The S2E Platform: Design, Implementation, and Applications

VITALY CHIPOUNOV, VOLODYMYR KUZNETSOV, and GEORGE CANDEA,
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

This article presents S^2E, a platform for analyzing the properties and behavior of software systems, along with its use in developing tools for comprehensive performance profiling, reverse engineering of proprietary software, and automated testing of kernel-mode and user-mode binaries. Conceptually, S^2E is an automated path explorer with modular path analyzers: the explorer uses a symbolic execution engine to drive the target system down all execution paths of interest, while analyzers measure and/or check properties of each such path. S^2E users can either combine existing analyzers to build custom analysis tools, or they can directly use S^2E's APIs.

S^2E's strength is the ability to scale to large systems, such as a full Windows stack, using 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 *execution consistency models*, a way to make principled performance/accuracy trade-offs during analysis. These techniques give S^2E 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.

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

General Terms: Reliability, Verification, Performance, Security

Additional Key Words and Phrases: Symbolic execution, testing, analysis, profiling

ACM Reference Format:
Chipounov, V., Kuznetsov, V., and Candea G. 2012. The S2E platform: Design, implementation, and applications. ACM Trans. Comput. Syst. 30, 1, Article 2 (February 2012), 49 pages.
DOI = 10.1145/2110356.2110358 http://doi.acm.org/10.1145/2110356.2110358

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 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 and energy efficiency. For small

The authors are grateful to Google and Microsoft for generously supporting their work through a Google Faculty Award, a Google Focused Research Grant, and a Microsoft Research Ph.D. fellowship.
Authors' address: V. Chipounov, V. Kuznetsov, and G. Candea, School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne, Switzerland; email: George.candea@epfl.ch.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701, USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2012 ACM 0734-2071/2012/02-ART2 $10.00
DOI 10.1145/2110356.2110358 http://doi.acm.org/10.1145/2110356.2110358

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 also for large, complex, real systems.

We introduce in this article a platform that enables easy construction of analysis tools (such as performance profilers, 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 now explain these properties.

First, predictive analyses often must reason about entire families of paths through the target system, not just one path. A family of paths is a set of paths that have a specific property. 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 [Musuvathi et al. 2008] and symbolic execution [Cadar et al. 2008] to find bugs in real systems—these are all multipath analyses. One of our case studies demonstrates multipath 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 in a set constitute proofs over the corresponding set of executions; the guarantee provided by a proof is 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 [Cadar et al. 2006]. Most approaches abstract away the environment behind a model [Ball et al. 2006; Cadar et al. 2008], but writing abstract models is labor-intensive (taking in some cases multiple person-years [Ball et al. 2006]), models are rarely 100% accurate, and they tend to lose 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 multipath analysis consistent.

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 [Bessey et al. 2010]. 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, multipath, and operate on binaries. Going from single-path analysis to multipath analysis turns a linear problem into an exponential one, because the number of paths through a program generally increases at least exponentially in the number of branches—the "path explosion" problem [Boonstoppel et al. 2008]. It is therefore not feasible today

[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 experiments), 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.

to execute fully symbolically an entire software stack (programs, libraries, OS kernel, drivers, etc.) as would be necessary if we wanted consistent in-vivo multipath analysis.

We describe in this article S^2E, a general platform for developing multipath in-vivo analysis tools that are practical even for large, complex systems, such as an entire Microsoft Windows software stack. First, S^2E simultaneously exercises entire families of execution paths in a scalable manner by using *selective symbolic execution* and flexible *execution consistency models*. Second, S^2E 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 [Ball et al. 2010; Cadar et al. 2008; Godefroid et al. 2005; Musuvathi et al. 2008; Sen et al. 2005]. Third, S^2E 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.

The abstraction offered by S^2E is that of an automated path exploration mechanism with 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^2E glues together path selectors with path analyzers. *Selectors* guide S^2E'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^2E-provided default analyzers, or can be written from scratch using the S^2E API.

S^2E comes with ready-made selectors and analyzers that provide a wide range of analyses out of the box. The typical S^2E 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^2E virtual machine, and run the S^2E launcher in the guest OS, which starts the desired application and communicates with the S^2E 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^2E 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^2E via custom opcodes (described in Section 5). In the S^2E 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^2E 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^2E took us on the order of 20–40 person-hours and a few hundred LOC. To illustrate S^2E's generality, we present here three very different tools built using S^2E: a multipath in-vivo performance profiler, a reverse engineering tool, and a tool for automatically testing proprietary software.
To our knowledge, this article 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 framework for advantageously balancing over- and under-approximation of paths in an analysis-specific way;
— A *general platform* for performing diverse in-vivo multipath analyses in a way that scales to large real systems;
— The first use of *symbolic execution in performance analysis*.

In the rest of the article, we describe selective symbolic execution (Section 2), execution consistency models (Section 3), the S^2E prototype (Section 4), S^2E's APIs for developing analysis tools (Section 5), we evaluate the S^2E prototype (Section 6), survey related work (Section 7), and conclude (Section 8).

2. SELECTIVE SYMBOLIC EXECUTION

In designing S^2E, we were inspired by the successful use of symbolic execution [King 1975] in automated software testing [Cadar et al. 2008; Godefroid et al. 2005]. 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.

This superposition of paths is unfurled by a symbolic execution engine [Cadar et al. 2008] 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, that is, instead of setting a variable x to 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_j, 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_j}, 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 execution 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 typically used to make this process efficient.

S^2E 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^2E should split executions to explore the various paths, but whenever the program calls into some other part of the system, such as a library, multipath execution can cease and execution can revert to single-path. Then, when execution returns to the program, multipath execution must be resumed.

Multipath 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 multipath mode, the tree grows in width and depth; in single-path mode, the tree only grows in depth. We therefore say S^2E's exploration of program paths is *elastic*. S^2E turns multipath mode off (i.e., do not further expand existing paths) whenever possible, to minimize the size of the execution tree and include only paths that are of interest to the target analysis.

S^2E's elasticity of multipath exploration is key in being able to perform in-vivo multipath exploration of programs inside complex systems, like Microsoft Windows. By combining elasticity with virtualization, S^2E offers the illusion of symbolically executing a full software stack, while actually executing symbolically only select components.

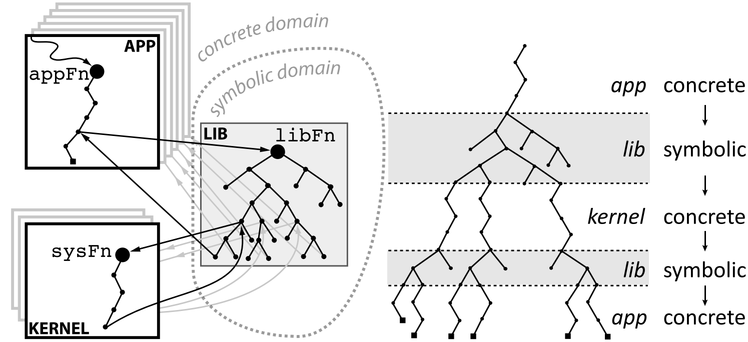

Fig. 1. Multipath/single-path execution: three different modules (left) and the resulting execution tree (right). Shaded areas represent the multipath (symbolic) execution domain, while the white areas are single-path.

For example, by concretely (i.e., nonsymbolically) 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*.

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 multipath (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 ($n : \lambda \rightarrow 5$), the family of executions is corseted. When a concrete value is converted to symbolic ($n : 5 \rightarrow \lambda$), 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 (Section 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 Section 3 we describe systematic and safe relaxations of execution consistency.

We now describe the mechanics of switching back and forth between multipath (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, while still preserving consistency of execution.

Figure 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 Figure 1) expands. After the execution returns again to the concrete domain, the execution tree is corseted and does not add any new paths, until execution returns to the symbolic domain again. Some paths may terminate earlier than others, e.g., due to hitting a crash bug in the program.

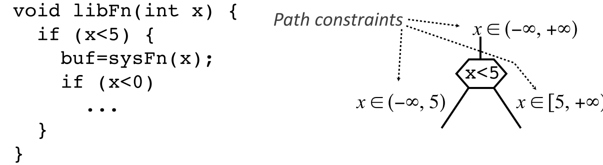

Fig. 2. The top level in *libFn*'s execution tree.

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 concrete → symbolic conversion is to use an S^{2}E selector to change the concrete arguments into symbolic ones, for instance, instead of *libFn(10)* call *libFn(*λ*)*. One can additionally opt to constrain λ, for instance, *libFn(* $\lambda \leq 15$ *)*.

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* has 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 multipath 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 other paths.

2.2. Symbolic → Concrete Transition

Dealing with the *libFn*→*sysFn* call is more complicated. Say *libFn* has the code shown in Figure 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 $x = 4$.

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 is about to branch on a condition that depends on 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 (nonlazy) 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 automatically 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 intimately tied to the $x = 4$ constraint. With this constraint, execution of *libFn* can continue, and correctness is 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^2E can track branch conditions in the concrete domain, which helps redo the call in a way that re-enables subsequent branches.

The “overconstraining” problem has two components: (a) the exclusion of paths (from the set of to-be-explored paths) that results directly from the concretization of x, and (b) the exclusion of paths that results indirectly via the constrained return value and side effects. Since S^2E implements VM state in a way that is shared between the concrete and symbolic domain (more details in Section 4), 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, that is, there exists a feasible concrete-execution path from the system’s start state to the system’s current state. However, there are many analyses for which this assumption is unnecessarily strong, and the cost of providing such strong consistency during multipath 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 exercised 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^2E aims to be a general platform for system analyses, so it provides the ability to choose the level of execution consistency that offers the best trade-offs. In this section, we take a first step toward systematically defining alternate execution consistency models (Section 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 (Section 3.2). In Section 3.3 we survey some of the ways in which consistency models appear in existing analysis tools.

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 during analysis. The appropriateness of the trade-off is determined by the nature of the analysis, that is, by the way in which feasibility of different paths affects completeness and soundness of the analysis.

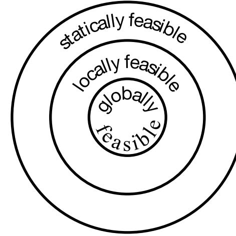

Diagram A. Venn diagram representing the relationship between globally, locally, and statistically feasible execution paths.

In the rest of the article, 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 Diagram A, 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 (by definition) to tell apart locally feasible from globally feasible paths.

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

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 Figure 3 using a representation corresponding to the Venn diagram (Diagram A). Their completeness and consistency are summarized in Table I. We invite the reader to follow Figure 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 the SC model. 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 knowledge of its internals 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

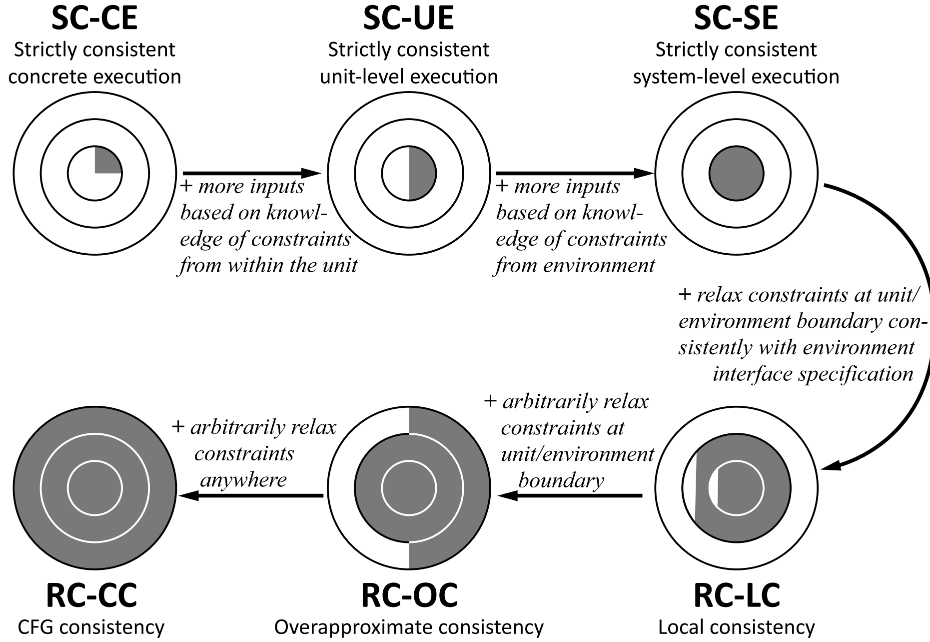

Fig. 3. Different execution consistency models cover different sets of feasible paths. The SC-CE model corresponds to concrete execution. The SC-UE and SC-SE models are obtained from the previous ones by using increasingly more information about the system, to explore increasingly bigger sets of concrete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

Table I.

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

Model	Consistency	Completeness	Use Case
SC-CE	consistent system-wide	incomplete	Single-path profiling/testing of units that have a limited number of paths
SC-UE	consistent system-wide	incomplete	Analysis of units that generate hard-to-solve constraints (e.g., cryptographic code)
SC-SE	consistent system-wide	complete	Sound and complete verification without false positives or negatives; testing of tightly coupled systems with fuzzy unit boundaries.
RC-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

blindly guessing new inputs. Classic fuzzing (random input testing) [Miller et al. 1990] 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, that is, 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 [Cadar et al. 2006, 2008; Godefroid et al. 2005]. Such tools usually instrument only the program but not the operating system code (sometimes such tools replace parts of the OS by models, effectively adding a simplified version of parts of the environment to 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 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 found eventually 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. Relaxed Consistency (RC). Under relaxed consistency (RC), all paths through the unit are admitted, even those that are not allowed by the SC 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 RC-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 three subcategories of the RC model, both of which are useful in practice.

Local Consistency (RC-LC). The local consistency (RC-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 operating system, 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 without having to find concrete inputs to the overall system that would enable those paths. This, however, introduces global inconsistency; for instance, according to the specification of the `write` system call, there exists no concrete execution in which (non-zero) `count` bytes are written to the file and `write` 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.

