Integration Testing of Protocol Implementations using Symbolic Distributed Execution

Raimondas Sasnauskas, Philipp Kaiser, Russ Lucas Jukić, and Klaus Wehrle
Communication and Distributed Systems (ComSys)
RWTH Aachen University, Germany
firstname.lastname@comsys.rwth-aachen.de

***Abstract*—Automatism and high-coverage are the core challenges in testing communication protocols in their early development phase. Ideally, the testing process should cope with a large input space, several sources of non-determinism, and heterogeneous operating environments to effectively explore the emerging execution paths. In practice, however, the missing tool support imposes a huge amount of manual effort to perform integrated conformance and interoperability testing of protocol implementations.**

In this paper, we first detail on the protocol testing issues, such as low coverage, missing code and automation, we experienced during the lifetime of an university-industry project. Second, we present *SymNet*, an integrated testing environment which targets the latter limitations using state-of-the-art symbolic execution techniques. Our approach is to interconnect several virtual machines, execute each of them using selective symbolic execution, and centrally coordinate the emerging distributed execution paths. The key challenges are the synchronization of distributed constraints, detection of false positives, and pruning of redundant execution states. We detail on SymNet architecture, show its applicability to real-world protocol software, and discuss future research directions.

I. Introduction

Implementing and testing new communication protocols and interactive systems is well-recognized to be a difficult task for many software developers. Ambiguous protocol specifications, distributed execution, node and network level non-determinism make the testing process very labour-intensive. In addition, achieving high code coverage, especially in testing of exceptions, is hard, resulting in insufficient manual testing.

Being a partner in an university-industry project [1] our goal was to develop a municipal Wi-Fi network based on Wi-Fi sharing. The core technique [2] of the project is based on the Host Identity Protocol [3] (HIP) which supports key exchange for secure communication, mobility, authentication, and enables multihoming in IP based networks. The implementation of HIP [4] was deployed on various devices ranging from ordinary PCs to tablets and smartphones. During the project phase, we experienced a number of typical debugging issues in a distributed setting. Our test cases were covering only a small part of the possible execution paths, hence, small changes in the test setup revealed new problems with the software. Moreover, some of the bugs (e.g., node crash or wrong protocol state) appeared only after prolonged operations making it difficult to explain and narrow down their root causes since the distributed state was lost. A further challenge was to perform HIP interoperability tests with the proprietary HIP implementation of our industry partner which was deployed on their wireless router. These manual integration tests usually come late and revealed some misinterpretations of the protocol specification.

To support the developers with automated and high-coverage protocol testing in such situations, we propose to analyze the distributed execution of different implementations of protocols using selective symbolic execution (S^2E) [5]. This technique allows efficient symbolic execution of unmodified software including its operating environment. To extend this technique for protocol testing, we connect several S^2E instances over the network and centrally coordinate the distributed testing process. We also propose to use simple assertion-like invariants on distributed protocol states to detect potential violations during execution.

In this paper, we describe SymNet (Symbolic Network), a testing environment for unmodified protocol implementations running in diverse operating systems. SymNet executes communicating protocol instances on symbolic, i.e., "any", input and thus achieves high code coverage giving the developers early feedback on their software behavior. To symbolically execute unmodified operating systems, SymNet employs the S^2E framework [5] built on the QEMU machine emulator [6] and the symbolic engine KLEE [7]. If SymNet detects local errors or violations of distributed invariants during testing, it generates concrete test cases leading to the failing scenarios and reports them to the developer for later analysis.

The core contribution of SymNet is the symbolic distributed execution of communicating S^2E instances in a real network setup. To achieve this goal, we combine time synchronization of VMs, distributed constraint synchronization, and aim at state reduction techniques to efficiently explore distinct distributed execution scenarios. We demonstrate the effectiveness of SymNet with an insidious bug detected in HIP during our preliminary evaluation.

In the next section we introduce the basic concept of our testing environment. Section III details on the design of SymNet and Section IV presents its prototype implementation and preliminary evaluation results. We relate our approach in Section V and discuss the future work in Section VI.

