Building Robust Distributed Systems and Network Protocols by Using Adversarial Testing and Behavioral Analysis

Endadul Hoque and Cristina Nita-Rotaru
Northeastern University
e-mail: {e.hoque, c.nitarotaru}@neu.edu

***Abstract*—We describe our experience over the past five years with building more robust distributed systems and network protocols by using adversarial testing and behavioral analysis. We describe the benefits and disadvantages of both approaches and the design of the tools we have built (Turret, Turret-W, SNAKE, and Chiron). We discuss how we applied them to byzantine-resilient state machine replication, wireless routing protocols, transport protocols, TLS, and IoT implementation of application-level protocols.**

***Keywords*-robustness; protocols; implementations; adversarial testing; behavioral analysis;**

I. Introduction

Most distributed systems and network protocols are designed to meet fault-tolerance, performance, and security goals. Fig.1 shows the typical steps involved in the life cycle of a protocol development. First, the fault-tolerance, performance, and security requirements are specified in informal prose descriptions (*e.g.*, RFC standards). Next, protocol designs are created in the form of well-defined messages and state machines. These designs are then implemented in different languages for various operating systems. Finally, the implemented protocols are deployed in production. During this life cycle, model checking of designs is used to ensure that designs match specifications, and random fuzz testing of implementations is used to check that implementations are free of bugs and vulnerabilities. However, model checking of designs and random fuzz testing of implementations are not sufficient to guarantee that implementations achieve their goals in practice. The informal nature of the prose specification, the increased design complexity, the inconsistent interpretations by developers, and the limited functionality of existing fuzz testing tools mostly focused on conformance testing only, often result in bugs and vulnerabilities that make the systems unusable. Worse, many of these vulnerabilities manifest after the code has already been deployed, making the debugging process difficult and costly.

In our work, we advocate for systematic testing of implementations of distributed systems and network protocols under adversarial conditions as a way to increase robustness against those attacks that are particularly targeted to impair the overall performance of the system. For example, this lack of systematic and *adversarial testing* for protocols and their implementations has resulted in a stream of new bugs and attacks [1]–[4]. Consider TCP, one of the most well studied and well tested network protocols; the list of discovered attacks extends from the mid-1980's to the present day [5]–[10]. Many of these attacks have been discovered repeatedly or rediscovered again in slightly different contexts.

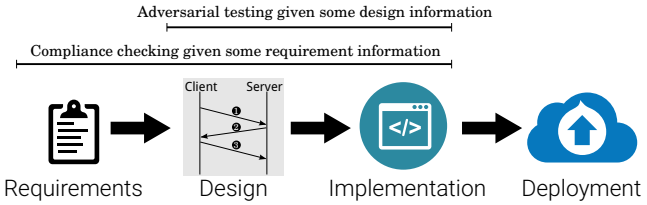

Figure 1. Lifetime of a network protocol development

While adversarial testing is very useful in testing a large class of failures and attacks, it does not cover the large state-space that network protocols executions define, leaving a residue of errors that can manifest deep in an execution and cause noncompliance. As a complementary mechanism, we also advocate for *behavioral analysis* to check compliance of a protocol implementation against given properties that prescribe the correct behavior of the protocol, and determine if the implementation satisfies such given properties or not.

Adversarial testing and behavioral analysis have complementary strengths and make different trade-offs to achieve their goals. Adversarial testing is geared towards finding low-level flaws in implementations caused by different input values [11], [12], thus the main challenge it has to overcome is the large space of input values. It does this by using fuzzing, at the cost of lower coverage [13], [14]. Contrarily, behavioral analysis focuses on high-level logical flaws in the execution of the protocol implementation, thus the main challenge it has to overcome is the large space of execution paths in the code. It does this by using static analysis, at the cost of higher processing overhead.

In this paper, we report on several of our projects focused on increasing assurance in protocol implementations by using adversarial testing and behavioral analysis. Specifically, we first describe several approaches to adversarial testing focused on how the malicious test cases are automatically created, injected, and the resulting attacks are detected. We created several platforms—Turret [3], Turret-W [15], and

DOI 10.1109/SecDev.2016.12

SNAKE [16]—that allowed us to test routing protocols, distributed systems, and transport protocols; they achieve different coverage and efficiency by assuming more knowledge about the system design. We then describe our experience and challenges encountered while creating Chiron [17], a behavioral analysis framework, to detect noncompliance in a network protocol implementation (*secure* or *non-secure*) against a rich set of properties prescribing the correct behavior of the protocol.

