ENSURING SPECIFICATION COMPLIANCE, ROBUSTNESS, AND SECURITY OF WIRELESS NETWORK PROTOCOLS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Md. Endadul Hoque

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2015

Purdue University

West Lafayette, Indiana

In dedication to *my mom*, *my wife*, and *my daughter*

ACKNOWLEDGMENTS

I would like to take this opportunity to express my deepest gratitude to a number of people without whom this dissertation would not have been possible.

To begin with, I want to express my biggest thanks to my advisor, Prof. Cristina Nita-Rotaru, for her in-depth guidance, insightful critiques, and never-ending encouragement to pushing the envelope during my PhD years at Purdue. She not only provided necessary support during the whole span of this research work, but also gave me thoughtful comments and suggestions, which were instrumental in my research. I am sure the skill-set that I learned from her would help me in my future career.

I also want to express my deepest thanks to Prof. Sonia Fahmy, Prof. Ninghui Li, and Prof. Dongyan Xu for serving on my advisory committee and for advising me with their insightful and constructive feedbacks on my thesis proposal which have helped me a great deal in completing this dissertation. A special thanks to Prof. Fahmy for helping me in a critical time and for agreeing to be a co-chair of my advisory committee.

I am thankful to Dr. Rahul Potharaju, an extremely dear friend whose consistent moral support and positivity kept me going through the grayest days of my life at Purdue. I thank him for being the power-house of inspirations during my PhD days and definitely for years to come. I am also thankful to Dr. Omar Haider Chowdhury, a dear friend who I always admire for his never-ending enthusiasm, continuous motivation, and critical guidance in my research. I thank him for believing in me and supporting me, when I needed it the most. I am greatly indebted to both of them for teaching me humility, grace, and above all, believing in myself.

I would like to thank Dr. Hyojeong Lee, a friendly lab mate and an excellent collaborator, for being a great source of motivation and for helping me in my research.

Another friend that requires special mention is Sze Yiu Chau for his great support and for all the intellectual debates we had during my final year at Purdue.

On a personal note, my immense gratitude goes to my parents for being supportive of my education all along the way. I am thankful to my in-laws for their support and encouragement throughout. A special thanks from the bottom of my heart goes to my wife, Dr. Farzana Rahman. This dissertation is largely due to her never-ending stream of support, encouragement, patience, and companionship through the difficult times of my PhD years. I am grateful to her for being by my side through my ups and downs, for spending countless hours to listen to my research problems, and for having undoubted belief in me even when I have doubts on my abilities. My daughter, Rinisha Rahman Hoque, requires a special mention because she has not only made us complete but also given me extra motivation—through her cute smiles—to go the extra miles to finish my PhD.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABBREVIATIONS

AODV	Ad hoc On-Demand Distance Vector
BATMAN	Better Approach To Mobile Adhoc Networking
BFS	Breadth-first Search
CEX	Counterexample
DFS	Depth-first Search
DHCP	Dynamic Host Configuration Protocol
DSDV	Destination-Sequenced Distance-Vector Routing
FSM	Finite State Machine
IoT	Internet-of-Things
LTL	Linear Temporal Logic
NFC	Near Field Communication
NVT	Network Virtual Terminal
OLSR	Optimized Link State Routing
PDR	Packet Delivery Ratio
pLTL	Propositional Linear Temporal Logic
RFC	Request for Comments
SE	Symbolic Execution
SMT	Satisfiability Modulo Theories
TCP	Transmission Control Protocol

ABSTRACT

Hoque, Md. Endadul PhD, Purdue University, December 2015. Ensuring Specification Compliance, Robustness, and Security of Wireless Network Protocols. Major Professor: Cristina Nita-Rotaru and Sonia Fahmy.

Several newly emerged wireless technologies (*e.g.*, Internet-of-Things, Bluetooth, NFC)—extensively backed by the tech industry—are being widely adopted and have resulted in a proliferation of diverse smart appliances and gadgets (*e.g.*, smart thermostat, wearables, smartphones), which has ensuingly shaped our modern digital life. These technologies include several communication protocols that usually have stringent requirements stated in their specifications. Failing to comply with such requirements can result in incorrect behaviors, interoperability issues, or even security vulnerabilities. Moreover, lack of robustness of the protocol implementation to malicious attacks—exploiting subtle vulnerabilities in the implementation—mounted by the compromised nodes in an adversarial environment can limit the practical utility of the implementation by impairing the performance of the protocol and can even have detrimental effects on the availability of the network. Even having a compliant and robust implementation alone may not suffice in many cases because these technologies often expose new attack surfaces as well as new propagation vectors, which can be exploited by unprecedented malware and can quickly lead to an epidemic.

