Selective Symbolic Execution

Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, George Candea
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

Symbolic execution is a powerful technique for analyzing program behavior, finding bugs, and generating tests, but suffers from severely limited scalability: the largest programs that can be symbolically executed today are on the order of thousands of lines of code. To ensure feasibility of symbolic execution, even small programs must curtail their interactions with libraries, the operating system, and hardware devices. This paper introduces selective symbolic execution*, a technique for creating the illusion of full-system symbolic execution, while symbolically running only the code that is of interest to the developer. We describe a prototype that can symbolically execute arbitrary portions of a full system, including applications, libraries, operating system, and device drivers. It seamlessly transitions back and forth between symbolic and concrete execution, while transparently converting system state from symbolic to concrete and back. Our technique makes symbolic execution practical for large software that runs in real environments, without requiring explicit modeling of these environments.*

1 Introduction

Symbolic execution has recently gained popularity in automated software testing [4, 6, 14, 9, 3, 12, 11] as well as for studying malware [10, 2]. Behaviors (such as bugs) discovered with symbolic execution can be easily reproduced using information collected during the corresponding symbolic execution, making this approach a powerful and cost-effective tool for developers and testers.

When a program is symbolically executed, it is provided with a *symbolic* value for each input (λ, β, ...), instead of the *concrete* inputs (9, "foo", ...) it would normally get. Every assignment in the program along a given execution path updates the program variables with a symbolic expression (e.g., $\texttt{x} = \lambda - 2$), instead of a concretely computed value (x=9-2=7). A conditional statement (e.g., "`if x>0 then... else...`") splits the execution into two new paths—one for the then-branch and one for the else-branch—with a common prefix. Along the then-path values are constrained by the `if` condition ($\lambda - 2 > 0$) and along the else-path by the `else` condition ($\lambda - 2 \leq 0$). The conjunction of the constraints collected along a path can be fed to a constraint solver to find a concrete input (e.g., $\lambda = 9$) that would take the program along that path. For the paths that lead, for instance, to `assert` statements, the solutions for λ, β, ... constitute test cases that reproduce the corresponding crashes.

A symbolic execution engine will typically represent the program's memory in an engine-specific data structure and, when the execution reaches a branch whose condition involves symbolic values, the engine *forks* the program state, such that each path has its private version of the program state. In this way, the engine can explore both branches independently and in parallel. The hierarchy of program states forms an *execution tree*, as in Figure 1.

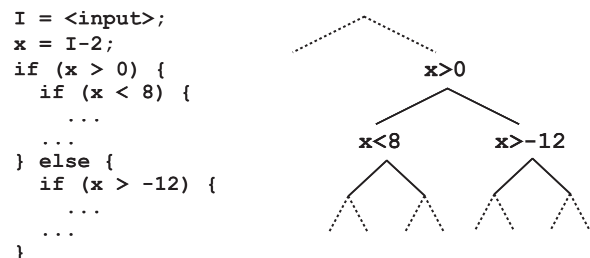

Figure 1. A program and its symbolic execution tree.

The size of the execution tree grows exponentially in the number of conditionals, an effect called *path explosion*. The total amount of state and number of constraints maintained during symbolic execution grow correspondingly. While researchers have proposed a variety of optimizations, current symbolic execution engines [3, 6, 10, 12] still scale only to small programs with thousands of lines of code. It is therefore not yet practical to execute symbolically the majority of software we use today (Firefox, OpenOffice, MySQL, Eclipse, etc.), each of which has millions of lines of code.

Another challenge in symbolic execution is the target program's *interaction with the environment*. For example, when a program like Firefox reads from a network socket, it calls a function in libc, which in turn executes a system call, which then invokes the NIC device driver, which then reads data from the NIC device, and returns it up the stack (see Figure 2). To symbolically execute Firefox would require symbolically executing all the invoked libraries, OS, and drivers, but doing so would compound path explosion and require inordinate amounts of memory and CPU time. Instead, current tools build custom libraries and abstract models of the environment [2, 3], which isolate the program and keep symbolic execution within the boundaries of the tar-

get program. Building complete models, however, is difficult and labor-intensive, since libraries and OSes have many thousands of API functions with complex semantics. There are few such models, so symbolic execution is limited to programs that interact little with the outside world.