II. Adversarial Testing

Adversarial testing pushes implementations to be tested beyond just basic functionality (*i.e.*, examining edge cases, boundary conditions), and ultimately conducts destructive testing. While being effective in many contexts by focusing on finding bugs in processing program inputs (*e.g.*, strings, files), traditional fuzz testing [18] falls short by not considering specifics (*e.g.*, timings, data-layout) of protocol messages critical to performance attacks.

A. System and Attack Model

We consider the implementations of the system under test (SUT), such as distributed systems and network protocols, are conceptually modeled as message passing event-driven state machines. This model is predominant for network protocols and also representative for many distributed systems [19]–[22] implementations. Our focus is on systems that have measurable performance metrics (*e.g.*, throughput and latency), which provide an indication of the progress the respective system has in completing its goals.

We consider attacks against the performance of the SUT where compromised participants running malicious implementations attempt to impair the overall performance of the SUT through actions on exchanged messages. We assume that these attacks create an observable degradation of performance in a small window of time. Formally, we define a performance attack as a set of actions that deviate from the protocol, taken by a group of malicious nodes and resulting in a performance that is worse by a user-provided threshold Δ than in benign scenarios. Note that such performance attacks are a form of denial-of-service (DoS) attacks where the attacker's goal is not to completely take the system down, but instead degrade the performance enough to reduce the practical utility of the system and remain undetected to continue the damage.

The attacker is capable of observing and intercepting the exchanged messages and thus can impair the global system performance. We categorize all malicious actions on messages into two groups: (a) *message delivery actions* that affect when and where a message is delivered, and (b) *message lying actions* that affect the contents of a message.

B. Design Space

At a high level, our approach utilizes an emulated approach as shown in Fig. 2. In an emulated approach, the SUT

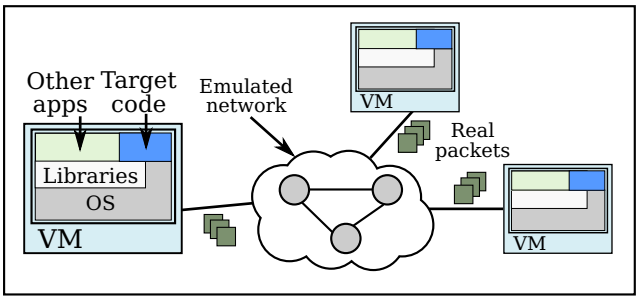

Figure 2. High-level design of our adversarial testing platforms

code is executed in a virtual machine (VM) using the same operating system and libraries as in the deployment, generating network events through transmitting actual network packets in real time. The network communication is emulated such that the network conditions are reproducible and controllable. Virtualization enables a realistic environment for the SUT, and the network emulation enables controllable network communication. Moreover, network emulation can accurately simulate the behavior of deployed networks and network technologies, and can interface with VMs running an actual implementation of the SUT.

Several design questions must be answered: (a) how to create meaningful attack and testing scenarios, (b) when to inject them in the environment, and (c) how to determine if an attack was successful or not. To answer these design decisions, we consider two (often competing) goals: scalability with the large space of test cases and coverage in terms of possible attacks to be captured.

Attack generation. Fuzz testing [18] was shown to be an effective black-box testing, where well-formed inputs are mutated to generate various test cases. However, this technique provides a limited coverage of the large attack search-space [13], [23]. Because of considering the SUT as a black-box, Turret addresses the first design question by leveraging *grammar-based fuzz testing* [13], which improves on fuzz testing by taking into account some SUT specific information to prune the search-space during the generation of new test cases. Turret uses the user-provided data-layout of protocol messages to identify *fields* (*e.g.*, `type`) of different protocol messages and apply general mutation techniques as proposed in [11], [12].

Attack injection. An important aspect of determining an attack search strategy is identifying the attack injection points, that is, the points where attacks can be inserted into a test run.

1) Send-packet-based attack injection. One simple approach is to have a malicious proxy intercept each packet generated by the SUT running in the virtual machine, apply any attack desired, and forward the packet on to its destination. This means that an attack injection point is

whenever there is a send for a particular packet type. This approach has the advantage that it is relatively simple and requires only message format information about the SUT. Thus, it works well for protocols with very few messages. It also has several disadvantages: since it is oblivious to the semantics of the SUT, it can repeat performing attacks that have the same semantics for the SUT, thus resulting in redundant executions and lengthening the time required to complete the search.

