Computer Systems*, 2010.

[14] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *Conf. on Programming Language Design and Implementation*, 2008.

[15] Dtrace. http://www.sun.com/bigadmin/content/dtrace/index.jsp.

[16] P. Godefroid. Model checking for programming languages using Verisoft. In *Symp. on Principles of Programming Languages*, 1997.

[17] P. Godefroid. Compositional dynamic test generation. In *Symp. on Principles of Programming Languages*, 2007. Extended abstract.

[18] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Conf. on Programming Language Design and Implementation*, 2005.

[19] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symp.*, 2008.

[20] IEEE. Standard 1666: SystemC language reference manual, 2005. http://standards.ieee.org/getieee/1666/.

[21] Java PathFinder. http://javapathfinder.sourceforge.net, 2007.

[22] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 1976.

[23] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *USENIX Annual Technical Conf.*, 2010.

[24] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Symp. on Principles of Database Systems*, 2005.

[25] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Intl. Symp. on Code Generation and Optimization*, 2004.

[26] Lua: A lightweight embeddable scripting language. http://www.lua.org/, 2010.

[27] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. PIN: building customized program analysis tools with dynamic instrumentation. In *Conf. on Programming Language Design and Implementation*, 2005.

[28] C. Murphy, G. Kaiser, I. Vo, and M. Chu. Quality assurance of software applications using the in vivo testing approach. In *Intl. Conf. on Software Testing Verification and Validation*, 2009.

[29] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Symp. on Operating Systems Design and Implementation*, 2008.

[30] Oprofile. http://oprofile.sourceforge.net.

[31] A. Pesterev, N. Zeldovich, and R. T. Morris. Locating cache performance bottlenecks using data profiling. In *ACM SIGOPS/EuroSys European Conf. on Computer Systems*, 2010.

[32] C. Păsăreanu, P. Mehlitz, D. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Intl. Symp. on Software Testing and Analysis*, 2008.

[33] 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*, 15(4), 1997.

[34] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Working Conf. on Reverse Engineering*, 2002.

[35] K. Sen. Concolic testing. In *Intl. Conf. on Automated Software Engineering*, 2007.

[36] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Symp. on the Foundations of Software Eng.*, 2005.

[37] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Intl. Conf. on Information Systems Security*, 2008.

[38] Valgrind. http://valgrind.org/.

[39] D. Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/, 2010.

[40] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MoDist: Transparent model checking of unmodified distributed systems. In *Symp. on Networked Systems Design and Implementation*, 2009.

[41] J. Yang, C. Sar, and D. Engler. EXPLODE: a lightweight, general system for finding serious storage system errors. In *Symp. on Operating Systems Design and Implementation*, 2006.

[42] M. T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *IEEE Intl. Symp. on Performance Analysis of Systems and Software*, 2007.