978-1-4673-2447-2/12/$31.00

II. Basic Concept

In this section we present the basic idea of SymNet and give an overview of the testing process. Figure 1 depicts a typical test setup in SymNet during the protocol development life cycle. The protocol implementations are deployed at several nodes in the testbed, each running the same or different operating system (e.g., Linux, proprietary) in a virtualized environment under QEMU. The nodes might be connected either directly or in a multihop fashion to test different protocol operating modes. SymNet executes unmodified OS images inside virtual machines which can be interconnected in various ways. Hence, the complete test scenario can be emulated inside one machine or spread over several nodes in the lab.

SymNet employs selective symbolic execution to automatically generate test cases at high-coverage. Symbolic input can be injected at any place in the system (e.g., packet headers, configuration options). SymNet serializes symbolic data and the corresponding constraints behind VM boundaries and delivers them to other VMs in the network. For example, one typical test scenario is to mark the header (or parts of it) of the initial data packet to be symbolic and track the emerging distributed execution paths which should cover the state machine of the tested protocol. While manually covering even small input space is time-consuming, SymNet covers these inputs automatically.

For test specification and execution control SymNet uses a central coordination unit which has the global view of the tested network. This unit implements a round-based time barrier algorithm to synchronize the emerging execution paths of involved VMs. After each time slice execution all VMs stop and wait for further commands before continuing the testing process. For example, if any of the specified test cases fail, SymNet can put the distributed system on hold allowing the developers to analyze the VMs using well-established tools (e.g., attach GDB to a stopped instance of a VM). At each round, the central unit of SymNet collects interesting data from all VMs to feed it to an assertion checking module.

In summary, the overall testing process in SymNet consists of the following steps: (1) The developers decide which input should be marked as symbolic and which VM memory regions should be extracted for distributed invariant checking; (2) The central unit is configured with the number of VMs and the distributed invariants; (3) The testing process runs automatically writing the results to a file for later analysis. In the following section, we detail on SymNet main modules, discuss the architectural challenges, and present our solutions.

III. System Overview

SymNet combines selective symbolic execution [8] and symbolic distributed execution [9] to enable transparent and automated testing with as little manual intervention as possible. Both approaches were extended to support the architecture presented in Section II.

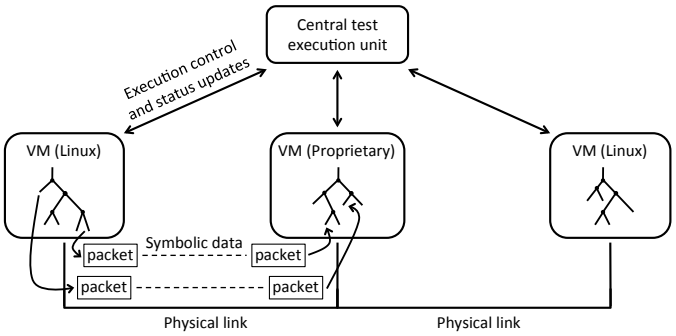


Fig. 1. Basic overview of an exemplary test setup in SymNet. The software to be tested (e.g., different protocol implementations) is deployed inside diverse virtual machines which are connected in a desired way. The testing process is started and controlled by a central unit which collects execution events and issues control commands. At the network level, (symbolic) packet data is delivered to the matching execution path(s) on the destination VM(s) via an emulated or physical link.

A. Selective Symbolic Execution of Protocols

The idea of selective symbolic execution is to symbolically execute selected software parts while executing the rest of the system concretely. This way, S^2E is able to analyze complete operating systems without the need of environment modeling or manual code instrumentation. This technique has been successfully applied to performance profiling, automated testing of user-mode binaries, and reverse engineering of proprietary software [5].

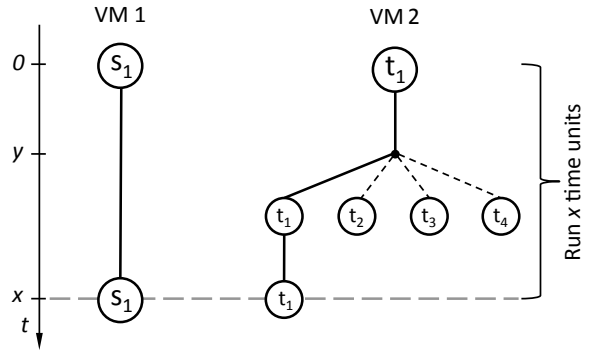
Instead of analyzing a single operating system, i.e., a single protocol instance, SymNet supports the execution of several communicating S^2E instances[1] in parallel containing different OSs (cf. Figure 1). This architecture poses a number of challenges.