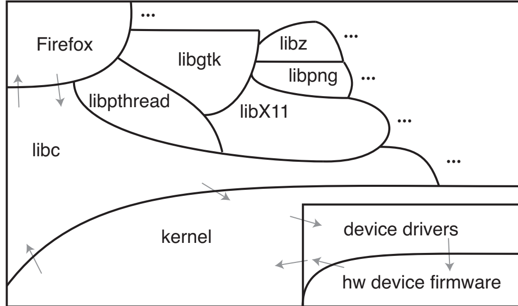

Figure 2. The execution space of a full system.

Our observation is that developers generally do not test/study an entire system at a time, but rather focus on only a small portion of it—a kernel module, a pair of user programs, recently-added functionality, code that touches a recently-modified data structure, etc. Thus, the results of symbolically executing the entire system are not necessary.

This paper introduces *selective symbolic execution* (S^2E), a technique for providing the illusion of full-system symbolic execution. Our virtual execution platform allows users to specify a scope of interest within a system's execution space, and then focuses CPU and memory resources on symbolically executing that scope only. In-scope executions are explored symbolically, as if the full system was executing symbolically, while out-of-scope ones are run concretely. Execution flows seamlessly back and forth between the symbolic domain (i.e., in-scope execution paths) and the concrete domain (i.e., out-of-scope paths), and system state is suitably converted on every boundary crossing.

The main challenge in S^2E is how to provide this back and forth flow transparently, while ensuring a consistent and efficient execution.

S^2E helps scale symbolic execution by a priori pruning parts of the execution tree that the developer would not even look at once execution completed. Approaches like hybrid concolic testing [9] and symbolic JPF [12] are specific instances of such pruning, and selective symbolic execution generalizes and advances beyond prior work. S^2E enables the productive use of symbolic execution in large software systems with complex environment interactions, without requiring explicit environment modeling.

2 Use Cases

S^2E can be used for many challenging software development tasks; we present some of them in this section.

Testing in complex environments: Real programs, like Firefox, link against many libraries. Running a real program with symbolic values requires executing the libraries/OS/etc. in a way that can handle symbolic values. While state-of-the art symbolic execution engines could do a one-way conversion of symbolic data into concrete data upon calling a library, they cannot handle the return path; e.g., it is not possible for a program to pass a buffer pointer to `read()` and then read the returned data as symbolic. In S^2E, the developer can specify the program of interest and S^2E will execute it symbolically and correctly regardless of how many calls are made into libraries or the kernel. In terms of the symbolic execution tree, this use case illustrates sound pruning of all subtrees rooted at calls to the environment, thus substantially reducing the overall tree size.

Fine-grain module testing: When testing large programs like Firefox, a developer typically wants to focus attention on a particular area of the code, such as a module that is known to deadlock, or code that was recently added or modified (e.g., "the password management module"). S^2E allows indicating which functions or portion of the executable's text segment is to be run symbolically, and then the rest of the program can run concretely. In this use case, all paths that do not intersect with the designated code are pruned away from the symbolic execution tree. Compared to unit-testing approaches [12], S^2E offers substantially more flexibility: execution can enter and leave the module more than once and can span multiple layers (libraries, operating system kernel, drivers, etc.).

Data-driven testing: Often developers modify a data structure (e.g., alter the size of a field in a C++ class) and want to test *all* code that touches that data structure, regardless of which module the code is in. S^2E enables the selective symbolic execution of the code in question, without having to specify the code blocks explicitly. The approach works even in the presence of pointer aliasing and can guarantee that all desired code paths are covered. In this use case, S^2E soundly prunes from the symbolic execution tree all paths that do not touch the modified data structure.

Hybrid-input testing: Another way to select execution paths a priori is by placing constraints on program inputs. For instance, $0 \leq \lambda \leq 99$ may encode knowledge of the ranges of representative values for input λ (e.g., "purchase price will not exceed $99"), and the program will be symbolically executed starting with this constraint. Stricter constraints, like $\lambda = 1$, make the input fully concrete. Some inputs can be fully concrete, some hybrid, and others fully symbolic (i.e., initially unconstrained). Similar testing can be done with grammar-based fuzz testers [7] or hybrid concolic engines [9], but they require that the program be re-run with each set of concrete inputs. S^2E is more efficient, because it does not retrace any of the common execution prefixes. As observed in [9], some bugs lurk deep down in the execution tree, where a regular symbolic execution engine may take long to reach. By using the initial constraints on inputs, S^2E can allow other inputs, such as network packets, to be symbolic, while reducing path explosion: conditionals

that only access concrete values do not fork new states, thus pruning the corresponding symbolic paths from the tree.