In other words, the RC-LC model allows for inconsistencies in the environment, while keeping the state of the unit internally consistent. To preserve RC-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 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 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 RC-LC model. For this reason, we call the RC-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 outside the unit. This means that the RC-LC model is not complete. In practice, the less a unit interacts with its environment, the fewer such paths are aborted or missed.

Using the RC-LC model in practice requires writing annotations for API functions called by the unit under analysis. An anotation specifies how to turn concrete values into symbolic ones, like in the case of the `write` system call described earlier. Writing such annotations is fairly straightforward, in contrast to writing environment models, where one must specify the complete behavior of the API function (and this makes both the writing of environment models and the ensuring of their correctness and completeness in the face of code evoluation hard [Ball et al. 2006]).

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 [Kuznetsov et al. 2010] 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 Chipounov and Candea [2010].

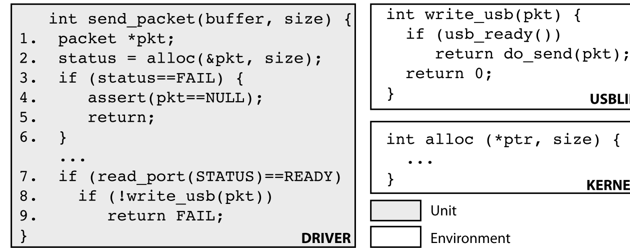

Fig. 4. Example of a “unit” (device driver) interacting with its “environment” (kernel-mode library and OS kernel itself). The unit is shaded.

CFG Consistency (RC-CC). The RC-CC model admits any execution paths, as long as they 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 paths that are completely unconstrained. 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 RC-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.

To summarize, we presented six different consistency models that offer flexible trade-offs between false positives, false negatives, and performance (Table I). The SC-CE model has zero false positives but yields many false negatives because it explores a tiny fraction of the paths in the system. The SC-UE model reduces the number of false negatives and SC-SE eliminates them at the expense of high exploration cost. Relaxed consistency models alleviate this high cost by allowing inconsistencies. This allows the RC-LC model to explore paths through the unit that would be followed by some concrete execution without actually incurring the cost of finding such paths. The RC-OC model introduces further inconsistencies to guarantee zero false negatives at the expense of introducing false positives. Finally, RC-CC completely unconstrains execution to speed up path exploration.

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 $\leftrightarrow$ concrete conversion as execution goes from the unit to the environment and then back again.

We illustrate the implementation details with an example of a kernel-mode device driver (Figure 4). The driver reads/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.

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 Figure 4, the SSE concretizes the content of packet `pkt` consistently with the path constraints when calling `write_usb` and, from there on, this soft constraint (see Section 2.2) is treated as a hard constraint on the content of `pkt`. The resulting paths through 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 external input from the USB host controller via the `usb_ready` function. 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, for instance, is deemed to eventually execute line 9.

3.2.2. Implementing Relaxed Consistency (RC).

Local Consistency (RC-LC). For RC-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 multipath exploration of the unit. Referring to Figure 4, the SSE would turn `alloc`'s return value v into a symbolic value $\lambda_{\mathrm{ret}} \in \{v, \texttt{FAIL}\}$ and `pkt` into a symbolic pointer, while ensuring that $\lambda_{\mathrm{ret}} = \texttt{FAIL} \Rightarrow \texttt{pkt} = null$, so that the `alloc` API contract is satisfied.

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 symbolic 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. RC-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).

Overapproximate Consistency (RC-OC). In this model, the SSE converts concrete values at the 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` promises 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 models, like RC-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 nonexisting edges, as in the case of an indirect jump pointing to an unconstrained memory location.

3.3. Consistency Models in Existing Tools

Some of these consistency models already appear in existing tools; we survey them here as a way to further explain S^2E's consistency models.

Most dynamic analysis tools use the SC-CE model. Examples include Valgrind [Valgrind 2011] and Eraser [Savage et al. 1997]. These tools execute and analyze programs along a single path, generated by user-specified concrete input values. Being significantly faster than multipath 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 [Godefroid et al. 2005] uses SC-UE: it executes the program concretely, starting with random inputs, while collecting 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, which is characteristic of SC-UE.

As another example, KLEE [Cadar et al. 2008] 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 defined here.

Static analysis tools usually implement some form of the RC model. For example, SDV [Ball et al. 2006] 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 the tool also finds additional infeasible paths.

4. S^2E PROTOTYPE

We implemented selective symbolic execution in the S^2E prototype. This system is meant to be a platform for rapid prototyping of custom system/program analyses that employ various execution consistency models. S^2E offers two key interfaces, one for path selection and one for analysis; we describe these interfaces further in Section 5. S^2E explores paths by running the target system in a virtual machine and selectively executing selected parts of it symbolically. Depending on which parts are desired, some of the machine instructions of the system being analyzed 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^2E 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 multipath mode.

Figure 5 shows the S^2E architecture. We reused parts of the QEMU virtual machine [Bellard 2005], the KLEE symbolic execution engine [Cadar et al. 2008], and the LLVM tool chain [Lattner and Adve 2004]. To these, we added 30 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^2E currently runs on Mac OS X, Microsoft Windows, and Linux, it can execute any guest OS that runs on x86 or ARM (e.g., Android OS), and can be easily extended to other CPU architectures, like MIPS or PowerPC. S^2E can be downloaded from `http://s2e.epfl.ch`.

In the rest of this section, we explain how S^2E uses dynamic binary translation (Section 4.1), how the execution engine handles concretely and symbolically running code (Section 4.2), and the details of the plugin infrastructure (Section 4.3). Finally, we conclude the section with some of the optimizations that are key to making the illusion of whole-system symbolic execution feasible (Section 4.4).

4.1. Dynamic Binary Translation

In this part of the section, we first discuss the general idea behind dynamic binary translation used in QEMU. Then, we explain how S^2E extends QEMU to interface dynamic translation with KLEE.

The dynamic binary translator (DBT) converts at run-time the executable code of one platform to executable code of another. For example, QEMU can run x86 Windows on a MIPS machine by translating the x86 code to the MIPS instruction set. QEMU works in a loop: it continuously fetches blocks of guest code, translates them to the host's instruction set, and passes the resulting translation to the execution engine. The DBT determines which code to fetch and translate by reading the state of the

[2]All reported LOC measurements were obtained with SLOCCount [Wheeler 2010].

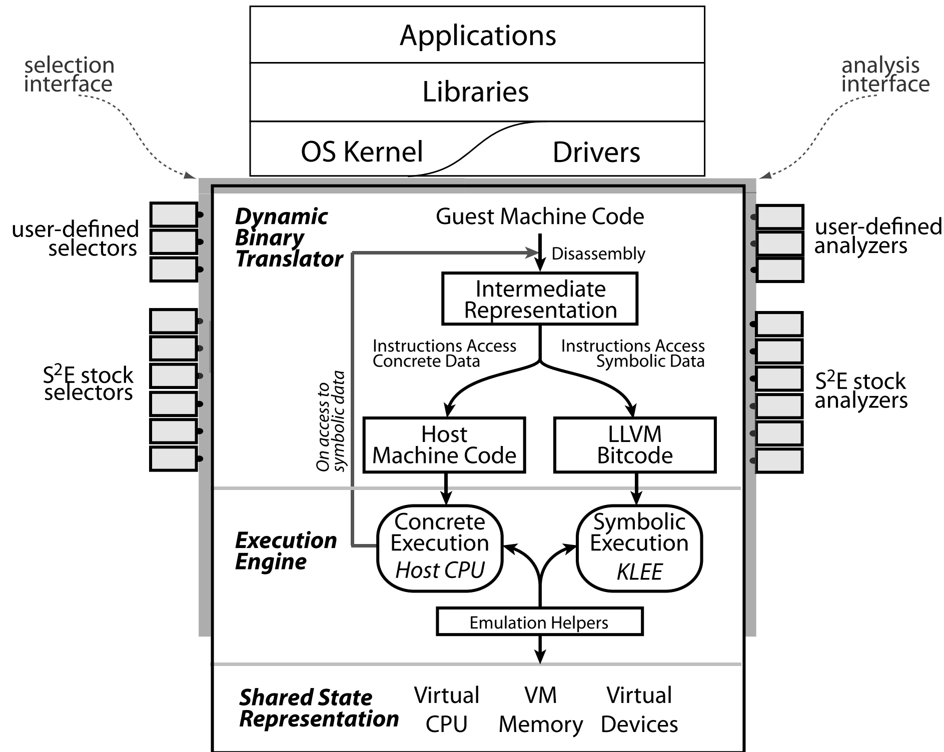

Fig. 5. S^{2}E architecture, centered around a custom VM. The VM dynamically dispatches guest machine instructions to the host CPU for native execution, or to the symbolic execution engine for symbolic interpretation, depending on whether the instruction accesses symbolic data or not. The system state is shared between the code running natively and the code interpreted in the symbolic execution engine; this enables S^{2}E to achieve the level of execution consistency desired by the user.

virtual CPU and the guest memory (i.e., the current program counter in the CPU and program code stored in memory). This state is updated as part of the execution of the translated code.

First, the DBT translates guest instructions to *microoperations*, the QEMU intermediate representation (IR). Microoperations split complex guest instructions into simpler operations that are easier to emulate. Consider the x86 instruction `inc [eax]`, which increments the value in the memory location whose address is stored in the `eax` register. The DBT decomposes this instruction into a memory load to a temporary register, an increment of that register, and a subsequent memory store to the original location.

The DBT packages microoperations into *translation blocks*. A translation block contains a sequence of microoperations up to and including the first microoperation that modifies the control flow, such as a branch, a call, or a return. The translator cannot add to the translation block the instructions past the control flow change, because the translator cannot always determine statically at which code location to continue the translation process.

The DBT transforms the microoperations forming the translation block into machine instructions of the host instruction set by turning each microoperation into an equivalent sequence of host instructions, using a code dictionary that maps microoperations to host instructions. Most of the conversions consist of a one-to-one mapping

from a microoperation to the corresponding machine instruction. For more complex instructions, like those that manipulate the processor's control state or that access memory, the DBT emits a microoperation that calls emulation helpers (which are C functions that emulate the original guest machine instruction). Some helpers are big and are used frequently (e.g., for memory accesses), therefore inlining them in the translated code would be prohibitively expensive in terms of generated code size.

Microoperations simplify the translation process by abstracting away the guest's instruction set. Without such an IR, translation would require a different translator for every pair of guest and host instruction sets. For example, supporting 8 guest platforms and 4 hosts would require 32 translators. In contrast, the use of an IR requires only 8 *frontends* (to transform the guest code to the IR) and 4 *backends* (to convert the IR to the host's instruction set). QEMU comes with many frontends, including Alpha, ARM, x86, Microblaze, Motorola 68K, MIPS, PowerPC, and SPARC. Backends include ARM, x86, MIPS, SPARC, PowerPC, and PowerPC 64.

We extended the native QEMU backend and wrote a new x86-to-LLVM backend to interface S^2E with the KLEE symbolic execution engine that interprets LLVM instructions. The DBT invokes the LLVM backend when at least one microoperation in the translation block references registers with symbolic content. If all register references are concrete, the DBT uses the native backend. We now explain how we modified the native back-end to accommodate symbolic execution, and how our new LLVM backend works.

4.1.1. Translating to Native Code. We extended QEMU's native x86-64 backend to run S^2E on 64-bit Windows hosts. First, since Windows uses a different calling convention from Linux, supporting Windows required us to change how the DBT allocates host CPU registers in order to pass parameters to QEMU helper functions. Second, we also turned pointers to `long`s into `uintptr_t`s, because `long`s are always 32-bit on Windows, even in 64-bit mode. Third, since native code cannot manipulate symbolic data, we inserted a check that switches to symbolic execution mode when a symbolic value is detected (Section 4.2).

4.1.2. Translating to LLVM. We added the LLVM backend to QEMU, in order to interface S^2E with the KLEE symbolic execution engine. This backend translates microoperations to the LLVM intermediate representation, which is directly interpretable by KLEE (see Section 4.2). Neither the guest OS nor KLEE are aware of the x86-to-LLVM translation: The guest OS sees that its instructions are being executed, and KLEE only sees LLVM instructions, just as if they were coming from a program entirely compiled to LLVM. In this way, the guest thinks the "entire world" is concrete, while KLEE thinks the "entire world" is symbolic.

The DBT must translate code in a way that allows precise CPU exception handling, given that execution could be interrupted at any time by hardware interrupts, page faults, etc. S^2E extends the DBT to enable precise exception handling from LLVM code. When an exception occurs, QEMU converts the address of the *translated* instruction that raised the exception to the program counter of the *guest* code. Such a conversion is possible because each guest instruction corresponds to a clearly delimited sequence of host machine instructions. However, there is no such clear correspondence in LLVM code, because LLVM applies more aggressive optimizations within each translation block. To solve this, we modified the DBT to insert microoperations that explicitly update the program counter before each guest instruction is executed. As a result, both the LLVM code and the native code see a consistent program counter at every point during execution, allowing precise exception handling.

4.2. Execution Engine

We now present the extensions to QEMU's execution engine that enable transparent switching between concrete and symbolic execution, while preserving the consistency of the execution state.

The execution engine consists of a loop that calls the DBT to translate guest code, then runs the translated code in native mode or interprets it in the symbolic execution engine. The execution engine does not know a priori whether to ask the DBT to generate LLVM or native code. It first instructs the DBT to generate native code and, if the code reads memory locations that contain symbolic data, it invokes the DBT to retranslate the code to LLVM. The DBT stores the translations in a cache to avoid needless retranslations, such as when executing a loop or repeatedly calling the same function).

S^2E mediates access to most of the VM state via *emulation helpers*. While simple instructions can access the CPU state directly, memory accesses, device I/O, as well as complex manipulations of the CPU state go through specific helpers, in order to reduce the size of the translated code. For example, the translated code for the software interrupt instruction triggers the `do_interrupt` helper during execution. This helper emulates the behavior of the software interrupt instruction by checking the current execution mode and privilege level, saving registers, taking care of potential exceptions, etc.