2) Time-interval-based attack injection. The send-packet-based attack injection applies attacks only on intercepted packets that were generated by the SUT. This approach unfortunately does not support off-path attackers; for example many attacks against transport protocols are mounted by off-path attackers. One approach to provide support for off-path attackers and finer time granularity is to divide the running time into fixed intervals and, for each of these intervals, attempt to inject packets following all basic attacks. This approach has the advantages that it is relatively simple and still assumes only message format information, while being able to handle off-path attackers. The disadvantage is that a small time interval must be used in order to catch many attacks, resulting in testing thousands of scenarios that either do not inject attacks or inject many redundant attacks, based on the semantics of the protocol. Hence, it can also take a very long time to complete the search.

3) Protocol-state-aware attack injection. Another approach to eliminate some of the redundant testing scenarios, to support off-path attackers, and to provide finer granularity for injecting attacks is to take into account the semantics of the protocol when injecting attacks. Such information about the semantics of the protocol can be obtained from its state machine. Many protocols have well documented state machines describing their connection establishment, and in the absence of such documentation, prior work [24] on state machine inference may be leveraged. We propose a state-based search strategy that leverages several characteristics of the protocol state machine to reduce the attack search space. Specifically, we inject attacks at specific states in the protocol execution. Because the protocol state machine defines key points in the operation of the protocol, this approach allows us to quickly gain wide coverage within the search space by focusing on each of these states. We also treat all attack injection points in a state in the same way. This further prunes the number of search paths to be explored. The motivation behind our approach is that two packets of the same type received in the same protocol state usually cause similar results; however, an identical packet received in two different states may cause significantly different results.

Attack finding. Efficiency and scalability are major requirements for the attack finding algorithm. We focus on performance attacks that have measurable effects in a relatively short amount of time. Thus an attack can be detected by comparing the *baseline* performance of the SUT (when no attack is injected) with the measured performance of the SUT when the attack is injected. Several approaches are possible when searching the space of possible attacks.

1) Brute-force search. The simplest approach is a *brute-force search algorithm* as the one showed in Fig. 3 (a). The algorithm relies on a list of all possible attack scenarios (malicious actions for each message type).[1] It first obtains a baseline, then for each attack scenario in the list, it runs the SUT until it encounters an attack injection point (*i.e.*, the sending event of a message with the type from the attack scenario), injects the attack scenario, and obtains the performance. An attack scenario (*i.e.*, action) is determined to be an attack if the difference between the baseline and the performance for that action was higher than the threshold Δ. This approach has the advantages that it is simple and does not require control during the execution of an attack scenario, but just the ability to start and stop the SUT. However, this approach wastes time due to some unwanted executions: (1) in case no attack injection point is encountered (*i.e.*, no message of the type listed in the scenario was actually sent), this approach still runs the entire execution, and (2) in case an attack injection point is encountered, for every malicious action scenario, this approach runs the execution from the beginning even if the performance is needed to be measured only from the attack injection point, not from the beginning of the execution.

2) Greedy search. One approach to avoiding wasted execution is to use a dynamic *greedy search algorithm*. When the algorithm encounters an attack injection point, it branches the execution and obtains a baseline performance as well as the performance for each malicious action for that message type. It then selects the action that caused the largest performance degradation. As an aggressive approach can also make mistakes, higher confidence is obtained by deciding that a scenario is an attack if it was selected more than a certain number of times, which in turn requires additional executions. This approach requires the ability to compare performances of non-malicious and malicious executions that are branched from the same attack injection point.

3) Weighted greedy search. A *weighted greedy* approach improves upon previous solutions by reducing the time required to find an attack to minutes. The algorithm relies on the observation that certain types of malicious actions are more effective than others, regardless of message types. Therefore, it attempts to learn what actions are more likely to be effective and use this information to improve the next search. It also clusters malicious actions into several categories, where each cluster is associated with a weight. Actions with higher weights are tested first.

[1]The malicious actions that can be applied to each messages type depend on the message format, *i.e.*, number of fields and types for each field.