Reproducing user-reported bugs: When users file bug reports, they try to provide details about symptoms, inputs, and configuration in an effort to help developers reproduce the failure in-house. Such reproduction is a key first step in debugging the problem, but can be quite difficult, especially in parallel programs. Symbolic execution can search for a path that evidences the failure, and selective symbolic execution can make this search efficient: concrete inputs and configuration parameters trim away paths that could not have occurred during the failure, while all other inputs are symbolic (e.g., data read from disk). In other words, the details in the bug report define an envelope of executions, and S^{2}E searches for the culprit path only within that envelope.

Dynamic failure analysis: Consider a program that is run inside a virtual machine and checkpointed periodically. If the program crashes, S^{2}E can selectively execute it symbolically from the last checkpoint and find the path that led to the present crashed state, entirely automatically. The fact that S^{2}E simulates a full-system symbolic execution enables it to find paths that are specific to the environment in which the program runs, without incurring undue overhead.

Failure avoidance: Analogously to the use case above, S^{2}E can symbolically execute a program a few steps ahead of the current state, to identify any potential failures lurking in the near future, akin to [15]. For example, a program could use S^{2}E to determine whether a deadlock is likely to occur if it takes a particular lock and, if yes, avoid acquiring that lock.

Reverse-engineering programs: S^{2}E can selectively execute symbolically a closed-source binary device driver with high coverage. If we collect its hardware interaction traces, they can be used to reverse-engineer the state machine encoded in the driver. The challenge for S^{2}E comes from the fact that drivers lie deep inside the system and are tightly coupled with the kernel and the hardware. Our S^{2}E engine executes drivers symbolically by providing them with symbolic inputs from *both* the OS kernel side and the hardware device side. As a result, the driver is unaware that the kernel is running concretely and the kernel is unaware that the driver is running symbolically. S^{2}E can even impersonate the hardware, obviating the need for a real device.

3 Selective Symbolic Execution

We view a system (i.e., a collection of applications, libraries, kernel, device firmware, etc.) as one big program. Selective symbolic execution is a way to specify which parts of this big "program" should run concretely and which ones should run symbolically. Such selectivity—running symbolically only the strictly necessary—is a key ingredient to scaling symbolic execution to real systems.

There are two dimensions along which path selection can be made a priori: code and data. In S^{2}E, one can specify a *code region* of interest by indicating an executable name, an object file, or a [min, max] range of program counters in the kernel's or a process' text segment. S^{2}E will then execute that code symbolically and make all variables involved in conditional branches symbolic, thus ensuring that all paths within the code region are explored. One can also designate a *portion of system state* as being of interest, by indicating a data structure by name or a [min, max] address range in the kernel's or a process' data segment. S^{2}E will then mark the data symbolic and symbolically execute all code that reads or writes that data. Everything else runs concretely.

The main challenge is to make this mix of symbolic/concrete data and execution coexist in a way that preserves correctness and is efficient. In principle, we can treat all system state as symbolic, with varying numbers of constraints on each byte range; a "concrete" variable is merely one with a constraint of the form λ = constant. However, there is a substantial efficiency difference when we distinguish qualitatively between the concrete and non-concrete state: instructions operating on concrete state can run natively, while those operating on symbolic state must be emulated. We therefore have a hard boundary between the concrete domain and the symbolic domain, and data must be suitably converted back and forth when execution crosses this boundary. Therein lies S^{2}E's contribution: correctly executing a real system along with the requisite state conversions, in a way that maximizes the amount of native execution.

We illustrate these conversions with an example in which we test a closed-source binary network device driver; the driver machine code runs symbolically, while applications, kernel, and firmware run concretely. To simplify presentation, assume the goal of testing is to check whether the driver reaches any `assert()` statements that cause a crash; if yes, the corresponding driver arguments are saved as a test case. A simple application can provide the test workload.

Concrete → Symbolic: When the application reads data, it invokes `read(fd,buf,len)` in libc, which in turn results in a system call, which leads to a call to the NIC device driver to DMA *len* bytes into a kernel buffer, which will subsequently be copied into the userspace buffer pointed to by *buf*. This requires that the driver write certain values into the NIC's hardware registers. The application calls the library function `read()` with concrete arguments, which eventually get passed to the kernel, which will invoke the `drv_read` function in the driver's interface.

S^{2}E can be used to explore the full execution tree rooted at `drv_read`, in order to fully test this entry point in the driver: S^{2}E converts all `drv_read` parameters to symbolic and explores all paths. Alternatively, one may want to test the driver only on specific workloads, in which case all or some of the parameters to `drv_read` are kept concrete.