S^2E provides emulation helpers both for concrete (native) execution and symbolic (LLVM) execution. Native-mode helpers mediate access to the shared state when S^2E executes concrete code on the host CPU, while LLVM helpers are used in symbolic execution mode. The execution engine runs native-mode helpers on the host CPU and interprets LLVM helpers in the symbolic execution engine; LLVM helpers must sometimes call native-mode QEMU code, for example to simulate a virtual device.

LLVM emulation helpers avoid forceful unnecessary concretizations that would arise from calling native emulation helpers from within KLEE. The emulation helpers, called by the translated code, are compiled twice: to x86 and to LLVM. When running in symbolic mode, KLEE executes the LLVM version of the helper in order to let the helper manipulate symbolic data. If that version was missing, KLEE would be forced to call the native x86 version of the helper, which would then forcefully concretize the symbolic data. For example, QEMU implements bit-shift operations in helpers; if the bit-shift helper was available in x86 form only, the data it manipulates would have to be concretized when called from KLEE.

4.2.1. S^2E's Shared State Representation. S^2E combines concrete with symbolic execution in a controlled fashion along the same path by using a representation of machine state that is shared between the VM and the embedded symbolic execution engine. S^2E redirects reads and writes from QEMU and KLEE to the common machine state, which consists of VM physical memory, virtual CPU state, and virtual device state (see Figure 6). In this way, S^2E can transparently convert data between concrete and symbolic, according to the desired consistency model, and provide distinct copies of the entire machine state to distinct paths. S^2E reduces the memory footprint of all these system states by several orders of magnitude through copy-on-write (Section 4.4).

S^2E implements transparent state sharing by using KLEE's `ObjectState` data structure for the CPU and the physical memory. This structure encapsulates an array of concrete bytes and symbolic expressions. It provides accessors to get and set concrete or symbolic bytes. To execute native code more efficiently, S^2E extends `ObjectState` to expose a direct pointer to the concrete array of bytes, bypassing getters and setters. It also exposes a pointer to a bitmap that indicates which bytes are symbolic and which are concrete.

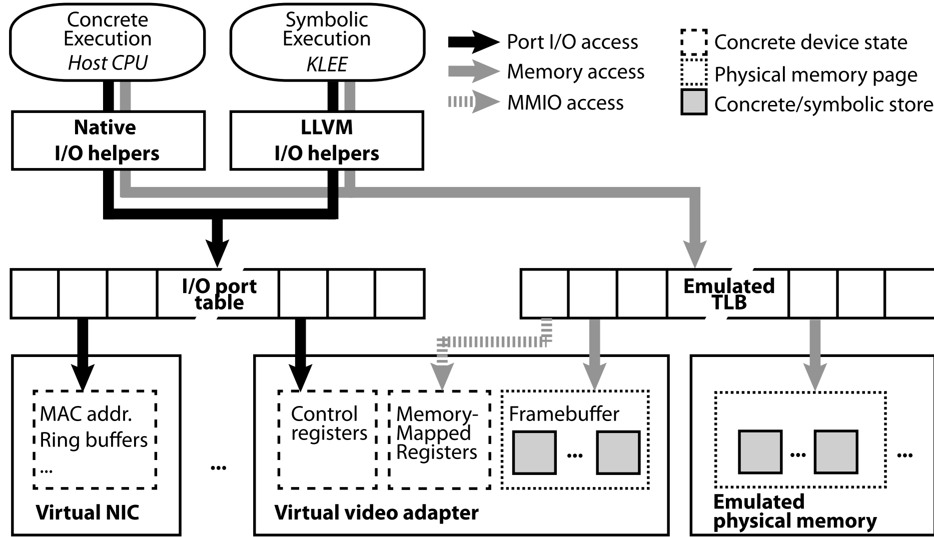

Fig. 6. Data paths to shared memory and device state.

Sharing the CPU State. S^2E splits the CPU state into symbolic and concrete regions, each in a different `ObjectState` structure. The *symbolic region* contains the general purpose registers and the flags registers. These registers can store symbolic values. The *concrete region* stores the control state of the system, including segment registers, program counter, control and debug registers, etc. S^2E does not allow this state to become symbolic, because doing so would cause execution to fork inside the S^2E emulation code, thus exercising the emulator and not the target software. For example, a symbolic `PE` (protection mode) bit in the `CR0` register would fork the translator excessively often, since many instructions behave differently in protected mode vs. in real mode.

S^2E concretizes all symbolic data written to the concrete region. For example, S^2E concretizes symbolic addresses when they are written to the (always concrete) program counter. To avoid reducing the completeness of exploration too much, S^2E actually allows execution to fork up to some predefined number of times, and then concretizes the program counter in each of the states. This behavior can be customized by the user, via the S^2E API. Finally, S^2E assigns floating point registers to the concrete region, because KLEE and the underlying constraint solver[3] do not yet support floating point operations on symbolic data.

The translated code accesses the CPU state directly by dereferencing the pointer to the CPU state or, in the case of native helpers, indirectly: read accesses to the symbolic CPU state are prepended with checks for symbolic data.

Sharing the Memory State. QEMU emulates a memory management unit (MMU) to handle all guest memory accesses. The MMU translates virtual memory addresses into physical addresses. The MMU caches the result of the translation in a TLB to speed up the translation on repeated accesses to the same memory pages. In QEMU, the TLB is a direct-mapped cache where each entry holds an offset that, when added to the virtual address of a memory page, results in the physical address inside QEMU's

[3]The constraint solver decides whether path constraints associated with each branch outcome are satisfiable and, if so, allows the symbolic execution engine to continue execution along that outcome.

address space where the data for that page is stored. Each TLB entry also contains information about access permissions and whether the memory page belongs to an emulated device.

S^{2}E extends the TLB with pointers to `ObjectState` structures in order to support symbolic memory. The `ObjectState` structures store the actual concrete and/or symbolic data of the memory pages.[4] When native code is running, the MMU checks whether memory reads would return symbolic data by looking at the `ObjectState`'s bitmap. If yes, the MMU instructs the execution engine to abort the execution of the current translation block and to re-execute the memory access in symbolic mode. In symbolic mode, the execution engine retrieves the `ObjectState` that corresponds to the physical address stored in the TLB entry and proceeds with the memory operation.

Sharing the Device State. A device is a piece of hardware that interacts with the CPU via a bus. To the CPU, devices look like memory: the CPU writes/reads data to/from them. Devices continuously observe the address bus. When the CPU issues a write access, if a connected device recognizes the address, it reads the data sent on the bus and processes it (e.g., displays a pixel on the screen). Likewise, when the CPU issues a read access, the device that recognizes its address on the bus replies with the requested value (e.g., a network packet). Unlike regular memory, which always returns the value last stored in that location, a device may return different data on each access. For example, it is common for devices to stream data packets over a single address, where consecutive reads from that address return successive bytes of the packet.

A device performs operations on state and produces output visible to the machine the device is attached to. In real hardware, the state consists of the contents of all internal registers (stored in flip-flops) and memory (e.g., DRAM chips). Virtual devices in QEMU emulate the behavior of the real devices: device state is kept in host memory, and the device's functionality is implemented by software running on the host CPU. QEMU supports both memory-mapped (MMIO) and port I/O-based devices.

S^{2}E modifies the QEMU *block layer* to support consistent disk state. Virtual block devices (e.g., hard disks and CD drives) provide storage to guests, which is backed by files stored on the host. Virtual block devices access the host files via the QEMU block layer. S^{2}E modifies the block layer to redirect to a state-local buffer all writes to the host files. When the guest OS issues a read request, S^{2}E returns the latest write from the buffer. If there were no writes, S^{2}E forwards the request to the block layer. This ensures that all execution states see a consistent disk state and do not clobber each other's writes by writing to a shared disk file. Failing to provide consistent disk state quickly leads to file system corruption, resulting in guest OS crashes.

4.2.2. Symbolic Hardware. To support whole-system symbolic execution, S^{2}E extends the virtual hardware with symbolic devices (e.g., to enable analysis of low-level code such as device drivers) and introduces a per-state virtual clock, to ensure that the guest sees a coherent time.

Symbolic Devices. A symbolic device is a special device that discards all writes, returns a symbolic value on every read, and triggers symbolic interrupts[5]; in other words, it does not implement specific functionality. S^{2}E instruments port I/O, MMIO, and physical memory accesses (for DMA-d memory) in order to determine on which read to return a symbolic value. To support symbolic reads for port I/O and MMIO, S^{2}E extends QEMU's emulation helpers. If a given port belongs to a symbolic device,

[4] One memory page can be split in multiple `ObjectState` structures, in order to optimize access times, as shown in Section 4.4.
[5] A symbolic interrupt is an interrupt with a symbolic arrival time.

S^{2}E returns a symbolic value on reads and discards writes. MMIO helpers are similar: each TLB entry contains a flag that specifies whether the memory page is mapped to physical memory or to a device, and is directed to the device emulation helpers as needed. These helpers return symbolic values on reads, exactly like for port I/O. To handle DMA, when the TLB entry of a memory page involved in a DMA transfer is loaded, S^{2}E modifies the flag in order to invoke MMIO emulation helpers whenever this memory page is accessed; in these helpers, if the access indeed falls inside the DMA region, a symbolic value is returned.

Supporting symbolic interrupts does not require any modification to QEMU. Triggering such interrupts consists of asserting the interrupt pin of the virtual device at the desired moment. This can be readily done by QEMU, which has different mechanisms to assert interrupts for each class of devices (e.g., for PCI, ISA, and USB devices). At which point in an execution to trigger the interrupt is decided by the S^{2}E plugins.

These mechanisms enable selection plugins to implement arbitrary symbolic devices. S^{2}E comes with a *SymbolicHardware* plugin that implements symbolic PCI and ISA devices. For ISA devices, the plugin registers port I/O ranges, MMIO, and DMA regions according to the user's configuration. For PCI devices, the plugin lets the user specify the device and vendor identifiers, as well as I/O and MMIO regions, interrupt channels, and all other fields available in a PCI descriptor. The plugin uses this information to instantiate an "impostor" PCI device that will induce the guest OS to load the appropriate device driver. Then, whenever the driver accesses the device, S^{2}E returns symbolic data.

Enabling DMA regions and symbolic interrupts is done with support from analysis plugins: they monitor the OS kernel, catch invocations of DMA-related APIs (e.g., registration of DMA regions), and pass address ranges to selector plugins (e.g., *SymbolicHardware*) that then register these regions through the S^{2}E API. Likewise, an analysis plugin can help determine when to trigger symbolic interrupts. For example, DDT$^+$, an automated testing tool for proprietary drivers, triggers such interrupts on every call to the kernel API in order to maximize the chances of exposing concurrency bugs. REV$^+$, a reverse engineering tool, triggers symbolic interrupts after having exercised the send entry point of a network card, in order to maximize the coverage of the interrupt handler.

Per-State Virtual Clock. QEMU maintains two types of clocks: a host clock and a virtual clock. The *host clock* reflects the current time of the host machine. The host clock is used by QEMU's virtual real-time clock device in order to provide the guest OS with a time source synchronized with the host machine. The *virtual clock* stores the number of ticks elapsed from the start of the system (i.e., when the VM was turned on). Unlike the host clock, the virtual clock is periodically incremented but not synchronized with the host machine's time.

Since S^{2}E splits "reality" into multiple executions, it must correspondingly offer multiple timelines. For this reason, S^{2}E maintains a separate virtual clock for each system state and does not rely on the host clock. S^{2}E increments the virtual clock of the state of the currently running path and keeps the respective clocks frozen in all other states. This way, the guest OS is given (a sufficiently good) illusion that the execution of those paths never stopped.

S^{2}E slows down the per-state virtual clock when running in symbolic mode. Interpreting LLVM instructions in KLEE is slower than running native code, and frequent timer interrupts make progress even slower. To avoid frequent interruptions when interpreting blocks of LLVM code, S^{2}E temporarily masks the timer interrupts and restores them after it finishes interpreting the LLVM code.

4.2.3. State Switching. Conceptually, S^2E executes only one path at a time and switches between paths to allow executions to progress in parallel. Since each execution path is characterized by its state, S^2E switches execution paths by switching states. The challenge is to automatically save and restore QEMU-specific *concrete* state (i.e., virtual devices and concrete CPU state) as well as to manage the translation block cache correspondingly.

S^2E explicitly copies the *concrete* region of the CPU state to/from QEMU's heap. Before S^2E is initialized, QEMU allocates a `CPUState` structure on the heap. Although S^2E stores the CPU state in an `ObjectState` structure, which LLVM helpers and symbolically running code access transparently, parts of QEMU also directly access the concrete region by dereferencing `CPUState` pointers (e.g., from the DBT). Finding and instrumenting all accesses to redirect them to the `ObjectState` is error-prone and unmaintainable (e.g., when upgrading QEMU versions). Therefore, S^2E leaves all the accesses unchanged (i.e., lets QEMU access the `CPUState` on the heap) and, during state switch, S^2E saves the concrete content on the heap in the `ObjectState` of the active execution state, fetches the new state, and overwrites the structure on the heap with the new `CPUState` data.

S^2E relies on QEMU's snapshot mechanism to automatically save and restore concrete virtual device data structures. QEMU uses snapshots to suspend and resume the virtual machine: S^2E redirects all writes and reads to/from the snapshot file to a per-path buffer. When S^2E is about to switch states, it calls QEMU to go through the list of all virtual devices and save their internal data structures. Then, S^2E selects the next execution state and restores the state of the virtual devices by calling `vmstate_load`.

Users can configure S^2E to not preserve the per-path device state upon state switching and let devices share their state between all execution paths, as done in KLEE. This causes inconsistencies, but reduces memory usage. For example, disabling state saving for the framebuffer avoids recording a separate `ObjectState` (multiple MBs) for each state and copying this data between the heap and the `ObjectState`. This makes for intriguing visual effects on-screen: multiple erratic mouse cursors and BSODs blend chaotically, providing free entertainment to the S^2E user.