Central coordination: Currently, each S^2E instance represents a distinct analysis process with alternating symbolic and concrete execution phases. During the execution of an active path, a number of new execution paths may emerge which are then selected by an internal search strategy for execution.

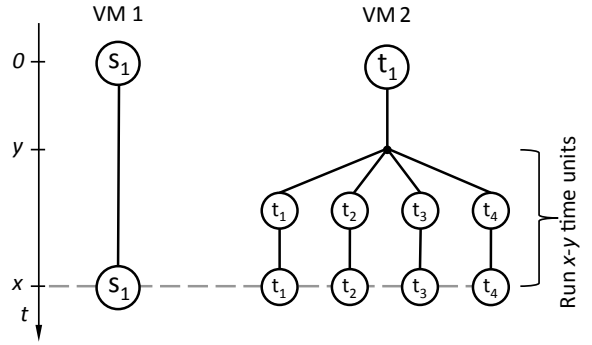
In SymNet, we move the execution control to a central unit in the network to have a global view on the tested network. This requires us to periodically signal execution events (e.g., local branching instructions) of each S^2E instance to a central location and process received control commands (e.g., select next execution path) locally. Thus, we designed a simple protocol in JSON format which is both human readable and easy to process. For example, if the root execution state branches once, SymNet emits the following event to the central coordination unit: {*"event"*: {*"onStateFork"*: {*"parentState": 0, "newStates": (1)*}}}

Time synchronization: By transparently switching between concrete and symbolic execution on demand, S^2E gives an illusion of a full-system symbolic execution. Naturally, the execution speed of software instructions accessing symbolic values is slower compared to concrete instructions because they have to be executed inside a symbolic engine as opposed

[1]We use the term S^2E instance and VM interchangeably.



(a) VM 1 and VM 2 are instructed to run a time slice of x time units. VM 1 starts the execution and stops at time x in the concrete domain. VM 2 starts the execution and hits a symbolic condition resulting in a switch from concrete to symbolic domain. The evaluation of the symbolic condition at time y leads to the creation of three new execution paths (t_2, t_3, t_4). Finally, the original execution path t_1 returns to concrete domain and reaches its time barrier x.



(b) After the execution paths s_1 (VM 1) and t_1 (VM 2) reach the time barrier x, all remaining execution paths on all VMs are instructed to consume the remaining time budget $x - y$.

Fig. 2. An exemplary scenario of VM synchronization using a time barrier algorithm. SymNet detects new execution paths and executes them until the synchronization point is reached.

to native CPU. Consequently, if we connect several communicating S^2E instances for testing, the execution of each VM will switch from concrete to symbolic execution and vice versa at different moments in time. This fact inevitably leads to time drifts between the VMs, falsifying test results [10].

To overcome this issue, SymNet employs a round-based time barrier algorithm to synchronize S^2E instances. The idea is to repeatedly distribute time slices, letting all active VMs proceed to the same virtual time (cf. Figure 2(a)). This way, we can compensate the time drifts occurring due to the switch between concrete and symbolic domain. If the execution branches within a time slice, all emerging execution paths have to finish the remaining time budget of the active time slice until the synchronization point (cf. Figure 2(b)).

In SymNet, VMs do not send out packets immediately during the time slice. Instead, we buffer outgoing packets within a time slice first and deliver them at the beginning of the next time slice, because we do not know in advance which of the execution paths on the target VM should receive the data (cf. Section III-B). As a result, delaying packets increases the expected RTTs between the VMs: Regardless the transmission

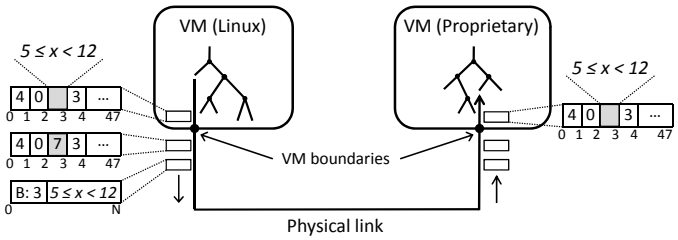


Fig. 3. An exemplary constraint serialization in SymNet. One of the execution paths on the left VM is about to send a packet to the target VM on the right. Assume that the third byte of the packet is symbolic and is constrained with $5 \leq x < 12$. Before concretizing the third byte upon leaving the VM domain, we serialize the constraint and transmit an additional packet carrying the result. Afterwards, the original packet is transmitted including the concrete value, say 7. At the receiving VM, the first packet is deserialized, the constraint is added, and the symbolic data is written to the third byte of the second packet which is then passed to the VM.