`drv_read` parameters are not the only inputs to the driver. First, there are hardware responses, which S^{2}E converts to symbolic. Second, there are non-data inputs, such as timer and hardware interrupts, which to our knowledge

have never before been considered in the symbolic execution literature. A "symbolic interrupt" has a symbolic time-of-delivery, i.e., could be delivered to the driver at any point, subject to constraints derived from the execution.

Regardless of whether `drv_read` parameters are kept concrete or symbolic, all other inputs are symbolic. Thus, symbolically executing with symbolic parameters will explore all paths rooted at the `drv_read` entry point. Conversely, symbolically executing with concrete parameters will explore all possible ways in which that concrete call could execute, finding all its success and failure paths.

Notice that the presence of the real hardware is not required, since the returned results can all be simulated by the S^2E engine, based on what the driver expects. This is akin to providing an implicit model of the hardware, which we refer to as "symbolic hardware" in §4. Of course, there may be hardware behaviors that the driver is not expecting, which cannot be simulated without the presence of the real hardware. Providing symbolic interrupts requires simply a general model of hardware to be embedded in the S^2E engine; this model encodes generic behaviors, such as following a DMA setup with an eventual DMA completion interrupt. S^2E identifies constraints on the time of interrupt delivery, corresponding to equivalence classes of interrupt handling behavior. Symbolic interrupts are a key feature for finding concurrency bugs in kernels and device drivers.

Symbolic → Concrete: Two important challenges remain: First, the symbolic domain (driver) must return a concrete value to the concrete domain (kernel). Second, the symbolic domain may call into the concrete domain, such as when the driver calls `kmalloc` to get a temporary buffer, or when it places values into the hardware registers. All of these require converting symbolic values to concrete ones.

In the first case, the only consistency requirement is that the returned value be feasible given the supplied concrete parameters. This is satisfied by concretely executing the driver in addition to its symbolic execution and returning the concrete result. Since the symbolic execution tree contains a superset of the paths that can be executed with the concrete parameters, S^2E could theoretically just return a value corresponding to a path whose symbolic inputs satisfy all constraints imposed by the concrete parameters. Using the additional concrete execution, however, allows the result to be returned before the symbolic execution completes; the latter can continue asynchronously in the background.

In the second case, when the symbolic domain calls into the concrete domain, symbolic call arguments (e.g., λ) can be concretized, as above, by choosing concrete values for them that satisfy the current constraints (e.g., setting $\lambda = 5$). But this concretization must correspondingly subject all subsequent paths to the added $\lambda = 5$ constraint.

This constraint can limit the number of paths that will be explored upon return from the concrete domain. Therefore, S^2E keeps track of which constraints are due to concretizations vs. due to branches. If, at a subsequent point, a concretization constraint limits the choice of paths (e.g., $\lambda = 5$ may prevent S^2E from taking a then-branch which requires $\lambda \neq 5$), then S^2E remembers this juncture. S^2E uses the constraint solver to find a minimal set of additional concrete values for λ that would permit the exploration of all skipped paths; it then re-executes the calls into the concrete domain with these values and explores the remaining paths, provided the path constraints are satisfiable.

If these repeated calls have side effects (such as `kmalloc`), they can perturb the concrete domain (e.g., run out of memory), so system state must be forked at the point when the extra calls are made. As will be seen in §4, S^2E has full control over system state, so this is feasible. However, it can be expensive, so we use on-demand conversion (described below) and a caching scheme (described in §4).

On-Demand Conversion: There exist many situations in which, although execution is crossing the concrete/symbolic boundary, conversion of the data is not necessary. Consider, for instance, the case of an application that reads data from a socket and writes it out to another. The network data crosses from the concrete domain of the NIC into the symbolic domain of the driver, then into the concrete domain of the kernel, libc, and the application, then back down through libc, kernel, into the symbolic domain, and back out to the NIC. If the data is merely copied several times from one buffer to another and never really used in any control flow decisions, there is no reason to make it symbolic. Similarly, if a driver allocates a buffer by calling the kernel, then uses a kernel function to copy some symbolic data into that buffer, and then uses the data without the kernel ever doing anything but copying it from one place to another, there is no need for concretizing the symbolic data. A similar situation arises when a complex data structure is passed between symbolic/concrete domains, but only one field of the data structure is actually read, which means that only the accessed field needs to be converted.