Since different states may execute different code at the same address, stale code might end up being executed if the translation block cache is not flushed on state switches. However, since many programs do not change their code at run-time, disabling flushing makes sense, since it improves emulation speed. We plan to make the translation block cache state-local, to avoid unnecessary flushes.

4.3. Plugin Infrastructure

The S^2E plugin infrastructure connects selector and analyzer plugins via events, as will be described in Section 5. An S^2E plugin is a C++ class that subclasses the `Plugin` base class, which in turn registers the plugin with S^2E, automatically checks that plugin dependencies are satisfied, and provides an API to retrieve the instance of other plugins in order to communicate with them. During initialization, a plugin must subscribe to at least one core event; it can also subscribe to events exported by other plugins. A plugin can later modify its event subscriptions from its event handlers.

We wrote an event library that defines *signals* (the S^2E events) to which it is possible to connect *callbacks* (the event subscribers). We originally used the `libsigc++` [Pulkkinen et al. 2011] library for the plugin infrastructure, but it incurs an unacceptable performance overhead, because it calls memory allocation routines during signal invocation. S^2E plugins can trigger signals at a high rate (up to thousands of signal invocations per second). For example, it took 250 seconds to open the Windows control

panel while using the *FunctionMonitor* plugin (12 seconds without the plugin). The new implementation reduced the overhead to 25% (15 seconds).

S^{2}E instruments translated code to generate run-time events. For each guest instruction that the DBT translates, S^{2}E invokes the *onInstrTranslation* event, described in Section 5. One parameter of this event is a pointer to a list of callbacks. Subscribers that want to be notified every time that a guest instruction is executed append their callback to that list. After S^{2}E processed all subscribers of *onInstrTranslation*, S^{2}E saves the list of *onInstrExecution* callbacks in the translation block and inserts a microoperation that triggers the invocation of a specific emulation helper every time that instruction is executed. This emulation helper goes through the list stored in the translation block and invokes the callbacks.

S^{2}E extends the x86 instruction set with custom instructions that trigger events. S^{2}E uses the opcode `0x0f 0x3f` for custom instructions, which is unused according to the Intel instruction set manual [Intel 2011]. In S^{2}E, this opcode is followed by an 8-bytes operand that is freely definable by the plugins. The DBT translates this opcode into a call to the S^{2}E custom instruction emulation helper and passes the operand as a parameter. At runtime, the helper invokes all the callbacks registered by the subscribers of the *onCustomInstruction* event, the subscribers check the operand and perform whatever action is appropriate. Note that executing on a normal machine a program instrumented with S^{2}E opcodes would trigger an invalid instruction exception.

S^{2}E triggers all other events without requiring the translated code to be instrumented. For example, S^{2}E triggers the *onTimer* event from QEMU's timer handler in order to allow plugins to process periodic events. Likewise, S^{2}E triggers *onException*, *onExecutionFork*, and *onTlbMiss* from the exception emulation helpers, KLEE, and the MMU, respectively.

4.4. Key Optimizations

In this section, we describe four optimizations that have brought the greatest improvement in S^{2}E's performance: pervasive use of copy-on-write, aggressive simplification of symbolic expressions, optimized handling of symbolic pointers, and multicore parallelization.

Copy-on-Write. Copy-on-write (COW) minimizes memory usage by sharing as much data as possible between execution states. When a state is copied (upon path splitting), the child states share the data stored in the parent. When a write occurs, S^{2}E copies the data from the parent to the child that initiated the write. S^{2}E splits the physical memory into multiple `ObjectState` structures and then reuses KLEE's COW mechanisms. S^{2}E uses a similar technique for device state as well, but does not use fine-grained COW, because device state is small (a few kilobytes per state).

Expression Simplification. 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 to set bits in the `eflags` register).

We built a bitfield-theory expression simplifier to optimize these expressions that, if parts of a symbolic variable are masked away by bit operations, removes those bits from the corresponding expressions. First, the simplifier starts from the bottom of the expression's tree representation and propagates information about individual bits whose value is known. If all bits in an expression are known, S^{2}E replaces

the expression 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 only modifies bits that upper parts ignore, the simplifier removes that entire operation.

We say a bit in an expression is known to be one (respectively, zero), when that bit is not symbolic and has the value one (respectively, zero). For example, if x is a 4-bit symbolic value, the expression $x \mid 1000$ has its most significant bit (MSb) known to be one, because the result of an `or` of a concrete bit set to one and of a symbolic bit is always one. Moreover, this expression has no bits known to be zero, because the MSb is always one and symbolic bits `or`-ed with a zero remain symbolic. Finally, the ignore mask specifies which bits are ignored by the upper part of an expression. For example, in $1000 \ \& \ (x \mid 1010)$, the ignore mask at the top-level expression is 0111 because the `and` operator cancels the three lower bits of the entire expression.

To illustrate, consider the 4-bit wide expression $0001 \ \& \ (x \mid 0010)$. The simplifier starts from the bottom (i.e., $x \mid 0010$) and propagates up the expression tree the value $k_{11} = 0010$ for the known-one bits as well as $k_{10} = 0000$ for the known-zero bits. This means that the simplifier knows that bit 1 is set but none of the bits are zero for sure (because x is symbolic). At the top level, the `and` operation produces $k_{21} = 0000$ for the known-one bits ($k_{11} \ \& \ 0001$) and $k_{20} = 1110$ for the known-zero bits ($k_{10} \mid 1110$). The simplifier now knows that only the least significant bit matters and propagates the ignore mask $m = 1110$ top down. There, the simplifier notices that 0010 is redundant and removes it, because $1101 \mid m$ yields 1111, meaning that all bits are ignored. The final result is thus $1 \ \& \ x$.

We implemented this simplification in the early stage of expression creation rather than in the constraint solver. This way, we do not have to resimplify the same expressions again when they are sent to the constraint solver several times (for example, as part of path constraints). This is an example of applying domain-specific logic to reduce constraint solving time; we expect our simplifier to be directly useful for KLEE as well, when testing programs that use bitfields heavily.

Symbolic Pointers. A symbolic pointer is a pointer whose value depends on symbolic inputs, therefore referring to a range of memory locations (as opposed to a concrete pointer, which refers to only one particular address). Symbolic pointers commonly occur when indexing arrays, for instance, in jump tables generated by compilers for switch statements. When a symbolic pointer is dereferenced, S^2E 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^2E splits the default 4KB-sized memory pages into smaller pages of configurable size (e.g., 128 bytes), so that the constraint solver need not reason about large areas of symbolic memory. In Section 6.2, we show how much this helps in practice.

S^2E can also concretize symbolic pointers to further reduce overhead. This is most useful in the case of switch statements and symbolic writes to the program counter register (which is always concrete in S^2E). S^2E uses binary search to determine to which interval the symbolic pointer belongs, and forks n states, each state having one concrete address that satisfies the path constraints. n is usually bounded, since the path constraints often limit the interval (e.g., switch statements have a limited number of cases). n can be user-configurable to avoid path explosion in case the symbolic pointer references a large memory range.

Multicore S^2E. S^2E explores different paths concurrently by running multiple S^2E instances in parallel. Whenever an execution path splits due to a branch depending on a symbolic condition, S^2E assigns the exploration of the newly created path to a new

S^{2}E instance that runs on a different core. If all cores are already busy exploring paths, then the S^{2}E instance behaves like in single-instance mode: each split path is added to the local queue of the instance that split it. When an S^{2}E instance has explored all the paths in its queue, it terminates, leaving the core available for new instances.

The simple parallelization algorithm used by S^{2}E does not address the issues of redundant exploration (i.e., two cores exploring identical states) and load balancing (i.e., moving a subset of states from one instance to another). This can be solved by combining S^{2}E with the Cloud9 [Bucur et al. 2011] parallel symbolic execution engine. Section 6.1 analyzes the impact of this multicore design on S^{2}E's performance.

S^{2}E uses the `fork` system call to run instances on multiple processors/cores. This system call maps naturally to the concept of execution path splitting in symbolic execution. Consider an execution path p that is explored by an S^{2}E process q. When p splits on a branch that depends on a symbolic value, S^{2}E creates a path p' and forks a child process q', which is an identical copy of the S^{2}E process q. The child process q' receives the execution path p', and the parent process q continues the execution of p. After the fork system call completes, each instance starts exploring an independent subtree. A similar approach is used by EXE [Cadar et al. 2006] to implement symbolic execution.

S^{2}E plugins can be kept aware of the various running instances: S^{2}E triggers *onInstanceFork* whenever it creates a new instance. For example, the *Logger* plugin listens to this event to create a fresh execution trace file each time a new instance is created; this avoids expensive synchronization, yet writing traces to separate files does not burden offline processing tools: each file contains an independent subtree, and recreating the full tree through trace concatenation is straightforward.

4.5. Summary

In this section, we showed how S^{2}E combines virtualization, dynamic binary translation, native execution, and symbolic interpretation to give the illusion of whole-system symbolic execution. We explained how S^{2}E shares CPU, memory, and device state between native and symbolic execution, described how to efficiently implement the plugin infrastructure, and presented some of the key optimizations that make the S^{2}E approach feasible. We describe next how S^{2}E can be used to write new analysis tools.

5. SYSTEM ANALYSIS WITH S^{2}E

As mentioned before, S^{2}E is in effect 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 was described in the previous section. The DBT decides which guest machine instructions to execute concretely vs. which ones to interpret 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 Section 5.1. One can also extend S^{2}E with new plugins, using S^{2}E's developer API (Section 5.2). Figure 7 shows a snapshot of the S^{2}E plugins that are part of S^{2}E today.

5.1. User Interface

In this section, we show how an S^{2}E user can combine the various S^{2}E plugins in order to construct custom analysis tools, without writing any additional plugins.

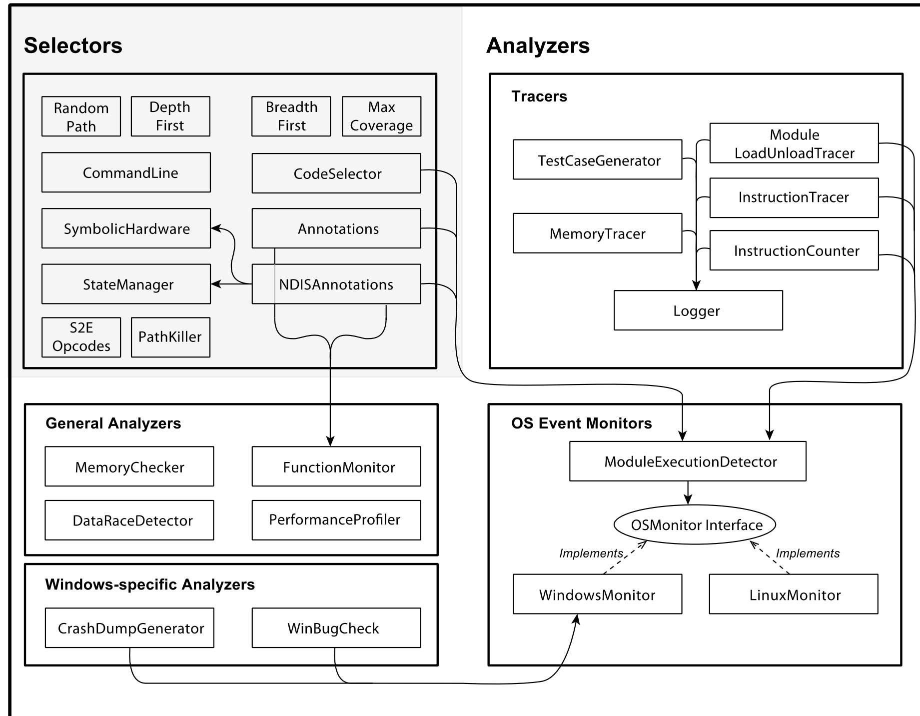

Fig. 7. S^{2}E stock plugins. The arrows represent plugin dependencies (e.g., the *CodeSelector* plugin uses the functionality provided by *ModuleExecutionDetector*). To distinguish selectors from analyzers, we show the former on a shaded background.

For instance, an S^{2}E user could be a device driver tester performing quality checks during driver development. Such a user would combine various path selectors to limit multipath exploration to the driver under test (Section 5.1.1) with path analysis plugins to check the driver for the presence of bugs (Section 5.1.2).

5.1.1. Path Selection. The first step in using S^{2}E is deciding on a policy for which part of a program to execute in multipath (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, which fall into three categories:

Data-based selection provides a way to expand a regular execution into a multipath one by introducing symbolic values into the system; then, any time S^{2}E encounters a branch predicate that depends on symbolic values, execution will fork accordingly. Symbolic data can enter the system from various sources, and S^{2}E provides a selector for each, ranging from command-line arguments using the *CommandLine* plugin to hardware input with the *SymbolicHardware* plugin.

Often it is useful to introduce a symbolic value at an interface that is internal to the system. 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, S^{2}E provides an *Annotations* plugin, which allows direct injection of custom-constrained symbolic values anywhere they are needed.

Code-based selection enables/disables multipath execution depending on whether the program counter is or not within a target code area; for instance, 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 (or library, driver, etc.) and a list of program counter ranges. Code within these ranges should be explored in multipath mode, while code that is outside should be run in single-path mode. *CodeSelector* is typically used together with data-based selectors to constrain the data-selected multipath 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 policies, 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.

5.1.2. Path Analysis. Once the family of paths to be analyzed is defined via the choice of selector(s), the user needs to choose the analyzer(s) to which S^{2}E will expose the chosen paths.

An analyzer is a piece of logic that checks for properties along execution paths. For example, a bug finder is an analyzer that may check for various types of crashes or assertion violations along the executed paths. A performance profiler is also a type of analyzer that checks for properties such as the number of cache misses along a path or the TLB hit count. S^{2}E has several multipath analysis plugins, such as a performance profiler, memory checkers, crash detectors, or execution tracers.

S^{2}E also lets users take advantage of existing unmodified off-the-shelf single-path analysis tools, such as Valgrind, Oprofile [Levon and Elie 1998], or Microsoft Driver Verifier [Microsoft 2011a], which can be wrapped into a plugin that adapts the output of these tools to the S^{2}E API (e.g., such a plugin could terminate every path reported as faulty by Microsoft's Driver Verifier).