time within a time slice, all packets are delivered after the next synchronization point. This artificial delay does not change the temporal ordering of the exchanged packets, however, if the protocol is sensitive to packet interarrival times or specific delay we might lose the corresponding runtime behavior. Decreasing the size of the time slice decreases the introduced delay, but at the same time increases the synchronization overhead between the VMs and the central coordination unit making the testing process slower. In our experiments, the observed overhead ranged from 0,3% for 1000ms time slice to 9% (20ms) and 61% (1ms). Therefore, it's up to the user of SymNet to make a trade-off between the delay granularity and performance overhead.

Symbolic packets: As soon as symbolic data (e.g., packets containing symbolic values) is about to leave the boundaries of a VM, it must be concretized since the real-world is not able to handle symbolic data. However, by choosing one concrete input we lose test cases which might lead to interesting protocol behavior or even corner-cases of execution.

To handle outgoing data containing symbolic values we use (distributed) constraint synchronization: If a VM is about to transmit symbolic data, we extract and serialize the constraints on symbolic data into a text-based format before the data gets concretized. Afterwards, both the serialized constraints and the concretized packet are transmitted to the target VM via a physical link. The receiving S^2E instance deserializes the received constraints, merges them into the constraint set of the receiving state, and writes the symbolic data to the according bytes of the received packet (cf. Figure 3) before passing the packet to the OS network stack. By merging constraints upon communication we (1) avoid constraint over-approximation and (2) effectively prune any false-positives if the constraints after merging are not satisfiable. Note that nodes in the network might constrain the exchanged symbolic data with contradictory constraints during symbolic execution leading to false positives.

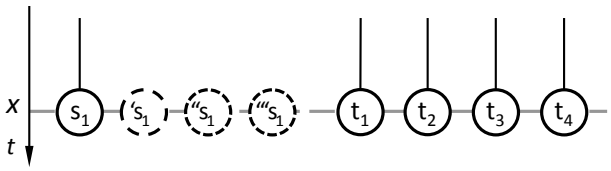


Fig. 4. State mapping conflict resolution in the scenario depicted in Figure 2(b). After the execution of the time slice, TRON detects three new execution paths, namely t_2, t_3 and t_4. Hence, TRON calculates the cross product of the state sets $\{s_1\}$ and $\{t_1, t_2, t_3, t_4\}$ resulting in the following network scenarios: (s_1, t_1), (s_1, t_2), (s_1, t_3), and (s_1, t_4). Consequently, TRON instructs VM 1 to copy state s_1 three times (states with dashed circles) and updates the respective distributed scenarios accordingly to (s_1, t_1), $('s_1, t_2)$, $(''s_1, t_3)$, and $('''s_1, t_4)$.

B. Symbolic Distributed Execution of Protocols

So far, we have presented the challenges we were facing to connect and synchronize an arbitrary number of S^2E instances. Next, we will detail on *TRON* (Time SynchRONizer)—our central test execution and coordination unit (cf. Figure 1)—which performs the following tasks.

State mapping execution: In addition to time synchronization, SymNet has to tackle the following problem efficiently: If an execution path on one VM is about to send data to its destination VM, which execution path on the destination VM should receive the data? We refer to this problem as *state mapping problem* which was presented in [9]. Assume that the execution path t_2 on VM 2 in Figure 2(b) is about to send data to VM 1. In this scenario, we must fork the execution path s_1 on VM 1 before delivering the data. Otherwise we would be unable to deliver a potential future packet from, e.g., t_3 without causing logical violations.

To perform the state mapping centrally, TRON first collects events from each VM such as execution path branching and termination during the execution of a time slice. Second, after all VMs have finished their time slices, TRON resolves the conflicts (if any) by implementing the Copy on Branch (CoB) algorithm [9] which builds a cross product of all execution paths emerged during a time slice execution (cf. Figure 4). By doing so, TRON calculates and triggers the required number of state forks on the according VMs. Afterwards, we always have a consistent symbolic distributed execution before we continue with the next time slice distribution. However, CoB is inefficient since the resulting state space contains two redundant execution states, namely $''s_1$ and $'''s_1$. For the transmitting state t_2 only one copy of s_1 is necessary to avoid logical conflicts. The SDS algorithm from [9] detects and avoids such redundancies, but it is not yet implemented in SymNet.