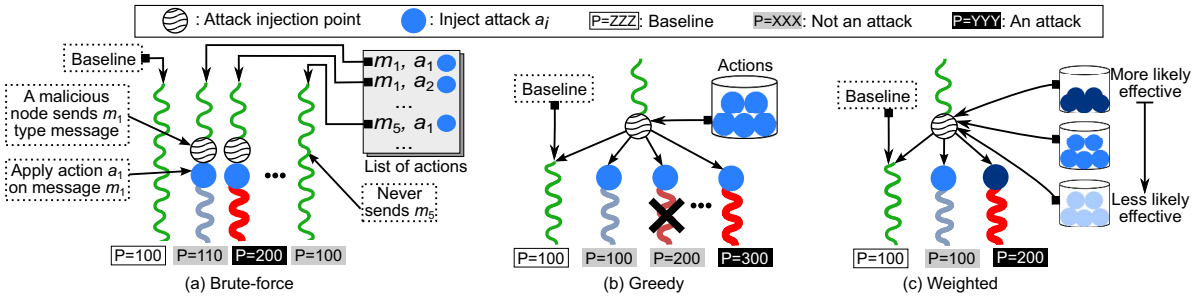

Figure 3. Attack finding algorithms

Execution branching support. More efficient performance attack finding algorithms such at the greedy search and weighted greedy search algorithms described above need to compare between several malicious executions that are branched from an identical attack injection point. We use checkpointing as an approach to save and restore snapshots of the entire environment while finding attacks against the SUT. The details can be found in [3].

Attack detection. We define an action as a successful attack if the difference between the achieved performance and the baseline is higher than a threshold Δ. This threshold is supplied by the user and is selected to meet an acceptable performance of the SUT.

Our tools have a very low or zero of false positives because they use a lightweight emulator that provides a low overhead and performance controlled environment. However, measuring false negatives is non-trivial as it is difficult to obtain the ground truth about the number of vulnerabilities present in a given implementation.

C. Platforms and Results

We implemented several platforms[2] in the design space described above: Turret designed for Byzantine state machine replication distributed systems, Turret-W for wireless routing protocols, and SNAKE for transport protocols.

Byzantine-resilient state machine replication. Turret was implemented utilizing the KVM [25] virtualization infrastructure and the NS3 network emulator.[3] Each participant of the SUT runs in its own VM, and each VM is mapped to a *shadow* node inside NS3. The malicious proxy is placed in NS3, which makes coordination simpler, as all inter-node messages is controlled by NS3. A user provided network configuration file specifies which VMs (and their respective shadow nodes) are malicious so that the proxy intercepts only those messages transmitted by them. The attack finding algorithm is implemented by a controller which is a separate process that communicates with the network emulator and all the VMs. Turret was applied on 5 different distributed system implementations (PBFT [26], Steward [27], Zyzzyva [28], Prime [29], and Aardvark [30]) specifically designed to tolerate insider attackers and found 30 performance attacks, 24 of which were not previously reported [3].

Wireless routing protocols. Turret-W [15] was built for adversarial testing of wireless routing protocols. Turret-W leverages the design of Turret and includes new functionalities such as the ability to differentiate routing messages from data messages, support for protocols that use homogeneous or heterogeneous packet formats, support for protocols that run on geographic forwarding (not only IP), support for protocols that operate at the data link layer instead of the network layer, support for replay attacks, and ability to establish side-channels between malicious nodes. As a result, Turret-W can test not only general attacks against routing, but also wireless specific attacks such as blackhole and wormhole attacks. Our approach is cost effective in comparison with the hardware and manpower required by the approach in [31]. We demonstrated Turret-W by applying it on implementations of 5 wireless routing protocols: a reactive protocol (AODV [32]), a secure reactive protocol (ARAN [33]), and three proactive protocols (OLSR [34], DSDV [35], and BATMAN [36]). We found 1 new and 7 known attacks in AODV, 6 known attacks in ARAN, 5 known attacks in OLSR, 4 new and 7 known attacks in DSDV, and 7 known attacks in BATMAN, for a total of 37 attacks. While most of attacks we found are protocol level attacks, one attack in AODV and 4 attacks in DSDV were solely implementation level attacks, and such attacks could have been discovered only by testing the actual implementations under adversarial environments. In addition, Turret-W can also find bugs that cause performance degradation in benign executions. We discovered 3 bugs in total, 2 in AODV and 1 in ARAN. The bugs in AODV were due to a subtle interplay between the AODV code and the operating system kernel.