Reusing Existing Single-Path Analysis Tools. We illustrate how S^{2}E can reuse existing tools with the example of Valgrind and Oprofile. Valgrind instruments programs in order to analyze their cache behavior, memory safety, execution times, and call graphs. Oprofile is a sampling-based profiler that can analyze an entire software stack, including the OS kernel.

Both of these tools are single-path and rely on testers to guess the concrete inputs that would drive a program down a path of interest. In other words, testers have to design test cases that exhibit the behaviors to be studied, such as bugs or slowdowns. In contrast, S^{2}E automatically enumerates various execution paths and exposes these tools to them. Off-the-shelf tools are not aware of the multipath exploration: they just operate as usual, without any modification in their logic, but ultimately yield multipath results.

However, S^{2}E does not remove the limitations of the existing tools. For example, Valgrind is still limited to profiling user-mode processes and cannot analyze the kernel, while Oprofile is still subject to imprecise results because it is based on sampling. Moreover, these tools remain single-path in nature: they cannot reason about multiple paths at a time. For instance, Valgrind cannot tell whether the path it has just run has the lowest instruction count. For this, a Valgrind-based plugin would need to be modified to look at all the paths that were explored so far.

Multipath Analyzers. When off-the-shelf tools are not enough, users can employ S^{2}E analysis plugins. Plugins run outside of the guest OS and can observe the entire system state, without interfering with the software under analysis.

One class of analyzers are bug finders, such as the *WinBugCheck* and *MemoryChecker* plugins, which look for Windows "blue screens of death" and memory errors, and output the execution paths leading to the encountered bugs. Another class of analyzers are execution tracers, such as *InstructionTracer*, which selectively records the instructions executed along a path, or *MemoryTracer*, which logs memory accesses and hardware I/O. Tracing can be used for many purposes, like offline coverage measurement or profiling. Finally, the *PerformanceProfiler* 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. We describe it in more detail in the evaluation section (Section 6).

While most plugins are OS-agnostic, S^{2}E also includes a set of analyzers that intercept Windows-specific events using undocumented interfaces or other hacks. For example, *WindowsMonitor* parses and monitors Windows kernel data structures and notifies other plugins when the kernel loads a driver, a library, or an application. Another example is the *CrashDumpGenerator* plugin, which generates a memory dump compatible with Microsoft WinDbg.

Offline Multipath Analyzers. S^{2}E provides plugins for collecting execution traces and saving them to a file for offline analysis. This is useful for complex analyses that are hard to do online, such as reverse engineering of device drivers (Section 6.1.2). The core tracing plugin *Logger* provides an interface to other plugins that they can use to log arbitrary data. *Logger* wraps the written data into a trace item object containing a path identifier, timestamp, size, and the plugin that wrote the item. This allows offline analysis tools to focus only on trace items of interest.

The collected traces can be parsed by offline analysis tools to reconstruct the execution tree, walk through all trace items on a given path, and perform analyses on them. For example, a tool that measures code coverage would look for trace items written by the *InstructionTracer* plugin to determine which instructions were executed. That tool would also rely on the module information provided in the trace by the *ModuleLoadUnloadTracer* plugin in order to associate each program counter with the corresponding module and display relevant debug information (e.g., function names).

5.1.3. Configuration Interface. Users can combine plugins using the S^{2}E configuration interface, which accepts scripts written in the Lua language [Lua 2010]. For each selector and analyzer used, there is a section in the script that lets the user control the plugin's behavior. For instance, the user can configure a data selector plugin to write symbolic values to some memory location after the system executes a particular function (e.g., users may want to fill a freshly `malloc`-ed buffer with symbolic values in order to track uses of uninitialized data).

5.2. 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 of the selected execution paths.

Plugin Interface. S^{2}E has a modular architecture, in which plugins communicate via events in a publish/subscribe fashion. S^{2}E events are generated either by the S^{2}E

Table II. Core Events Exported by the S^{2}E Platform

Event	Description
onInstrTranslation	The DBT is about to translate a machine instruction. A plugin can use this event to mark instructions of interests (e.g., calls or returns).
onInstrExecution	The VM is about to execute a marked instruction. This event invokes the callbacks registered via *onInstrTranslation*.
onExecutionFork	S^{2}E is about to split (fork) the current execution path in two. Mainly used by tracing plugins to embed the execution tree in trace files.
onInstanceFork	S^{2}E is about to spawn a new process instance of itself in order to explore the forked path on a different CPU core (more details in Section 4.4).
onException	The VM interrupt pin has been asserted. Provides a convenient means of intercepting interrupts and exceptions (e.g., fatal double faults).
onMemoryAccess	The VM is about to execute a memory access. This event can be used to simulate a cache hierarchy, record a memory trace for offline analysis, etc.
onPortAccess	The VM is about to execute a port I/O operation.
onCustomInstruction	The VM is about to execute a custom opcode. Listening plugins parse the custom instruction's operands to decode the action to perform.
onPageFault	A page fault occurred in the guest code.
onTlbMiss	A TLB missed occurred in the memory management unit. Using this event is faster than checking for misses every time *onMemoryAccess* fires.
onTimer	Timer event to let plugins implement periodic tasks, such as flushing trace files, periodically terminating uninteresting paths, etc.

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, and it is passed parameters specific to the event.

Table II 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 execution at the lowest possible level of abstraction: instruction translation, execution, memory accesses, and state forking.

Execution Path Abstraction. For each path being explored, there exists a distinct *ExecutionState* object instance; when an execution path splits (or forks), each child execution receives its own private copy of the parent *ExecutionState*. The *ExecutionState* object captures the current state of the entire virtual machine along a specific individual path. It is the first parameter of every event callback. *ExecutionState* enables plugins to toggle multipath execution on/off and gives them read/write access to the entire VM state, including the virtual CPU, VM physical memory, and virtual devices (see Table III for some of the *ExecutionState* 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.

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 *ExecutionState* object. *PluginState* must implement a *clone* method, so that it can be cloned by S^{2}E together with *ExecutionState* whenever execution forks. 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

Table III. A Subset of the *ExecutionState* Object's Interface

`void setForking(bool enable)`	Turn on/off multipath execution
`void read/writeMemory(uint64_t addr, Expr *buffer, size_t length)`	Read or write contents of memory (symbolic or concrete) at address *addr*
`Expr readReg(int reg)`	Read *val* (symbolic or concrete) from *reg*
`void writeReg(int reg, Expr val)`	Write *val* (symbolic or concrete) to *reg*
`TranslationBlock *getTb()`	Get currently executing code block from DBT
`PluginState *getPluginState(Plugin *plugin)`	Get per-path state object for the specified plugin instance

```
void s2e_make_symbolic(void* buf, int size, const char* name)
{
    __asm__ __volatile__(
        /* Binary encoding for S2SYM */
        ".byte 0x0f, 0x3f\n"
        ".byte 0x00, 0x03, 0x00, 0x00\n"
        ".byte 0x00, 0x00, 0x00, 0x00\n"
        : : "a" (buf), "b" (size), "c" (name) : "memory"
    );
}
```

Fig. 8. Embedding S^{2}E custom instructions in C programs.

inspects the *ExecutionState* object to see which instruction is about to be translated; if it is an instruction of interest (say, for example, a CALL), the plugin marks it. Whenever the VM executes a marked instruction, it raises the *onInstrExecution* event, which notifies the corresponding 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. When such an instruction gets subsequently executed, having the *onInstrTranslation* and *onInstrExecution* events separate leverages the fact that each instruction gets translated once, but may get executed millions of times (e.g., as in the body of a loop). For most analyses, *onInstrExecution* ends up being raised so rarely that using it introduces no runtime overhead (e.g., catching the kernel panic handler requires marking only the first instruction of that handler).

Custom Instructions. *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 multipath execution (S2ENA and S2DIS) and logging debug information (S2OUT). They give developers the finest grain control possible over multipath execution and analysis; they can be injected into the target code manually or with the help of binary instrumentation tools like PIN [Luk et al. 2005]. In practice, opcodes are the easiest way to mark data symbolic and get started with S^{2}E, without involving any plugins.

The code fragment in Figure 8 shows a C function that writes unconstrained symbolic values to the *buf* buffer. Symbolic values can be given a name, for instance, to improve readability when printing an expression that involves symbolic values. Note that S^{2}E opcodes do not require specialized compiler or assembler support, since most compilers and assemblers can already emit arbitrary byte sequences within the generated code.

The interface presented here was sufficient for all the multipath analyses we attempted with S^{2}E. Selectors can enable or disable multipath 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. We now provide two examples to illustrate the use of S^{2}E plugins.

5.3. Example 1: Using the S^{2}E API to Build the *Annotations* Plugin

An S^{2}E annotation is a piece of code written by an S^{2}E user in order to observe and manipulate the execution state. Therefore, annotations may be used for what might be called system-wide aspect-oriented programming, in which any instruction sequence can be preceded/followed/replaced by any other sequence of instructions.

Annotations can be used to implement different execution consistency models. Therefore, the *Annotations* plugin is a central piece in tools like DDT$^+$ and REV$^+$ (Section 6.1). For example, annotations can implement the RC-LC consistency by carefully replacing some function parameters and return values with symbolic data.

Appendix A details the steps that an S^{2}E plugin developer would take to develop the *Annotations* plugin as well as how an S^{2}E user would use such a plugin to perform the analysis of a Windows network device driver. In particular, we show how to implement *Annotations* in a platform-independent manner, making it suitable to analyze any kind of code on arbitrary guest operating systems. For this, we explain how an S^{2}E plugin developer could break down its functionality into smaller plugins that can be used independently or in combination by an S^{2}E user.

5.4. Example 2: Combining S^{2}E Plugins and In-VM Tools

In this second example, we show an alternate way of implementing annotations with in-VM tools, using SystemTAP [Prasad et al. 2005]. SystemTAP is a tracing framework for Linux that can intercept any function call or instruction in the kernel and invoke custom scripts. The scripts have full access to the system state. They can also leverage debug information to access variables by name.

S^{2}E users can leverage SystemTAP to obtain a flexible way of controlling path exploration. Users write SystemTAP scripts with embedded calls to S^{2}E custom instructions. This allows the injection of symbolic values in any place, terminate states based on complex conditions, interact with S^{2}E plugins, and more generally develop arbitrary selection schemes directly inside the guest OS.

Suppose we want to analyze the behavior of the Linux network stack when a network packet is received (e.g., check whether there is a packet that could crash the kernel). One approach is to replace the content of the incoming packets with symbolic values, in order to explore all the paths that depend on the packet's content.

Injecting symbolic packets in the Linux kernel can be done in a few lines of code with SystemTAP, as shown in Figure 9. We define a SystemTAP *probe* that intercepts calls to `netif_receive_skb`. Network drivers call this function when they are ready to pass incoming packets to the kernel. Besides the probe, the SystemTAP script also contains a call to the `s2e_make_symbolic` function. This function is the same as the one in Figure 8, except that it uses the SystemTAP syntax.

Note that it is also possible to use the *Annotations* plugin to perform this analysis, because the concept of annotation is similar to the SystemTAP probes. S^{2}E gives users the freedom to choose any method that is the most convenient for them to carry out a given analysis. For example, users may choose to adapt their existing SystemTAP scripts instead of rewriting them using the *Annotations* plugin's configuration syntax. Likewise, users may employ the *Annotations* plugin if the guest OS does not have an equivalent of SystemTAP or if the use of such a tool interferes with some aspect of the analysis (e.g., performance profiling).

```
function s2e_make_symbolic(buf: long, size:long, name:string) %{
    #SystemTap allows arbitrary C code (including inline assembly)
    __asm__ __volatile__(
      ".byte 0x0f, 0x3f\n"
      ".byte 0x00, 0x03, 0x00, 0x00\n"
      ".byte 0x00, 0x00, 0x00, 0x00\n"
          : : "a" ((uint32_t)THIS->buf),
              "b" ((uint32_t)THIS->size), "c"(THIS->name)
          : "memory"
    );
%}

#Insert symbolic values on each invocation of netif_receive_skb
probe kernel.function("netif_receive_skb") {
    s2e_make_symbolic($skb->data, $skb->len, "symbolic packet");
}
```

Fig. 9. Example of a SystemTAP probe that injects symbolic data into network packets.

5.5. Summary

In this section, we showed how S^2E users can combine various S^2E plugins to carry out the desired analysis tasks and how S^2E developers can write custom plugins using the S^2E developer API. An S^2E user can combine path selection plugins to limit the multipath exploration to the modules of interest with different analysis plugins, such as bug finders, performance profilers, and execution tracers. We explained how S^2E turns existing single-path analysis tools, such as Valgrind and Microsoft Driver Verifier, into multipath analyzers without any modification. Finally, we showed how developers can write modular plugins by taking the example of the *Annotations* plugin, which is a central piece of tools like REV$^+$ and DDT$^+$.

6. EVALUATION

S^2E's main goal is to enable rapid prototyping of useful, deep system analysis tools. In this vein, our evaluation of S^2E aims to answer three key questions: Is S^2E truly a general platform for building diverse analysis tools (Section 6.1)? Does S^2E perform these analyses with reasonable performance (Section 6.2)? What are the measured trade-offs involved in choosing different execution consistency models on both kernel-mode and user-mode binaries (Section 6.3)? All reported results were obtained on a 48-core, 2.0 GHz AMD Opteron machine with 512 GB of RAM, unless otherwise noted.

6.1. Three Use Cases

We used S^2E to build three vastly different tools: an automated tester for proprietary device drivers (Section 6.1.1), a reverse engineering tool for binary drivers (Section 6.1.2), and a multipath in-vivo performance profiler (Section 6.1.3).

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

6.1.1. Automated Testing of Proprietary Device Drivers. We used S^2E to build DDT$^+$, a tool for testing closed-source Windows device drivers. This is a reimplementation of DDT [Kuznetsov et al. 2010], an ad-hoc combination of changes to QEMU and KLEE, along with hand-written interface annotations: 35 KLOC added to QEMU and 7 KLOC modified, 3 KLOC added to KLEE and 2 KLOC modified. By contrast, DDT$^+$ has 720

Table IV.