For this reason, S^2E performs all conversions on-demand. Conceptually, each memory byte and CPU register has associated metadata indicating whether it is concrete or symbolic and, if symbolic, it also contains the associated constraints. When the bytes are copied from one place to another, by either concretely or symbolically running code, the associated metadata is copied as well. The conversion is performed only when the data is read as part of a branch condition or when doing arithmetic on it, i.e., when the value of the data would actually make a difference. S^2E must keep track of where metadata was copied from, because a conversion must propagate to all related memory bytes. In this way, concretely running code can handle symbolic data transparently.

Needless conversions can be costly, as illustrated earlier, and on-demand conversion reduces their number. Static analysis could reduce needless conversions even further, by

analyzing the code and determining a priori which variables are guaranteed to never require conversion.

4 S^2E Prototype for x86 Binaries

We are currently developing a S^2E engine that can selectively execute x86 binaries symbolically. We opted for machine code-level execution in order to have maximum flexibility, including the ability to use S^2E on closed-source systems, such as commercial applications and OSes.

Our prototype builds upon the QEMU virtual machine [1] and the KLEE symbolic execution engine [3]. QEMU uses dynamic binary translation to turn guest machine instructions into host-suitable instructions; it supports a wide variety of guest architectures, including x86, MIPS, and ARM. Our prototype currently supports only x86 binaries, but it is straightforward to extend it to other architectures. KLEE is an efficient engine for symbolically executing LLVM [8] bytecode. As will be seen below, this combination enables S^2E to take full control of system state, including interfacing with the hardware.

We wrote a new target for QEMU that dynamically translates x86 into LLVM, building upon an earlier effort [13]. We modified QEMU to select the guest x86 instructions that need to be executed symbolically and translate them to LLVM and then pass them on to KLEE; all other instructions run untranslated. As a further optimization, even from among the instructions that must run symbolically, we translate to LLVM only those that operate on symbolic registers or memory. Symbolically-executing code that operates on concrete operands runs natively; we borrowed this optimization from earlier work [4, 6, 2].

The key element that makes S^2E practical is our *shared representation of machine state*, used jointly by QEMU and KLEE. The original QEMU virtual machine manages the state of the virtual CPU, VM physical memory, and VM devices. KLEE operates on roughly the same type of state, but uses different data structures. We modified QEMU to use KLEE's state store, such that the symbolic domain (KLEE) can be kept synchronized with the concrete one (QEMU) with no copying. The concrete state can now be forked whenever required by the symbolic execution. By operating directly on the VM's physical memory (instead of the guest OS' virtual memory), S^2E can seamlessly support IPC and shared memory both within and between the concrete and symbolic domains. Zero-copy and the direct representation of physical memory sets S^2E apart from Bitscope [2].

We use a hierarchical caching scheme to speed up memory lookups. The frequent state-forking inherent in symbolic execution leads to rapidly growing trees of machine state objects; KLEE mitigates this through copy-on-write semantics for memory objects. This means that an object representing the machine state will contain pointers to objects in the parent states. When the hierarchy becomes deep, as is the case for full-machine symbolic execution, the chains of parent pointers can become long, making lookups expensive. In S^2E, every time a lookup is resolved through a chain of pointers to an ancestor, we update the current memory location's parent pointer to point directly to the eldest ancestor. On all subsequent lookups, the chain of parent pointers will be short-circuited by this updated pointer.

As mentioned before, S^2E can run drivers entirely on symbolic hardware. Initially, we modified individual virtual devices (NE2000 network card, USB devices, etc.) to return symbolic answers. Although the effort was minimal (~30 minutes for each device), we preferred a unified layer in QEMU that mediates all accesses to hardware. In this layer we return the symbolic hardware data as well as generate the symbolic interrupts. With this layer, it is not necessary to have the hardware at all, be it virtual or real. Note that the use of modified virtual devices still presents the advantage of being able to store symbolic data in them. E.g., a disk could store metadata-enhanced data, which captures whether bytes are concrete or symbolic and, if symbolic, include the associated constraints. When applications subsequently read this data, S^2E could properly interpret the metadata and perform any conversions that may be required.

Preliminary Results: We successfully used our S^2E prototype to symbolically execute closed-source device driver binaries in full Windows, with the goal of reverse-engineering them. The smallest one, a NE2000 network driver, calls 37 different kernel functions (spinlock operations, memory allocation, handler registration for I/O and timers, NDIS management, HAL calls, etc.). Modeling each of these kernel functions would have been prohibitively labor-intensive. Measurements using the Dhrystone benchmark in S^2E on a quad-core Intel Xeon CPU with 4GB of RAM resulted in a 1.7x slowdown compared to unmodified QEMU, suggesting that S^2E is feasible.