Distributed searcher: To effectively explore emerging execution paths during testing, S^2E uses different searchers (e.g., DFS, random etc.) to switch between the execution paths. However, these searcher strategies operate locally without any knowledge of the other VMs in the network. Therefore, in SymNet we give TRON the full control over the path selection at each new time slice. This is reasonable since only TRON has global knowledge about matching execution paths and their progress on all VMs. In our example in Figure 4, TRON could choose the next distributed scenario randomly by picking the state tuple $(''s_1, t_3)$ and instructing the VMs to switch to execution paths $''s_3$ and t_3, respectively.

Distributed invariant checking: S^2E already offers a plethora of plugins to monitor the execution of software inside a VM. For example, it is straightforward to detect assertion failures and generate concrete test cases for the respective execution paths. In addition, SymNet offers the developers to specify invariants on distributed system states. First, the developer has to determine which VM runtime information is interesting and instruct S^2E plugins to signal this information periodically to TRON. Second, TRON provides a simple interface to parse and check the received data at the VM synchronization points using simple assertions. Once a distributed invariant is violated, SymNet generates a test case giving the developer concrete input values leading to the same failure when replayed concretely (assuming determinism).

Test case generation: When an execution path terminates (e.g., crash, assertion failure), S^2E generates a test case by feeding all path constraints into a constraint solver and asking for concrete values that satisfy the path conditions. In SymNet, we cannot generate test cases directly since there might be further data constraints transitively created by other VMs in the network during communication. Thus, once a (distributed) failure occurs, S^2E notifies TRON which in turn collects the corresponding constraints from the execution states of the active scenario and delivers them to the VM containing the violating path. Afterwards, we merge the collected constraints, check the result for satisfiability, and generate a test case (e.g., concrete packet header), which leads to the same failing distributed scenario when replayed. Finally, the concrete test case values are emitted to TRON and logged for convenient post-mortem analysis.

C. SymNet's workflow

In summary, the testing process in SymNet consists of the following steps. The developers setup VM images containing their ready-to-test protocol implementations. In addition, S^2E plugins are configured by the developers to extract the software runtime information which will be signalled to TRON for distributed invariant checking. Next, TRON is configured with the number of VMs, distributed invariants to check, and the time of the test run. Note that such tests are configured only once, hence, every following test can be run automatically using either the same or slightly different parameters. Finally, VMs and TRON are started, thereby initiating the testing process:

1) If the test time is over, TRON exits shutting down the VMs. If the test's end time is not reached, TRON distributes a time slice to all VMs and any buffered packets from the previous time slice execution are delivered to the VMs.
2) During time slice execution, any outgoing packets are buffered and TRON is updated with any runtime events such as execution path branches or assertion failures.

3) At the end of the time slice, all VMs, i.e., the active execution paths, stop and wait for further commands from TRON. If necessary, TRON performs state mappings or triggers test case generation if, for example, a (distributed) failure has been detected. Afterwards, TRON searcher selects the next distributed scenario to execute and returns to step 1).

IV. Prototype and Preliminary Evaluation

SymNet prototype implementation consists of the S^2E framework[2] extensions based on plugins, TRON acting as a central instance, and a packet tunnel for packet buffering. Moreover, we applied several ideas from our tool KleeNet[3], which is a symbolic distributed execution tool for self-contained distributed systems, e.g., discrete event simulators.

For the preliminary evaluation, we deployed two Linux VMs (Ubuntu 11.04) running on the same multi-core machine. Nonetheless, although not tested, SymNet supports the different VMs to be spread over the network. On both VMs we installed and started the HIP daemon[4]. Additionally, we prepared a `ping6` command on the first VM to trigger a HIP association establishment to the second VM. At the same time, parts of the header of the initial `I1` packet of HIP are marked by S^2E to be symbolic. We deployed TRON on the same machine as well, configuring it with the two VMs and the testing time of one emulated minute. In this setup no distributed invariants were specified.