Given the stake associated with these wireless technologies, the requirement to ensure secure and reliable operations calls for both pre- and post-deployment mechanisms. In this dissertation, we focus on fortifying these emerging technologies along three dimensions. First, we propose an automatic compliance checking technique allowing a developer to ensure—before deployment—that the implementation is compliant with the protocol specifications. Second, we develop an automated adversarial

testing platform to help developers find vulnerabilities in their protocol implementations prior to deployment, thereby ensuring robustness of the implementations in adversarial environments. Finally, we devise several countermeasures to mitigate infection in the event of attacks after deployment.

1 INTRODUCTION

In recent years, new wireless technologies have emerged and changed the way we live and interact with the environment through various devices ranging from tiny smart objects such as smart home appliances, implantable medical devices, to large computing devices such as automobiles. The devices interact using a variety of methods including WiFi [1], 6LoWPAN (IPv6 over IEEE 802.15.4 [2]), Bluetooth [3], RFID [4], near field communication (NFC) [5], Internet-of-Things (IoT) [6]. These networks have become the foundation of many applications and services that our daily life depends on. Therefore, secure and reliable operations of these emerging networks have strong impact on our socio-economic life.

Like any traditional networks, the core of these new wireless networks consists of several communication protocols, which the user applications and services are built on. Most of these protocols are standardized through explicit specifications, which are often carefully studied to uncover design flaws and errors. However, many errors and bugs can be introduced during the implementation phase, which often manifest after the deployment of the implementation. Errors leading to inconsistent output or incorrect behaviors cause the implementation fail to comply with its specifications and thus make it a *non-compliant* implementation. Therefore, checking only the design for compliance is not enough. Moreover, checking implementations for compliance is a painstakingly time-consuming task, which is aggravated due to the increased design complexity of the protocols and the limited functionalities of existing compliance checking tools. Therefore, it is imperative to develop automated techniques that can assist a developer to ensure whether the implementation is complaint with its specifications prior to deployment.

Despite having an implementation compliant with its specifications, the implementation may contain vulnerabilities that manifest only at the presence of compromised

nodes, which can behave arbitrarily and thus mount attacks. Lack of robustness to such attacks can limit the practical utility of the protocol. Such vulnerabilities often remain undetected using traditional testing approaches. While such testing approaches have been shown to be fruitful, they have some significant weaknesses. For instance, tedious manual testing can easily become exhausting with the increased complexity of the implementation and leave portion unexplored due to developers' inability to reason about such cases; similarly, static analysis is inevitably imprecise for vulnerabilities that manifest only during concrete execution in an adversarial environment. Therefore, it is necessary to ensure robustness of a protocol implementation in an adversarial environment before deployment by developing automated testing techniques to find vulnerabilities in the implementation.

Ensuring security has always been the "arms race" between malware creators and those seeking to thwart their activities. "Zero day" attacks after deployment are not unprecedented in case of well-known and widely used protocol implementations [7, 8], let alone for the protocols developed for the emerging networks [9, 10]. Furthermore, the emerging networks often introduce new attack surfaces as well as new malware propagation vectors, which can ensue an epidemic from any malware infection. As a precautionary measure, applying compliance checking and adversarial testing techniques—separately or in tandem—can help developers safeguard their protocol implementations from numerous errors and attacks. Nevertheless, for a holistic defense, it is important to investigate countermeasures to mitigate infection in the event of attacks after deployment.

1.1 Focus and Motivation

In this thesis, we focus on fortifying the emerging wireless networks along three dimensions. Firstly, we strive to develop automatic compliance checking techniques aiding developers to ensure that their protocol implementations are complaint with the respective specifications. Secondly, we aim to develop automatic testing tech-

niques to help developers find vulnerabilities in their protocol implementations and thus ensure robustness of the implementations in adversarial environments. Finally, we intend to devise countermeasures to mitigate infection in the networks as post-deployment measures.

Compliance checking of network protocol implementations. Finite state machines (*FSM*s) are often used to specify stateful network protocols (*e.g.*, Telnet, DHCP). Such FSMs essentially identify the protocols' internal states and also indicate under what conditions (*e.g.*, occurrence of an event) the protocols change their internal states. Such (stateful) network protocols are expected to comply with numerous properties specified in the protocol specification documents such as RFCs. For instance, a desired property specific to the Telnetd (Telnet server) implementation for the Contiki [11] operating system[1] is: "*when the server has an on-going connected session with a client, any further connection requests should be rejected by the server*". Failing to adhere to these properties can result in inconsistencies in the internal states, interoperability issues, incorrect behaviors, or even security vulnerabilities. In the above example, if the Telnetd accepts another client connection when there is already an ongoing client connection, the protocol can misbehave and affect confidentiality and integrity by sending one client's (partial) command output to another. This is a real non-compliance reported in the Contiki forum [12].