Transport protocols. SNAKE was designed primarily for transport protocols like TCP. Like the other platforms, it uses virtualization to run unmodified transport layer implementations in their intended environments and a network emulator

[2]Available at: http://nds2.ccs.neu.edu/autoattack.html
[3]http://www.nsnam.org

to tie these virtual machines together into a realistic, emulated network. Unlike the other platforms, SNAKE uses protocol-state-aware attack injection, and thus it assumes knowledge about the state machine of the SUT. The network emulator intercepts and modifies packets, tracks the current protocol state during execution, and uses this information to create packet-based attacks at specific points in the state machine. SNAKE can be used for many transport protocols, requiring only the description of the packet header formats and the transport protocol state machine as input. SNAKE was used to examine a total of 5 different implementations of 2 transport protocols (TCP and DCCP) on 4 different operating systems. It discovered 9 attacks, 5 of which are, to the best of our knowledge, unknown in the literature. We also compared our state-based attack search with other attack injection approaches and showed its effectiveness in search space reduction.

Curriculum integration. We integrated Turret with a graduate-level distributed systems course offered at Purdue University in Spring 2013 and in Fall 2014 [37]. Turret was used as the testing platform for the programming assignments given in this course. The projects were designed based on Paxos [38], Byzantine Generals Problem [39], and Total Order Multicast [40]. Students were given access to Turret at the beginning of the semester so that they could leverage the platform to test the robustness of their ongoing assignment. Both the students and the instructor tested the same unmodified binary under the same conditions without implementing the necessary code for the malicious test case scenarios. To use the platform, students were given a 30-min demo and supplied with an instruction manual on how to use the platform. Based on the anonymous, *IRB-approved* surveys collected from this course, students rated the effectiveness of Turret in finding bugs as 4.0 out of 5.

III. Behavioral Analysis

Many protocols have to meet complex requirements which involve *temporal properties* that prescribe the correct temporal behavior in response to network events (*e.g.*, arrival of a packet, occurrence of timeout); Detecting noncompliance against such requirements demands for a more in-depth analysis of the protocol behavior than message-based adversarial testing for performance degradation.

A. System Model

We require the source code of the protocol implementation for our analysis. We focus on protocols written in C using an *event-driven* paradigm. Many network protocols designed for IoT devices and popular protocols such as TCP and TLS are implemented in an event-driven paradigm. Such event-driven code typically contains a common entry point function that contains the event-loop. We also assume that the protocol states are explicitly represented by some program variables where the possible values of these variables are drawn from a small finite domain.

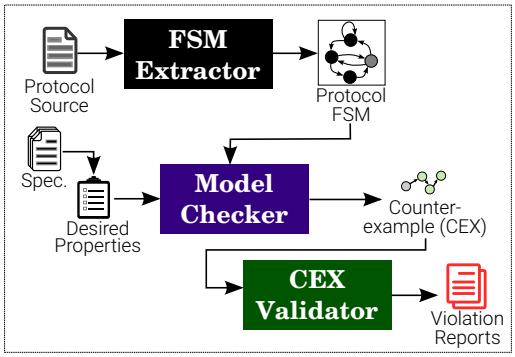

Figure 4. High-level approach of Chiron

B. Design Space

We observe that noncompliance is often caused by how the protocol implementation reacts to network events (*e.g.*, by changing the protocol's internal state, by sending a response). Such reactions of the protocol are described as finite state machines (FSMs) in informal prose specifications (*e.g.*, RFC standards), either explicitly (*e.g.*, DHCP, TCP) or implicitly (*e.g.*, Telnet, TLS). While implementations intend to closely follow the specified FSMs, informal descriptions of the FSMs often leave room for inconsistent interpretations giving rise to errors related to state machines [41].

Execution-based software model checkers [42]–[45] have been applied to detect violation against the properties in a given implementation. They systematically enumerate the entire state-space that the program defines and check the properties to find violations. However, their use of explicit-state model checkers can cause the state-space explosion problem due to the large state-space that a network protocol defines. Therefore, we adopted an abstraction-based approach where a developer can check whether a stateful, event-driven network protocol implementation violates the user provided desired temporal properties. In order to perform this check, we need an abstract model (*e.g.*, FSM) extracted from the protocol source code. A high-level approach is presented in Figure 4.

Several design questions must be answered: (a) how to specify meaningful protocol properties from their specifications, (b) how to extract the state machine corresponding to the protocol, and (c) how to check if a property is met or not in the extracted finite state machine.