Initially, we marked symbolically the version (4 bits) and the reserved (3 bits + 1 fixed bit) header fields (a total of 1 byte) of the first packet. Our expectation of this simple test case was to detect two distributed execution paths: In the first scenario, we should see a successful HIP association establishment between the VMs followed by ping replies, whereas in the second scenario the initial packet must be dropped due to the invalid header version. Surprisingly, SymNet revealed three distributed scenarios, two of them hitting local assertion failures on the second VM.

As expected, in the first scenario a successful HIP association between the two VMs was established, generating a test case with the concrete value `0x11` which is the default value for the according header fields of a HIP `I1` packet. Consequently, in the second scenario, the HIP daemon on VM 2 fired an assertion indicating an invalid header version and dropped the initial packet. The test case had the value `0x0` for the `ver_res` variable whereas `HIP_VER_RES` is `0x01`:

```
hip_common->ver_res != ((HIP_VER_RES << 4) | 1).
```

The unexpected third scenario triggered a surprising assertion failure on VM 2 with the concrete value `0x10` for `ver_res`:

```
len != hip_get_msg_total_len(hip_common).
```

How can arbitrary header fields influence the message length? After digging into the code we found out that the empty reserved field was used by the `hipconf` tool to internally communicate with the HIP daemon for configuration purposes. To reuse the existing code base, the developers had decided to create configuration packets with the HIP header and fill the reserved field plus the one fixed bit with zeros. Since our `I1` packet contained `0x10` in the header, it was first identified as a configuration packet (`HIP_USER_VER_RES` is `0x10`) and afterwards failed a sanity check on packet length at a place where it should have not been performed. Obviously, the HIP daemon should not accept any configuration packets from outside since this is a potential security flaw. Finally, our detected scenario appeared to be a common situation in software development: Some of the original developers have left the project and new ones have joined which were not aware of the described code behavior.

This simple testing scenario demonstrates the core feature of SymNet: Executing protocols on symbolic input can discover such insidious corner-cases early in the development phase. We argue that such cases are very difficult to detect using manual or random testing.

We also ran further experiments subsequently marking the fields of the 40 byte *hip_header* struct as symbolic. The number of explored paths on the receiver node ranged from 3 (1 symbolic byte) to 92 (40 symbolic bytes).

V. Related Work

The approaches in the area of deployed distributed systems testing come closest to our work. In this section, we discuss and relate SymNet's ideas to the recent efforts in this area.

D^{3}S [11] is a framework to continuously create and check global snapshots of deployed distributed systems. The protocol states are extracted using binary instrumentation (bound to one OS) in user-space and transmitted to a global verifier with virtual time stamps for correct distributed event ordering. D^{3}S is a post-mortem analysis tool, hence, it is not possible to hold the distributed execution once a failure is detected making further analysis difficult. In contrast, SymNet can transparently access protocol information from any VM and immediately stop the distributed execution on failure detection. Moreover, it employs symbolic input to explore many different distributed system runs in parallel resulting in high code coverage.

To predict and prevent inconsistencies in deployed distributed systems, the authors in [12] present CrystalBall—an approach to deploy a model checker on each of the running nodes in the network and continuously explore the state space on a recent neighborhood snapshot. This way, CrystalBall is able to detect possible safety property violations in advance and actively steer the execution away from inconsistent states at runtime. In contrast, SymNet is not able to explore different event interleavings in the distributed system at runtime since the execution is driven by symbolic input only. Nevertheless, SymNet can automatically explore different input equivalence classes whereas CrystalBall dynamically explores variations of one specific test run and is bound to one domain specific language.

[2] https://s2e.epfl.ch/

[3] https://code.comsys.rwth-aachen.de/redmine/projects/kleenet-public

[4] https://code.launchpad.net/~hipl-core/hipl/trunk (rev: 6277)

To automatically explore the execution paths of a network configuration protocol, the authors of SymNV [13] propose a framework which (1) symbolically executes a network daemon on symbolic packets and (2) replays the concrete inputs checking if the input and output packets match a set of rules derived from the specification. SymNV discovered a number of non-trivial bugs, but it only considers a single protocol instance and its interaction with a modeled environment. By connecting several protocol instances, SymNet is able to detect potential bugs in the distributed state of protocols.

In summary, in SymNet we realize the ideas of KleeNet [14] to enable automated testing of unmodified software running on different—even proprietary—operating systems communicating over a physical link. In contrast to KleeNet, we run VMs instead of simulation, add time synchronization, symbolic data serialization over the network, and a central coordination unit to solve different challenges we discovered during the design process (cf. Section III).