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.

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
Multipath in-vivo performance profiling	n/a	20	n/a	767

LOC of C++ code, which combine several exploration and analysis plugins, and provides the necessary kernel API annotations to implement RC-LC.

DDT+ combines several plugins: the *CodeSelector* plugin restricts multipath exploration to the target driver, while the *MemoryChecker*, *DataRaceDetector*, and *WinBugCheck* analyzers look for bugs. To collect additional information about the quality of testing (e.g., coverage), we use the *InstructionTracer* analyzer plugin. Additional checkers can be easily added. DDT+ implements local consistency (RC-LC) via interface annotations that specify where to inject symbolic values while respecting local consistency; examples of annotations appear in Kuznetsov et al. [2010]. None of the bugs reported by DDT+ 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 ran DDT+ on two Windows network drivers, RTL8029 and AMD PCnet; DDT+ finds the same 7 bugs reported in Kuznetsov et al. [2010], 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+ achieves 42% basic-block coverage of the PCnet driver in 30 minutes, exploring more than 164,000 paths. For the RTL8029 driver, DDT covers 76% and 380,000 paths in less than 15 minutes.

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 input from hardware devices) and values that were injected according to the RC-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 RC-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 [Microsoft 2011b]. 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.

Table V.

Basic-block coverage obtained by RevNIC, REV+, and REV+ using 48-core S^{2}E. We also show the coverage increase over RevNIC.

	RevNIC	REV+ (S^{2}E)		REV+ (Multi-Core S^{2}E)	
	Coverage	Coverage	Increase	Coverage	Increase
PCnet	59%	66%	+7%	74%	+15%
91C111	84%	87%	+3%	89%	+5%
RTL8139	84%	86%	+2%	89%	+5%

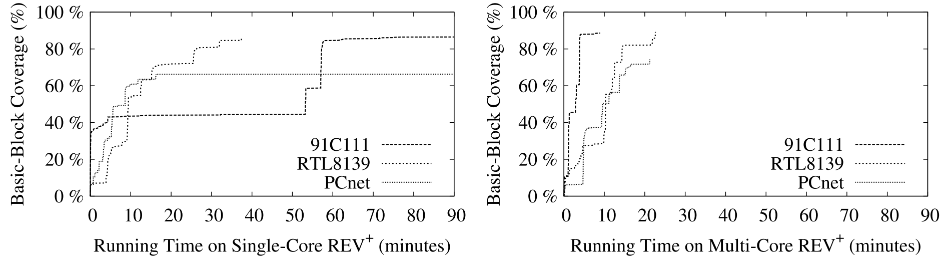

Fig. 10. Basic-block coverage over time for REV+ and REV+ on multicore S^{2}E. The graph for REV+ on multi-core S^{2}E shows data for 91C111, RTL8139, and PCnet on 48 cores.

6.1.2. Reverse Engineering of Closed-Source Drivers. REV+ is a tool for reverse engineering binary Windows device drivers; it is a reimplementation of RevNIC [Chipounov and Candea 2010]. REV+ takes a closed-source binary driver, traces its execution, and feeds the traces to an offline component that reverse engineers the driver's logic to produce new driver code that implements the 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 *InstructionTracer* plugin is configured to log to a file the driver's executed instructions and register accesses, while *MemoryTracer* is set up to record memory 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 CFG; details appear in Chipounov and Candea [2010]. By using RC-OC, REV+ sacrifices strict consistency in exchange for obtaining coverage fast.

We ran REV+ on the same drivers reported in Chipounov and Candea [2010], and REV+ reverse engineers them with better coverage than RevNIC in the same amount of time (see Table V). 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 [Chipounov and Candea 2010]. Figure 10 shows how coverage evolves over time during reverse engineering both for single-core S^{2}E [Chipounov et al. 2011] and multi-core S^{2}E. The single-core prototype completes the exploration of each driver in 30-90 minutes, whereas multi-core S^{2}E completes in a few minutes.

Table VI shows that more cores allow REV+ to explore many more paths and achieve higher basic block coverage. The exploration time increases with the number of cores because REV+ invokes each driver entry point sequentially, and when it does

Table VI.

Impact of additional cores on exploration time, basic-block coverage, and number of explored paths for REV+ on multicore S^2E using the overapproximate consistency model.

	PCnet			**RTL8139**			**91C111**		
# of cores used by S^2E	Time (min)	Cov (%)	# Paths	Time (min)	Cov (%)	# Paths	Time (min)	Cov (%)	# Paths
1	9	64	4,551	5	76	4,274	6	85	2,185
2	10	66	8,466	7	86	13,841	7	84	2,287
4	14	67	18,631	17	83	27,266	6	85	5,853
8	17	69	38,919	32	84	61,743	7	87	15,322
16	19	71	126,590	17	88	164,426	10	89	31,625
32	21	74	257,813	27	87	398,895	12	89	100,977
48	26	70	539,037	22	89	554,458	21	89	145,498

not discover any new basic block within some time interval, it kills all paths except one and invokes the next entry point (see Chipounov and Candea [2010] for more details). Since multiple cores can explore more paths in parallel and have thus a higher likelihood of discovering more basic blocks, REV+ resets the timeout more frequently, thus increasing the average exploration time. We used a timeout of 5 to 10 seconds for multicore S^2E and up to 1200 seconds for the single-core S^2E prototype combined with a random-path search heuristic [Cadar et al. 2008]. Given these settings, 32 cores are enough to get 74% to 89% basic-block coverage in less than 25 minutes.

Although more cores allow the exploration of more paths, this does not necessarily yield higher basic-block coverage. First, S^2E does not ensure that different cores do not perform redundant work. It may happen that all the cores explore one particular part of a driver (e.g., one function) instead of covering different parts. In future work, we plan to focus on achieving disjunction of explored paths, as in Cloud9 [Bucur et al. 2011], in order to minimize the amount of redundant work. Second, we believe that more cores influence the path selection heuristics that S^2E uses; we plan to investigate this phenomenon in future work.

6.1.3. Multipath In-Vivo Performance Profiling. To further illustrate S^2E's generality, we used it to develop $PROF_S$, a multipath in-vivo performance profiler and debugger. To our knowledge, such a tool did not exist previously, and we believe 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 [Valgrind 2011], which can simulate only L1 and L2 caches, and can measure only cache misses.

For $PROF_S$, we developed the *PerformanceProfiler* 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 64KB I1 and D1 caches with 64-byte cache lines and associativity 2, plus a 1MB 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 $\mathrm{PROF_S}$ experiment analyzes the distribution of instruction counts and cache misses for Apache's URL parser. In particular, we were interested to see whether there are opportunities for a denial-of-service attack on the Apache Web server via a carefully constructed URL. The analysis ran on $\mathrm{PROF_S}$ using 48 cores under local consistency for 1 hour and explored 51,530 different execution paths. The analysis spent 44% of the running time in the constraint solver.

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 $13,315 \pm 65$. These are examples of behavioral insights one can obtain with a multipath 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 are opportunities 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 attack of 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 (1.3 KLOC). The performance analysis ran under local consistency, focusing exploration on the IP packet options parser. $\mathrm{S^2E}$ explored 907 different paths in 1 hour using 48 cores. Around 30% of the time was spent in the constraint solver. Note that, in Chipounov et al. [2011], we focused the analysis on the entire packet-parsing code but with additional constraints on the packet content to prevent path explosion; we now analyze the part of the parser that focuses on packet options but include all possible paths through it. This makes the results easier to interpret because, now, all the obtained paths go through the packet options parser.

The analysis does not find a bound on execution time, and it points to a path that could enter 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 `ping` is patched, the performance envelope becomes 2,581 to 2,728 executed instructions. With the bug, the maximum during analysis had reached 1.1×10^6 instructions and kept growing.

$\mathrm{PROF_S}$ can find "best case performance" inputs without having to enumerate the input space. For this, we modify slightly the *PerformanceProfiler* 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 Section 5. 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 multipath analysis.

To conclude, we used $\mathrm{S^2E}$ to build a thorough multipath in-vivo performance profiler that improves upon classic profilers. Valgrind [Valgrind 2011] is thorough, but

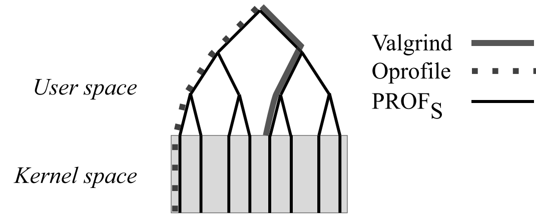

Fig. 11. The path coverage of classic profilers vs. the coverage level enabled by S^2E: $PROF_S$ is a thorough multipath in-vivo analyzer; Oprofile is in-vivo, but only single-path and sampling-based; Valgrind is thorough, but only single-path and not in-vivo.

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 [Levon and Elie 1998] perform in-vivo measurements, but not multipath, they are based on sampling, so they lack the accuracy of $PROF_S$; it is not feasible, for instance, to count the exact number of cache misses in an execution. Figure 11 summarizes the capabilities of Valgrind, Oprofile, and $PROF_S$. Such improvements over state-of-the-art tools come relatively easily when using S^2E to build new tools.

6.1.4. Other Uses of S^2E. We believe S^2E can be used for pretty much any type of system-wide analysis. We give here four additional examples.

First, S^2E could be used to profile energy use of embedded applications: given a power consumption model, S^2E could find energy-hogging paths and help the developer optimize them. Second, S^2E could serve as a hardware model validator: S^2E can symbolically execute a SystemC-based model [IEEE 2005] 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^2E could perform end-to-end certification of binaries, for instance, verify that memory safety holds along all critical paths. Finally, S^2E 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^2E could tell whether any of the data can leak outside the system. S^2E alleviates the need to trust a compiler, since it performs all analysis on the final binary.

6.2. Implementation-Specific Performance Overhead

S^2E introduces approximately $3\times$ runtime overhead over vanilla QEMU when running in concrete mode, and $78\times$ in symbolic mode. Concrete-mode overhead is mainly due to checks for accesses to symbolic memory, while symbolic-mode overhead is due to LLVM interpretation and constraint solving. S^2E incurs these overheads along each execution path both in single and multicore mode.

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 (Section 6.1.3), S^2E executed $30{,}000\times$ 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) run in concrete mode. Moreover, S^2E distinguishes inside the symbolic domain instructions that can execute concretely (e.g., that do not touch symbolic data) and runs them natively. `ping`'s four orders of magnitude difference between the number of concretely vs. symbolically running instructions is a *lower* bound on the amount of savings that selective symbolic execution brings over classic symbolic

execution: by executing concretely those paths that would otherwise run symbolically, S^2E *also* saves the overhead of further forking paths that are ultimately not of interest (e.g., on branches in the concrete domain).

Another source of overhead is symbolic pointers. We compared the performance of symbolically executing the `unlink` UNIX utility's x86 binary in S^2E on a single core 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^2E must pass entire memory pages. In 1 hour, using one core, with a 256-byte page size, S^2E explored 7,082 paths, compared to 7,886 paths in KLEE. Average constraint solving time was 0.06 sec for both. With 4 KB pages, though, S^2E explored only 2,000 states and averaged 0.15 sec per constraint.

We plan to reduce the overhead in concrete and symbolic modes in three ways: First, by instrumenting the LLVM code generated by S^2E with calls to the symbolic execution engine, before JITing it into native machine code, we can avoid the overhead of interpreting each instruction in KLEE. This is similar in spirit to the difference between QEMU and the Bochs emulator [Bochs 2011]: 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 implement LLVM optimization passes to bring the generated LLVM code close to generated-from-source quality. Currently, the translator generates LLVM code that can be up to two orders of magnitude longer than what a native compiler would produce. Finally, we expect that by introducing hardware virtualization, we would obtain a ~30× speedup in concrete mode, which is close to the speed of native execution.

6.3. Trade-Offs in Using Execution Consistency Models

Having seen the ability of S^2E to serve as a platform for building powerful analysis tools, we now experimentally evaluate the trade-offs 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 trade-offs 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 [Lua 2010]). 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 and parser (2 KLOC) and the environment, while the symbolic domain is the remaining code of 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 [Godefroid et al. 2008]. The ease of separating the Lua interpreter from its parser and lexer in S^2E without touching the Lua source code 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` method. We configure a selector plugin to exercise the entry points one by one. If S^2E has not discovered any new basic block for some time, this plugin kills all paths but one. The plugin chooses the remaining path such that execution can proceed to the driver's next entry point. We use the same settings as in Section 6.1.2 and vary the consistency model.

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

Table VII.

Time to complete exploration of two device drivers and the Lua interpreter under different consistency models.

Consistency	91C111 Driver	PCnet Driver	Lua
RC-OC	6 min	9 min	32 min
RC-LC	10 min	13 min	31 min
SC-SE	5 min	9 min	33 min
SC-UE	$<$1 min	$<$1 min	$<$1 min

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 program input, under SC-SE consistency. In SC-UE mode, the input is symbolic like in SC-SE, but all symbolic data is concretized when accessed by code outside of the symbolic domain (i.e., outside of the Lua execution engine). 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 Lua opcodes completely unconstrained.

In this section, we run S^2E in single-core mode and average results over 10 runs for each consistency model. Running S^2E in multicore mode would introduce additional randomness to the results, making it difficult to compare different data points; since the state selection is local to each S^2E process, some of the processes may end up executing states that would never be selected if the state selection was global. Whether this happens or not depends on the initial distribution of the states between S^2E processes, which is difficult to predict.

Generally speaking, weaker (more relaxed) consistency models help achieve higher basic-block coverage in a given amount of time; Figure 12(a) shows results for the running times from Table VII. For PCnet, coverage varies between 14% and 65%, while 91C111 ranges from 10% to 84%. 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. For the 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 and prevents Lua from executing a meaningful command, thus yielding poor coverage and short running time.