Protocol properties specification. Due to its proficiency in capturing the relative chronological order of events succinctly, we use propositional linear temporal (pLTL) to express the desired protocol behavior. Requiring the user to formulate the desired temporal properties in pLTL is a daunting task. Instead, we envision users to write the

temporal properties in the SALT (Structured Assertion Language for Temporal Logic) language. SALT is close to a high-level programming language, containing constructs for frequently occurring property patterns. In addition to being user-friendly and permitting users to express properties using regular expressions, SALT properties can be automatically translated to optimized pLTL formulas using its compiler.

FSM extraction. Manually extracting FSMs from implementations is error-prone and a tedious task. Existing work [24], [46]–[50] has looked at automatically inferring FSMs from implementations. Such inferred FSMs either merely capture the sequences of messages valid in a protocol session or represent the low-level program state machines rather than the protocol's FSMs that we require for noncompliance detection. Therefore, we devise an FSM extraction technique that takes as input the protocol source code (written in C) and outputs an approximated protocol FSM. Our technique leverages a program analysis technique, *symbolic execution* [51], because it precisely simulates a program's execution and explores all possible execution paths while maintaining symbolic information about the program.

Compliance checking. To check whether the extracted FSM complies with the given temporal properties, we use a symbolic model checker. If the FSM violates a property, the model checker generates a counterexample, CEX, (*i.e.*, an execution of the protocol demonstrating the violation) as evidence. Like abstraction-based model checkers [52], [53], our approach may suffer from spurious CEXs due to the abstractions in our analysis. Such spurious CEXs are not be realizable in any actual execution of the protocol. Therefore, we use a two-step validation process to rule out unrealizable CEXs [17]. As a result, any realizable CEX reported is indeed an actual noncompliance instance (*i.e.*, true violation).

Chiron is geared towards detecting noncompliance instead of assuring compliance [54] of a protocol implementation against a temporal property. Any realizable CEX reported by Chiron is indeed an actual noncompliance instance as it rules out any false CEX generated by the model checker. Conversely, when Chiron cannot find any violation, it does not assure that the property holds.

C. Platforms and Results

Based on the design space described above, we implemented Chiron, an automated behavioral analysis framework, to help developers detect noncompliance in event-based implementations of network protocols. Chiron was implemented on top of KLEE symbolic execution engine [55]. For the symbolic model checker, Chiron uses NuSMV [56]. Chiron was demonstrated by applying it on a total of 6 implementations of three protocols: one secure protocol (TLS) and two non-secure protocols (Telnet and DHCP). For TLS, we used the implementation from the OpenSSL library,[4] which is designed for traditional operating systems (*e.g.*, Linux). Due to Chiron's general approach, it can also be applied to protocols implemented for Internet-of-Things (IoT) devices. Therefore, for Telnet and DHCP, we used several implementations from two separate TCP/IP protocol stacks designed for IoT devices: uIP [57] and FNET[5]; they are widely used but have not been extensively studied. In our evaluation, we used 6 properties for TLS, 11 for Telnet, and 7 for DHCP, which are derived from their respective RFCs and documentation. Chiron detected 11 noncompliance instances in total, 2 of which have security implications.

IV. Conclusion

Given the importance of network protocols and distributed systems, it is important to subject their implementations to rigorous proactive assurance increasing techniques before deployment. In this paper, we present two such techniques: (a) adversarial testing that allows implementations to be automatically tested beyond basic functionalities in an adversarial environment and (b) behavioral analysis that helps developers find logical flaws in the protocol execution flow through automatically detecting noncompliance against the user-provided protocol properties. We also report our experience, challenges encountered, and findings while creating and evaluating three adversarial testing platforms (Turret, Turret-W, and SNAKE) and one behavioral analysis framework (Chiron).

Acknowledgment

We would like to thank all the people who were involved in the design and development of the systems presented here (Turret, Turret-W, SNAKE and Chiron): Sze Yiu Chau, Omar Chowdhury, Samuel Jero, Charles Killian, Hyojeong Lee, Ninghui Li, Rahul Potharaju, and Jeffrey Seibert.

This work was supported in part by grants CNS-1223834 and CNS-1421815 from the SaTC program of the National Science Foundation and by grant N660001-1-2-4014 from the Mission-Resilient Clouds program of DARPA. Its contents are solely the responsibility of the authors and do not represent the official view of the National Science Foundation, DARPA or the Department of Defense.