VI. Conclusion and Future Work

In this paper, we presented SymNet, a testing environment for unmodified communication software using symbolic distributed execution. The key feature of SymNet is the ability to analyze protocol implementations running in different operating systems communicating over a real network. Our preliminary evaluation demonstrates SymNet's ability to explore unmodified protocol implementations and thereby uncovering an insidious corner-case in the code under development.

In the future work, we plan to implement the SDS state mapping algorithm [9] which drastically reduces the state space compared to the Copy on Branch algorithm. Consequently, we will evaluate SymNet with larger setups with regard to scalability, distributed code coverage, and testing efficiency. Selective symbolic execution is very resource consuming because the differences between the execution paths grow over the time filling up the RAM. Moreover, even small programs may quickly lead to state explosion during the execution. Therefore, we plan to investigate how much time and resources it takes to explore execution paths of typical protocol implementations and how much code coverage can we achieve in reasonable testing time.

At the time of writing, we are extending our prototype with additional usability features. The goal is to include SymNet into the automatic test suite of HIP which could be run automatically during nightly builds. For this purpose we will first deploy a line topology setup (cf. Figure 1) including the proprietary HIP implementation of our industry partner to test different HIP operation modes. Second, we plan to write a number of distributed invariants which will check our interoperability and specification requirements. On the symbolic input side, we will iteratively choose different parts of the packets for symbolic marking, since large symbolic inputs in packets may quickly lead to state explosion [13].

Finally, we argue that with the recent advances in the integrated tool support (e.g., KLEE, KleeNet, S^2E) and symbolic execution in general [15] this technique has become an attractive and practicable approach for automated high-coverage test case generation.

Acknowledgments

We thank René Hummen for his help and ideas on HIP protocol testing. We also thank Vitaly Chipounov and Oscar Soria Dustmann for helping us to improve the quality of the paper. This work is partly supported by DFG UMIC research cluster of RWTH Aachen University.

References

[1] "Mobile ACcess (German description)," http://www.mobile-access.org.

[2] T. Heer, S. Götz, E. Weingaertner, and K. Wehrle, "Secure Wi-Fi Sharing at Global Scales," in *Proc. of 15th International Conference on Telecommunication (ICT), St. Petersburg, Russian Federation*, vol. 1. Washington, DC, USA: IEEE, 6 2008, pp. 1–7.

[3] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson, "Host Identity Protocol," RFC 5201 (Experimental), Internet Engineering Task Force, Apr. 2008, updated by RFC 6253. [Online]. Available: http://www.ietf.org/rfc/rfc5201.txt

[4] "Host Identity Protocol for Linux," https://launchpad.net/hipl.

[5] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[6] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41.

[7] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[8] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea, "Selective Symbolic Execution," in *5th Workshop on Hot Topics in System Dependability (HotDep)*, 2009.

[9] R. Sasnauskas, O. S. Dustmann, B. L. Kaminski, K. Wehrle, C. Weise, and S. Kowalewski, "Scalable symbolic execution of distributed systems," in *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*, ser. ICDCS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 333–342.

[10] E. Weingaertner, F. Schmidt, H. vom Lehn, T. Heer, and K. Wehrle, "Slicetime: A platform for scalable and accurate network emulation," in *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*, 2011.

[11] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang, "D3S: Debugging deployed distributed systems," in *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*, J. Crowcroft and M. Dahlin, Eds. USENIX Association, 2008, pp. 423–437.

[12] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak, "CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems," in *NSDI*, 2009.

[13] J. Song, T. Ma, C. Cadar, and P. Pietzuch, "Rule-based Verification of Network Protocol Implementations using Symbolic Execution," in *IEEE International Conference on Computer Communications and Networks (ICCCN 2011)*, Maui, Hawaii, USA, 08/2011 2011.

[14] R. Sasnauskas, O. Landsiedel, H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle, "KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment," in *International Conference on Information Processing in Sensor Networks (ACM IPSN/SPOTS)*. New York, NY, USA: ACM, 2010, pp. 186–196.

[15] C. Cadar, P. Godefroid, S. Khurshid, C. Pasareanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic Execution for Software Testing in Practice – Preliminary Assessment," in *International Conference on Software Engineering, Impact Project (ICSE Impact 2011)*, Honolulu, Hawaii, USA, 05/2011 2011.