5 Discussion

The selective symbolic execution technique is not tied to the implementation sketched in this paper. An alternative implementation can be applied at the source code level, with the transitions between concrete and symbolic being embedded in the compiled executable. Yet another approach is to use binary instrumentation to control the domain transitions. Of course, the underlying representation of machine state will likely be different from the one presented here.

Even though running a program on a full system inside S^2E obviates the need for modeling most of the program's environment, there are cases where such modeling can help. For example, Windows maintains its registry in compressed form. If a test application writes symbolic data to the registry, the current S^2E would accumulate many superfluous constraints on that data as it gets transformed and compressed by the OS. If, instead, we used a model of the registry, the constraint metadata would no longer be necessary.

Repeated calls to a function (e.g., a library API call) that

is symbolically executed could result in large amounts of redundant exploration, especially if the parameters are symbolic every time. S^2E could save to disk the execution trees of functions that have no side effects and page them back in when the function is re-executed; this would speed up computation of the path constraints and of the final state. In essence, an execution tree is an explicit representation of the function encoded by a body of code, so it can be reused any time that code is invoked, as was done in [5].

6 Related Work

S^2E builds upon previous efforts to scale symbolic execution. One of the first was concolic testing [14], which runs a program concretely, while at the same time collecting path constraints along the explored paths. These constraints are used to find inputs that would take the program on alternate concrete paths. Hybrid concolic testing [9] combines random input generation with symbolic execution to improve coverage over classic concolic testing.

Symbolic Java Path Finder (SJPF) [12] is aimed at unit testing: it executes a program concretely until the target unit of code is reached, at which point execution switches to symbolic. Since SJPF was designed to test functionally-independent units, it does not track symbolic data across the symbolic/concrete boundary.

S^2E can be viewed as a generalization of prior work: we aim to provide the functionality of these previous systems within a single framework. Additionally, S^2E provides functionality that was previously not available, such as whole-system symbolic execution, symbolic interrupts, implicit modeling of the environment, etc.

Mixed-mode execution—running natively instructions that do not involve symbolic operands—first appeared in EXE [4] and DART [6], then in Bitscope [2]; we use it as an optimization. Mixed-mode execution by itself does not provide the necessary conversions and state tracking that enable selective symbolic execution.

Finally, we complement earlier work on modeling the environment [12, 2, 10, 3] with S^2E's symbolic state tracking and on-demand concretization.

7 Conclusion

There exists a spectrum of executions, framed by concrete execution on one end and symbolic execution on the other end, each one with its pros and cons. We introduce selective symbolic execution (S^2E) as a technique for navigating this spectrum in order to optimize the tradeoff between what users want vs. what is efficient to execute.

S^2E offers the illusion of full-system symbolic execution, while symbolically running only the strictly necessary. S^2E helps scale symbolic execution to enable program behavior analysis, bug finding, and test generation for real systems in real environments, without requiring explicit modeling of these environments.

S^2E provides the illusion of symbolic execution of an entire software stack, including applications, libraries, OS kernel, device drivers, and even firmware. We argue that the approach is practical and could open the door to the use of symbolic execution for much larger systems than has been done to date, which could bring symbolic execution into the realm of testing real, general-purpose software.

Acknowledgments: We thank the anonymous reviewers and Olivier Crameri, Rupak Majumdar, Ryan Johnson, and members of DSLab for their feedback. We thank Daniel Dunbar and Cristian Cadar for their support of KLEE.

References

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

[2] D. Brumley, C. Hartwig, M. G. Kang, Z. L. J. Newsome, P. Poosankam, D. Song, and H. Yin. BitScope: Automatically dissecting malicious binaries. Technical report, Carnegie Mellon University, 2007.

[3] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th Symp. on Operating Systems Design and Implementation*, 2008.

[4] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *13th ACM Conference on Computer and Communications Security*, 2006.

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

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

[7] P. Godefroid, M. Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Network and Distributed System Security Symp.*, 2008.

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

[9] R. Majumdar and K. Sen. Hybrid concolic testing. In *29th Intl. Conf. on Software Engineering*, 2007.

[10] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy*, 2007.

[11] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *8th Symp. on Operating Systems Design and Implementation*, 2008.

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

[13] T. Scheller. LLVM-QEMU Google Summer of Code. http://code.google.com/p/llvm-qemu/, 2007.

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

[15] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Symp. on Networked Systems Design and Implementation*, 2009.