References

[1] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi, "Finding protocol manipulation attacks," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, 2011.

[2] N. Cardwell, Y. Cheng, L. Brakmo, M. Mathis, B. Raghavan, N. Dukkipati, H. keng Jerry Chu, A. Terzis, and T. Herbert, "packetdrill: Scriptable network stack testing, from sockets to packets," in *USENIX Annual Technical Conference (ATC)*. USENIX, 2013.

[4] https://www.openssl.org
[5] http://fnet.sourceforge.net/

[3] H. Lee, J. Seibert, E. Hoque, C. Killian, and C. Nita-Rotaru, "Turret: A platform for automated attack finding in unmodified distributed system implementations," in *Proceedings of International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2014.

[4] F. Gont, "Security assessment of the transmission control protocol," Centre for the Protection of National Infrastructure, Tech. Rep. CPNI Technical Note 3/2009, 2009.

[5] B. Guha and B. Mukherjee, "Network security via reverse engineering of TCP code: Vulnerability analysis and proposed solutions," *IEEE Network*, vol. 11, no. 4, 1997.

[6] V. Kumar, P. Jayalekshmy, G. Patra, and R. Thangavelu, "On remote exploitation of TCP sender for low-rate flooding Denial-of-Service attack," *IEEE Communications Letters*, vol. 13, no. 1, 2009.

[7] A. Kuzmanovic and E. W. Knightly, "Low-rate TCP-targeted denial of service attacks: The shrew vs. the mice and elephants," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, 2003.

[8] R. Morris, "A weakness in the 4.2 BSD Unix TCP/IP software," AT&T Bell Leboratories, Tech. Rep., 1985.

[9] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson, "TCP congestion control with a misbehaving receiver," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 5, 1999.

[10] J. Touch, "Defending TCP against spoofing attacks," Internet Requests for Comments, RFC Editor, RFC 4953, July 2007.

[11] M. Stanojevic, R. Mahajan, T. Millstein, and M. Musuvathi, "Can you fool me? Towards automatically checking protocol gullibility," in *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*, 2008.

[12] H. Lee, J. Seibert, C. Killian, and C. Nita-Rotaru, "Gatling: Automatic attack discovery in large-scale distributed systems," in *Proceedings of Network & Distributed System Security Symposium (NDSS)*, 2012.

[13] P. Godefroid, P. de Halleux, A. Nori, S. Rajamani, W. Schulte, N. Tillmann, and M. Levin, "Automating software testing using program analysis," *IEEE Software*, vol. 25, no. 5, 2008.

[14] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, 2012.

[15] E. Hoque, H. Lee, R. Potharaju, C. E. Killian, and C. Nita-Rotaru, "Automated adversarial testing of unmodified wireless routing implementations," *IEEE/ACM Transactions on Networking (ToN)*, 2016, in press.

[16] S. Jero, H. Lee, and C. Nita-Rotaru, "Leveraging state information for automated attack discovery in transport protocol implementations," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2015.

[17] E. Hoque, O. Chowdhury, S. Y. Chau, C. Nita-Rotaru, and N. Li, "Detecting specification noncompliance in network protocol implementations," in *USENIX Annual Technical Conference (ATC) as a Poster*. USENIX, 2016.

[18] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute force vulnerability discovery*. Addison-Wesley Professional, 2007.

[19] M. Welsh, D. E. Culler, and E. A. Brewer, "Seda: An architecture for well-conditioned, scalable internet services," in *Proceedings of the 18th Symposium on Operating System Principles (SOSP)*, 2001.

[20] C. Killian, J. Anderson, R. Braud, R. Jhala, and A. Vahdat, "Mace: Language support for building distributed systems," *ACM SIGPLAN Notices*, vol. 42, no. 6, 2007.

[21] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *IFIP/ACM Middleware*, 2001.

[22] N. Lynch, *Distributed Algorithms*. Morgan Kaufmann, 1996.

[23] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2005.

[24] Y. Wang, Z. Zhang, D. Yao, B. Qu, and L. Guo, "Inferring protocol state machine from network traces: A probabilistic approach," in *Proceedings of the 9th International Conference on Applied Cryptography and Network Security (ACNS)*. Springer-Verlag, 2011.

[25] I. Habib, "Virtualization with KVM," *Linux Journal*, 2008.