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 and their operands (such opcodes could never reach the parser during normal execution, hence the parser does not check for them but instead continues erroneous execution for some time, leading to multiple forks before it finally crashes).

Path selection together with adequate consistency models reduce memory usage (Figure 12(b)). Stricter models generate fewer execution paths to explore, which in principle should reduce memory consumption. However, in practice memory usage is more strongly correlated with running time. For example, in RC-OC and SC, it takes roughly 6 minutes to complete the execution of the 91C111 driver, taking 1.5 GB of memory for 1,950 and 620 paths respectively. RC-LC takes longer, using 2 GB of memory for 955 paths. The reason is that, the longer the paths run, the bigger the corresponding program states grow, due to copy-on-write effects: various OS components have more time to write into more memory pages, yielding higher per-state memory consumption.

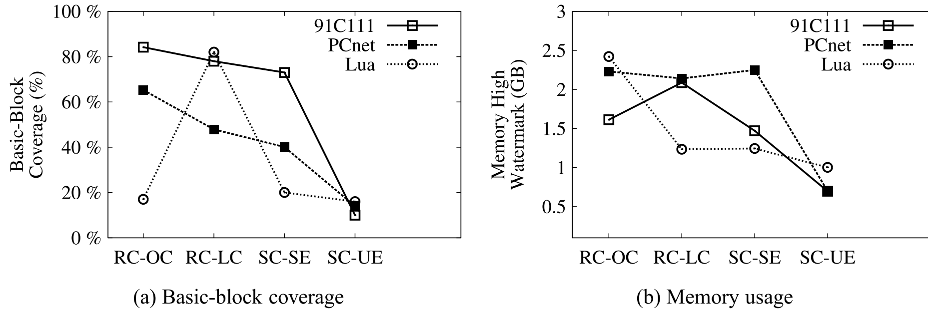

Fig. 12. Effects of memory consistency models on coverage and memory usage.

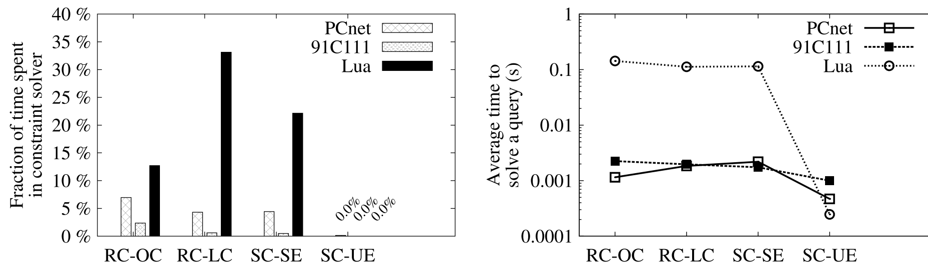

Fig. 13. Impact of consistency models on constraint solving.

Finally, consistency models affect constraint solving time (Figure 13). The relationship between consistency model and constraint solving time often depends on the structure of the system being analyzed; at a first level of approximation, the deeper a path, the more complex the corresponding path constraints. The fraction of execution time spent in the constraint solver decreases with stricter execution consistency models, because stricter models restrict the amount of symbolic data, generating fewer queries. Note that analyzing Lua under RC-OC exceptionally yielded a low fraction of time spent in the constraint solver for the same reason it got poor coverage: execution could not reach the more complex parts of the interpreter.

We observe that, except for SC-UE, the average time spent to solve a query remains roughly constant across consistency models. This is in contrast to our earlier results [Chipounov et al. 2011], where S^{2}E used an older version of STP (revision #943 vs. #1432 in this article). That older version took more time to solve queries generated by weaker consistency models. We plan to investigate this behavior in future work. SC-UE concretizes symbolic values early, which strongly reduces the number and complexity of solver queries and makes them quicker to solve on average.

We attempted to run Lua in KLEE in order to compare the results for different execution consistency models with those obtained in S^{2}E. 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` instructions, which turn into `libc` calls that manipulate the program counter and other registers in a way that confuses KLEE. Unlike S^{2}E, other analysis engines do not have a unified representation of the hardware, so all these details must be explicitly coded for (in KLEE's case, detect that `setjmp` / `longjmp` is used and ensure the execution state is appropriately adjusted). In S^{2}E, this comes "for free" because the

CPU registers, memory, and I/O devices are shared between the concrete and symbolic domain.

Our evaluation shows that S^2E 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 analysis. By employing selective symbolic execution and relaxed execution consistency models, S^2E 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^2E explores hundreds of thousands of paths through the binaries.

7. RELATED WORK

Several of the ideas behind S^2E appeared in various forms in earlier work, and we survey them in this section. We are not aware of any platform that can offer the level of generality in terms of analyses and execution consistency models that S^2E offers.

BitBlaze [Song et al. 2008] is the closest dynamic analysis framework to S^2E. 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^2E 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^2E 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 [Cadar et al. 2008]. 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 [Ball et al. 2006]. Of course, the advantage of using models is generally faster analyses.

Other bodies of work can include the environment directly in symbolic analysis by executing the environment concretely, with various levels of consistency that were appropriate for the specific analysis in question, most commonly bug finding. For instance, CUTE [Sen et al. 2005] can run concrete code consistently without modeling, but it is limited to strict consistency and code-based selection. SJPF [Păsăreanu et al. 2008] can switch from concrete to symbolic execution, but does not track constraints when switching back, so it cannot preserve consistency in the general case. Non-VM-based approaches, in general, 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 [Godefroid et al. 2005; Sen 2007] 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. CUTE, KLEE, and other similar tools cannot track the branch conditions following calls into concrete code (unlike S^2E), and thus cannot determine how to redo calls in order to enable overconstrained but feasible paths.

In-situ model checkers [Godefroid 1997; Musuvathi et al. 2008; Yang et al. 2006, 2009] can directly check programs written in a common programming language usually with some simplifications, like data-range reduction, without requiring

the creation of a model. Since S^2E directly executes the target binary, one could say it is an in-situ tool. However, S^2E 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^2E: analyzing the target code in-situ, while facilitating its consistent interaction with that code's unmodified, real environment. Note that other researchers have used the term "in vivo" in similar contexts as well, but with a different meaning from S^2E's; e.g., Murphy et al. [2009] propose a technique for testing where "in vivo" stands for executing tests in production environments.

Another approach to tackling path explosion is compositional symbolic execution [Godefroid 2007]. 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^2E, to further improve scalability.

Several static analysis frameworks have been used to build analysis tools. Saturn [Dillig et al. 2008] and `bddbddb` [Lam et al. 2005] prove the presence or absence of bugs using a path-sensitive analysis engine to decrease the number of false positives. Saturn uses analysis-specific function summaries; for many analyses, such as memory safety, these summaries can be prohibitively complex in large programs. `bddbddb` stores programs in a database as relations that can be searched for buggy patterns using Datalog. Besides detecting bugs, `bddbddb` was used for optimizing locking policies 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. From the practitioner's point of view, `bddbddb` requires learning a new language, which may be harder than using S^2E plugins.

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 [Schwarz et al. 2002]: disambiguating code from data, determining the targets of indirect jumps, unpacking code, etc.

S^2E can add multipath analysis abilities to any single-path dynamic tools, while not limiting the types of analysis. PTLsim [Yourst 2007] is a VM-based cycle-accurate x86 simulator that selectively limits profiling to user-specified code ranges to improve scalability. Valgrind [Valgrind 2011] is a framework best known for cache profiling tools, memory leak detectors, and call graph generators. PinOS [Bungale and Luk 2007] can instrument operating systems and unify user/kernel-mode tracers. PinOS relies on Xen and a paravirtualized guest OS, unlike S^2E. PTLsim, PinOS, and Valgrind implement cache simulators that model multi-level data and code cache hierarchies. S^2E allowed us to implement an equivalent multipath simulator with little effort.

S^2E complements classic single-path, non VM-based profiling and tracing tools. For instance, DTrace [DTrace 2011] 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 [Anderson et al. 1997], sampling-based profiling [Burrows et al. 2000], and data type profiling [Pesterev et al. 2010], 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^2E by allowing the analyses to be exposed to multipath executions.

S^2E uses mixed-mode execution as an optimization to increase efficiency. This idea first appeared in DART [Godefroid et al. 2005], CUTE [Sen et al. 2005], and EXE [Cadar et al. 2006], and later in Bitscope [Brumley et al. 2007]. 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 article described S^{2}E, a new platform for in-vivo multipath analysis of systems, which scales even to large, proprietary, real-world software stacks, like Microsoft Windows. To the best of our knowledge, 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/` and presently has a strong and diverse user community.

APPENDIX

A. USING THE S^{2}E API TO BUILD THE *ANNOTATIONS* PLUGIN

In Section 5.3, we gave an overview of how to build the *Annotations* plugin. An S^{2}E annotation is a piece of code written by an S^{2}E user in order to observe and manipulate execution states. The *Annotations* plugin can be used to implement different execution consistency models and is a central piece in tools like DDT$^+$ and REV$^+$ (Section 6.1).

Given its wide use, the *Annotations* plugin must be generic and work on any piece of code, no matter what guest OS is running. *Annotations* implements only the mechanisms that let users specify the desired annotations and relies on other plugins for unrelated tasks, such as OS monitoring.

We show here in detail how to build a plugin that monitors the guest OS and notifies other plugins when programs, drivers, libraries, or any kind of modules are loaded (Section A.1), how to use the information about loaded modules to detect the execution of a specific module (Section A.2), how to track function calls in those modules (Section A.3), and finally, how to let users annotate the desired code and make sure the annotations are executed at the right moment (Section A.4). Figure 14 summarizes the relationship between these plugins and Figure 17 shows the corresponding S^{2}E configuration file that we use throughout the remainder of this section as a running example. This example shows how users can configure the *Annotations* plugin in order to insert symbolic data in network packets during the analysis of the `rtl8029.sys` network device driver, that is part of Windows XP.

A.1. Monitoring Module Loads

S^{2}E requires a specific monitoring plugin for each OS in order to track OS-level events, such as module loads and unloads. Tracking these events in a system may be difficult and platform-specific. For example, getting the process identifier of the currently executing process requires parsing OS-specific data structures in the guest's kernel heap. Moreover, the exact layout of these structures varies for different versions of

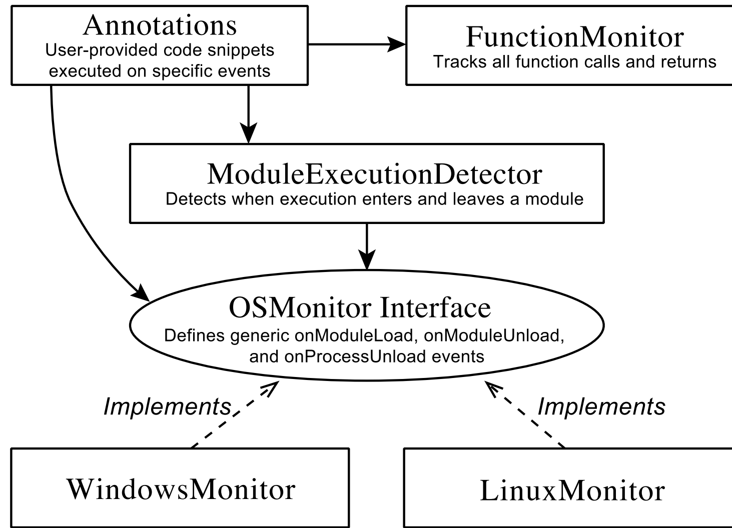

Fig. 14. Dependencies of the *Annotations* plugin. *Annotations* requires *ModuleExecutionDetector*, *FunctionMonitor*, and a plugin that implements the *OSMonitor* interface (such as *WindowsMonitor* or *LinuxMonitor*). This is a more detailed view of Figure 7.

the same operating system. Such implementation details must be hidden behind a generic interface.

S^{2}E plugins that monitor OS-level events must implement a generic interface in order to be used interchangeably by client plugins. For example, a plugin such as *Annotations* (Section A.4) does not care about whether the guest is running on Microsoft Windows or Linux. Such plugin only needs to know when the OS loads a specified module in order to activate annotations. For this, *Annotations* relies on the generic interface exposed by the underlying OS monitoring plugin (e.g., *WindowsMonitor* or *LinuxMonitor*).

S^{2}E provides the *OSMonitor* interface, which OS monitoring plugins implement. *OSMonitor* defines the *onModuleLoad*, *onModuleUnload*, and *onProcessUnload* events. An OS monitoring plugin triggers *onModuleLoad* (respectively *onModuleUnload*) when a module is loaded (respectively unloaded) and passes the name, size, load address, and address space identifier to the callback functions. The plugin triggers *onProcessUnload* when the OS frees the address space of a process. There is no corresponding *onProcessLoad* event, because the first *onModuleLoad* implicitly defines the new address space.

Consider *WindowsMonitor*, a plugin that implements the *OSMonitor* interface for Microsoft Windows. Detecting driver loads on Windows XP SP3 involves catching the execution of the instruction located at address `0x805A399A` in kernel space. When execution reaches this address,*WindowsMonitor* parses the driver descriptor located on the stack, extracts the name, load address, and size of the driver, then triggers the *onModuleLoad* event. Subscribers are notified of the driver load and can perform actions accordingly, e.g., detect when execution enters a particular module (see Section A.2).

Say the S^{2}E user wants to analyze the `rtl8029.sys` driver running on Windows XP SP3. Since device drivers run in kernel mode, *WindowsMonitor* must be configured to instrument kernel module loads and unloads. This requires five lines of configuration, as shown in Figure 15.

```
pluginsConfig.WindowsMonitor = {
    version="XPSP3",
    userMode=false,
    kernelMode=true
}
```

Fig. 15. Configuring *WindowsMonitor* to track driver load/unload events on Microsoft Windows.

```
pluginsConfig.ModuleExecutionDetector = {
    rtl8029_sys_1 = {
        moduleName = "rtl8029.sys",
        kernelMode = true
    }
}
```

Fig. 16. Configuring *ModuleExecutionDetector* to track the execution of the `rtl8029.sys` driver.