[26] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 1999.

[27] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage, "Steward: Scaling byzantine fault-tolerant replication to wide area networks," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 7, no. 1, 2010.

[28] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," in *Proceedings of the Symposium on Operating System Principles (SOSP)*, 2007.

[29] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Prime: Byzantine replication under attack," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 8, no. 4, 2011.

[30] A. Clement, E. Wong, L. Alvisi, and M. Dahlin, "Making byzantine fault tolerant systems tolerate byzantine faults," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[31] R. Gray, D. Kotz, C. Newport, N. Dubrovsky, A. Fiske, J. Liu, C. Masone, S. McGrath, and Y. Yuan, "Outdoor experimental comparison of four ad hoc routing algorithms," in *Proceedings of the 7th ACM International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM)*, 2004.

[32] C. Perkins and E. Royer, "Ad-hoc on-demand distance vector routing," in *Proceedings of the Second IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, 1997.

[33] K. Sanzgiri, B. Dahill, B. Levine, C. Shields, and E. Belding-Royer, "A secure routing protocol for ad hoc networks," in *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, 2002.

[34] P. Jacquet, P. Muhlethaler, T. Clausen, A. Laouiti, A. Qayyum, and L. Viennot, "Optimized link state routing protocol for ad hoc networks," in *Proceedings of the IEEE International Multi Topic Conference*, 2001.

[35] C. Perkins and P. Bhagwat, "Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers," *ACM SIGCOMM Computer Communication Review*, vol. 24, no. 4, 1994.

[36] A. Neumann, C. Aichele, M. Lindner, and S. Wunderlich, "Better approach to mobile ad-hoc networking (B.A.T.M.A.N.)," http://tools.ietf.org/html/draft-wunderlich-openmesh-manet-routing-00, accessed: 2016.

[37] E. Hoque, H. Lee, C. E. Killian, and C. Nita-Rotaru, "A testing platform for teaching secure distributed systems programming," Dept. of CS, Purdue University, Tech. Rep. 16-002, 2016.

[38] J. Kirsch and Y. Amir, "Paxos for system builders," Dept. of CS, Johns Hopkins University, Tech. Rep. CNDS-2008-2, 2008.

[39] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, 1982.

[40] K. P. Birman and T. A. Joseph, "Reliable communication in the presence of failures," *ACM Transactions on Computer Systems (TOCS)*, vol. 5, no. 1, 1987.

[41] D. Kaloper-Meršinjak, H. Mehnert, A. Madhavapeddy, and P. Sewell, "Not-Quite-So-Broken TLS: Lessons in re-engineering a security protocol specification and implementation," in *24th USENIX Security Symposium (USENIX Security)*. USENIX, 2015.

[42] P. Godefroid, "Model checking for programming languages using verisoft," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 1997.

[43] M. Musuvathi and D. Engler, "Model checking large network protocol implementations," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[44] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill, "CMC: Pragmatic approach to model checking real code," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, 2002.

[45] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE way to test openflow applications," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 2012.

[46] P. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2009.

[47] J. Caballero, P. Poosankam, C. Kreibich, and D. Song, "Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2009.

[48] C. Cho, D. Babić, E. Shin, and D. Song, "Inference and analysis of formal models of botnet command and control protocols," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2010.

[49] N. Kothari, T. Millstein, and R. Govindan, "Deriving state machines from tinyos programs using symbolic execution," in *Proceedings of the IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE, 2008.

[50] J. de Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *24th USENIX Security Symposium (USENIX Security 15)*. USENIX, 2015.

[51] J. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, 1976.

[52] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Păsăreanu, R. Bby, and H. Zheng, "Bandera: Extracting finite-state models from Java source code," in *Proceedings of the 2000 International Conference on Software Engineering (ICSE)*, 2000.

[53] M. Das, S. Lerner, and M. Seigle, "ESP: Path-sensitive program verification in polynomial time," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2002.

[54] R. Jhala and R. Majumdar, "Software model checking," *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, 2009.

[55] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2008.

[56] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV version 2: An open source tool for symbolic model checking," in *Proceedings of the International Conference on Computer-Aided Verification (CAV)*. Springer, 2002.

[57] A. Dunkels, "Full TCP/IP for 8-bit architectures," in *Proceedings of the ACM International conference on Mobile systems, applications and services (MobiSys)*. ACM, 2003.