A.2. Tracking Module Execution with *ModuleExecutionDetector*

The *ModuleExecutionDetector* plugin publishes two main events: *onModuleInstrTranslation* and *onModuleTransition*. *onModuleInstrTranslation* forwards all the *onInstrTranslation* core events that are triggered inside the modules of interest. *onModuleTransition* notifies its clients whenever execution enters or leaves modules of interest.

Subscribers can use these two events in several ways. For example, *CodeSelector* subscribes to *onModuleTransition* to be notified of when execution enters or leaves a module of interest in order to toggle symbolic execution. *onModuleInstrTranslation* is used by *InstructionCounter*, which relies on this event to register a callback that S^2E will call for each instruction executed by the module. The callback increments an instruction counter and periodically writes its value to a log file.

ModuleExecutionDetector relies on a plugin that implements the *OSMonitor* interface. When S^2E starts, *ModuleExecutionDetector* automatically looks for a plugin that implements the *OSMonitor* interface. It subscribes to the *onModuleLoad*, *onModuleUnload*, and *onProcessUnload* to maintain the current memory map of the system. This map allows to efficiently find the module that owns a particular address and trigger the *onModuleInstrTranslation* as well as *onModuleTransition* when appropriate.

To enable *ModuleExecutionDetector*, the S^2E user adds the appropriate section in the S^2E configuration script, as shown in Figure 16. This section can go right below the one for *WindowsMonitor* that we have seen previously. *ModuleExecutionDetector*'s configuration section accepts one subsection per module to be tracked. Each module to track is identified by its name and whether it is a kernel module or not. Subsections can be named (e.g., `rtl8029_sys_1`) to allow other plugins to refer to them, as we will illustrate later.

A.3. Monitoring Function Calls with *FunctionMonitor*

The *FunctionMonitor* plugin notifies its subscribers of function calls and returns. When subscribing, a client plugin passes to *FunctionMonitor* the address of the function to monitor, the identifier of the address space to which the function belongs, and an event callback. *FunctionMonitor* invokes the registered callback whenever a function call or return occurs. The address space identifier allows distinguishing functions at the same virtual address but in different processes.

FunctionMonitor tracks pairs of call and return machine instructions. When a call occurs, besides invoking the registered callback, *FunctionMonitor* also stores in a map the association between the current stack pointer, the address space, and the event callback that corresponds to the called function. When a return instruction is about

```
pluginsConfig.WindowsMonitor = {
    version="XPSP3",
    userMode=false,
    kernelMode=true
}

pluginsConfig.ModuleExecutionDetector = {
    rtl8029_sys_1 = {
        moduleName = "rtl8029.sys",
        kernelMode = true
    }
}

pluginsConfig.FunctionMonitor = { }

pluginsConfig.Annotation = {
  my_annotation = {
    module="rtl8029_sys_1",
    address=0x1233a,
    paramcount=4,
    callAnnotation="rtl8029_copyup_packet"
  }
}

function rtl8029_copyup_packet(state, pluginState)
   buffer = state:readParameter(1);
   length = state:readParameter(3);

   for i = 0, length - 1, 1 do
        state:writeMemorySymb("copyup_buffer", buffer + i, 1);
   end

   state:writeRegister("eax", 1);
   pluginState:setSkip(true);
end
```

Fig. 17. Combining S^{2}E plugins to inject symbolic network packets in the `rtl8029.sys` driver. This S^{2}E configuration file is written in the Lua scripting language.

to be executed, *FunctionMonitor* looks up the current stack pointer and address space identifier in the map and invokes the associated callback. Such tracking is required because return instructions do not carry any information about the function to which they belong.

FunctionMonitor subscribes to the *onInstrTranslation* core event in order to mark and intercept all call and return machine instructions. Whenever these marked instructions are executed, S^{2}E triggers the *onInstrExecution* event which invokes the callbacks previously registered by *FunctionMonitor* when processing the *onInstrTranslation* events. These callbacks check whether there are clients of *FunctionMonitor* that registered for the specific function call or return that is being executed and, if yes, invoke the corresponding client's event callback.

FunctionMonitor assumes that the processor's instruction set has explicit call and return instructions, which is the case, for instance, of x86 or MIPS. MIPS uses the `jal` (jump and link) instruction for function calls. This instruction jumps to the specified address while saving in the `$ra` register the program counter of the instruction that follows the jump. Since `$ra` holds the current return address by convention, it can be used to detect jumps that use this register to return to the caller.

FunctionMonitor does not have any user-configurable option. Thus, it is enough to write an empty configuration section as shown in Figure 17.

A.4. Annotating Code with the *Annotations* plugin

The *Annotations* plugin combines *FunctionMonitor* and *ModuleExecutionDetector* to let users annotate not only function calls but also arbitrary machine instructions. The user writes the annotation directly inside the S^{2}E configuration file, using the Lua language.

The *Annotations* plugin has four configurable parameters: the module name (`module`), the address of the function to intercept (`address`), the number of its parameters (`paramcount`), as well as the name of the Lua annotation to invoke (`callAnnotation`). It is also possible to use `instructionAnnotation` to annotate arbitrary instructions.

In our example, we configure the *Annotations* plugin to annotate the function that copies a data packet from the network card to a buffer allocated by the driver. This

```
pluginsConfig.Annotation = {
  my_annotation = {
    module="rtl8029_sys_1",
    address=0x1233a,
    paramcount=4,
    callAnnotation="rtl8029_copyup_packet"
  }
}
```

Fig. 18. Configuring the *Annotations* plugin to inject symbolic network packets in the `rtl8029.sys` driver.

```
function rtl8029_copyup_packet(state, pluginState)
   buffer = state:readParameter(1);
   length = state:readParameter(3);

   for i = 0, length - 1, 1 do
        state:writeMemorySymb("copyup_buffer", buffer + i, 1);
   end

   state:writeRegister("eax", 1);
   pluginState:setSkip(true);
end
```

Fig. 19. Example of an annotation written in the Lua language.

function has four parameters and is located at address `0x1233a` relative to the start of the `rtl8029.sys` driver (Figure 18).

The annotation is contained in the `rtl8029_copyup_packet` Lua function (Figure 19). All annotations have two parameters: the current execution state and the current plugin state. The execution state object can be manipulated using the *ExecutionState* object's methods. Similarly, the plugin state parameter exposes the API of the *Annotations* plugin, which allows annotations to change the plugin's configuration at runtime.

ACKNOWLEDGMENTS

We thank Andrea Arpaci-Dusseau, Herbert Bos, Johannes Kinder, Miguel Castro, Byung-Gon Chun, Jim Larus, Petros Maniatis, Raimondas Sasnauskas, Willy Zwaenepoel, the S^{2}E user community, and the anonymous reviewers for their help in improving our article.

REFERENCES

ANDERSON, J., BERC, L., DEAN, J., GHEMAWAT, S., HENZINGER, M., LEUNG, S.-T., SITES, D., VANDEVOORDE, M., WALDSPURGER, C. A., AND WEIHL, W. E. 1997. Continuous profiling: Where have all the cycles gone? In *Proceedings of the Symposium on Operating Systems Principles*.

BALL, T., BOUNIMOVA, E., COOK, B., LEVIN, V., LICHTENBERG, J., MCGARVEY, C., ONDRUSEK, B., RAJAMANI, S. K., AND USTUNER, A. 2006. Thorough static analysis of device drivers. In *Proceedings of the ACM EuroSys European Conference on Computer Systems*.

BALL, T., BOUNIMOVA, E., LEVIN, V., KUMAR, R., AND LICHTENBERG, J. 2010. The static driver verifier research platform. In *Proceedings of the International Conference on Computer Aided Verification*.

BELLARD, F. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference*.

BESSEY, A., BLOCK, K., CHELF, B., CHOU, A., FULTON, B., HALLEM, S., HENRI-GROS, C., KAMSKY, A., MCPEAK, S., AND ENGLER, D. 2010. A few billion lines of code later: Using static analysis to find bugs in the real world. *Comm. ACM 53*, 2.

BOCHS. 2011. Bochs IA-32 emulator. `http://bochs.sourceforge.net/`.

BOONSTOPPEL, P., CADAR, C., AND ENGLER, D. R. 2008. RWset: Attacking path explosion in constraint-based test generation. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.

BRUMLEY, D., HARTWIG, C., KANG, M. G., NEWSOME, Z. L. J., POOSANKAM, P., SONG, D., AND YIN, H. 2007. BitScope: Automatically dissecting malicious binaries. Tech. rep. CMU-CS-07-133, Carnegie Mellon University.

BUCUR, S., URECHE, V., ZAMFIR, C., AND CANDEA, G. 2011. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the ACM EuroSys European Conference on Computer Systems*.

BUNGALE, P. P. AND LUK, C.-K. 2007. PinOS: a programmable framework for whole-system dynamic instrumentation. In *Proceedings of the International Conference on Virtual Execution Environments*.

BURROWS, M., ERLINGSON, U., LEUNG, S.-T., VANDEVOORDE, M. T., WALDSPURGER, C. A., WALKER, K., AND WEIHL, W. E. 2000. Efficient and flexible value sampling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. 2006. EXE: Automatically generating inputs of death. In *Proceedings of the Conference on Computer and Communication Security*.

CADAR, C., DUNBAR, D., AND ENGLER, D. R. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Symposium on Operating Systems Design and Implemenatation.*

CHIPOUNOV, V. AND CANDEA, G. 2010. Reverse engineering of binary device drivers with RevNIC. In *Proceedings of the ACM EuroSys European Conference on Computer Systems*.

CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. 2011. S2E: A platform for in-vivo multipath analysis of software systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

DILLIG, I., DILLIG, T., AND AIKEN, A. 2008. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the Conference on Programming Language Design and Implementation*.

DTRACE. 2011. Dtrace. http://www.sun.com/bigadmin/content/dtrace/index.jsp.

GODEFROID, P. 1997. Model checking for programming languages using VeriSoft. In *Proceedings of the Symposium on Principles of Programming Languages*.

GODEFROID, P. 2007. Compositional dynamic test generation. In *Proceedings of the Symposium on Principles of Programming Languages*.

GODEFROID, P., KLARLUND, N., AND SEN, K. 2005. DART: Directed automated random testing. In *Proceedings of the Conference on Programming Language Design and Implementation*.

GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. 2008. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium.*

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

INTEL. 2011. *Intel 64 and IA-32 Architectures Software Developers Manual*. Vol. 2.

JAVA PATHFINDER. 2007. Java PathFinder. http://javapathfinder.sourceforge.net.

KING, J. C. 1975. A new approach to program testing. In *Proceedings of the International Conference on Reliable Software*.

KUZNETSOV, V., CHIPOUNOV, V., AND CANDEA, G. 2010. Testing closed-source binary device drivers with DDT. In *Proceedings of the USENIX Annual Technical Conference*.

LAM, M. S., WHALEY, J., LIVSHITS, V. B., MARTIN, M. C., AVOTS, D., CARBIN, M., AND UNKEL, C. 2005. Context-sensitive program analysis as database queries. In *Proceedings of the Symposium on Principles of Database Systems*.

LATTNER, C. AND ADVE, V. 2004. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*.

LEVON, J. AND ELIE, P. 1998. Oprofile. http://oprofile.sourceforge.net.

LUA 2010. Lua: A lightweight embeddable scripting language. http://www.lua.org/.

LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. 2005. PIN: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the Conference on Programming Language Design and Implemenation.*

MICROSOFT. 2011a. WHDC: Develop hardware for windows. http://www.microsoft.com/whdc.

MICROSOFT. 2011b. Windbg. http://msdn.microsoft.com/en-us/windows/hardware/gg463009.

MILLER, B., FREDRIKSEN, L., AND SO, B. 1990. An empirical study of the reliability of UNIX utilities. *Comm. ACM 33*, 12.

MURPHY, C., KAISER, G., VO, I., AND CHU, M. 2009. Quality assurance of software applications using the in vivo testing approach. In *Proceedings of the International Conference on Software Testing Verification and Validation*.

MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. 2008. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the Symposium on Operating System Design and Implementation*.

PESTEREV, A., ZELDOVICH, N., AND MORRIS, R. T. 2010. Locating cache performance bottlenecks using data profiling. In *Proceedings of the ACM EuroSys European Conference on Computer Systems*.

PRASAD, V., COHEN, W., EIGLER, F. C., HUNT, M., KENISTON, J., AND CHEN, B. 2005. Locating system problems using dynamic instrumentation. In *Proceedings of the Linux Symposium*.

PĂSĂREANU, C., MEHLITZ, P., BUSHNELL, D., GUNDY-BURLET, K., LOWRY, M., PERSON, S., AND PAPE, M. 2008. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Proceedings of the International Symposium on Software Testing and Analysis*.

PULKKINEN, T., NELSON, K., PULKKINEN, E., CUMMING, M., AND SCHULZE, M. 2011. libsigc++ — The Typesafe Callback Framework for C++. http://libsigc.sourceforge.net/.

SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst. 15*, 4.

SCHWARZ, B., DEBRAY, S., AND ANDREWS, G. 2002. Disassembly of executable code revisited. In *Proceedings of the Working Conference on Reverse Engineering*.

SEN, K. 2007. Concolic testing. In *Proceedings of the International Conference on Automated Software Engineering*.

SEN, K., MARINOV, D., AND AGHA, G. 2005. CUTE: A concolic unit testing engine for C. In *Proceedings of the Symposium on the Foundations of Software Engineering*.

SONG, D., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. 2008. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the International Conference on Information Systems Security*.

VALGRIND. 2011. Valgrind. http://valgrind.org/.

WHEELER, D. 2010. SLOCCount. http://www.dwheeler.com/sloccount/.

YANG, J., SAR, C., AND ENGLER, D. 2006. EXPLODE: A lightweight, general system for finding serious storage system errors. In *Proceedings of the Symposium on Operating Systems Design and Implementation*.

YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. 2009. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the Symposium on Networked Systems Design and Implementation*.

YOURST, M. T. 2007. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*.

Received August 2011; accepted October 2011