Checking protocol implementations for non-compliances is challenging as some of the non-compliant behaviors of the implemented protocol can *only* be triggered by a long and complex sequence of events. Such intricate non-compliant executions can remain undetected due to the developers' inability to reason about such cases. Hence, it is paramount to develop techniques and tools that can assist developers to detect protocol non-compliances with limited manual effort.

The formal verification community has extensively explored the problem of checking whether a program complies with some invariants [13–18]. Among the existing

[1]An operating system for Internet-of-Things devices

work, the work by Holzmann *et al.* [19] and Musuvathi *et al.* [18,20] are the most relevant. Holzmann *et al.* [19] relies heavily on the developer annotations to syntactically extract the FSM of an event-driven program which is then model checked with some desired temporal logic properties [21]. Musuvathi *et al.* [20] develop an explicit-state model checker for network protocols written in C, but the properties they can check are limited to only boolean formulas and focused primarily on low-level programming errors. Such techniques are not enough to detect logical programming errors (called functional properties) introduced while implementing the FSMs described in the specifications. Therefore, we focus on developing a protocol compliance checking framework that allows a developer to check whether the implementation complies with its desired functional properties derived from the RFCs, research papers, and code documentation.

Adversarial testing of network protocol implementations. While checking compliance of a network protocol implementation is beneficial to detecting violation of desired properties, this technique does not necessarily evaluate how robust the implementation is in an adversarial environment where compromised participant(s) of the protocol can behave arbitrarily and thereby mount attacks. Such attacks can be detrimental for protocols that run across several nodes of the network, *e.g.*, dynamic routing protocols.

Routing is crucial for wireless mesh networks—enabled by technologies like Wi-Fi, WiMAX—that have emerged as a solution for metropolitan area networks (MAN) to provide the last-few-miles connectivity. As traditional routing protocols do not perform well in a resource-constrained environment like wireless networks in general, a significant volume of work has been put into designing routing protocols for wireless networks [22–26]. Given the importance of routing as a fundamental component of wireless networks, many protocols have been subjected to model checking the design [27] and to testing the simulator-based implementations [28, 29]. While model checking helps to verify the validity of the design, it cannot conclude that the actual

implementation is free of bugs and vulnerabilities since implementations contain optimizations not captured by the model. Some optimizations even diverge from the design and thus introduce new bugs. In addition, while simulators provide easier and simpler ways to evaluate a protocol, they sacrifice some aspects of realism such as the interaction of the protocol with the components of the operating system.

Recent research [30–32] showed the importance of performing adversarial testing (*i.e.*, testing systems implementations beyond just basic functionality such as examining edge cases, boundary conditions, and ultimately conducting destructive testing) for message-passing distributed systems. Adversarial testing makes protocols more robust to arbitrary and extreme conditions and can discover vulnerabilities in implementations, many of which might have not occurred in simulator-based implementations. Previous work related to wireless routing implementations has focused exclusively on performance comparison across protocols [33–35] or on evaluating performance of TCP in multihop ad hoc networks [33, 36]. Therefore, it is important to ensure the robustness of a protocol implementation in an adversarial environment by finding bugs and vulnerabilities that can limit the practical utility of the implementation. In this thesis, we focus on automated adversarial testing of actual implementations of wireless routing protocols.

Infection mitigation. While proactive measures like compliance checking and adversarial testing augment the inventory of pre-deployment prevention mechanisms, reactive measures are required to address the aftermath of "zero-day" attacks, which are not unheard of in communication networks, especially, the Internet. In fact, with the advent of smartphones and Internet-of-Things, the number of wireless devices with complex capabilities has significantly increased. While the openness of such wireless devices—supported by operating systems like Google's Android [37], Contiki [11], FreeRTOS [38]—induces developers' motivation, it also introduces new propagation vectors for mobile malware. Recent reports show a significant increase of malware targeting Android devices [39–41] and IoT devices [42–44]. The most prominent mal-

ware propagation vectors include installation of malicious applications (apps) from third party app stores, as well as SMSs and emails with URL links tricking users to download malicious applications. However, the spread of malware through proximity-based communication has not left un-attempted. Recent incidents [9, 45–47] provide evidences of malware propagation using short-range communication such as WiFi, Bluetooth or NFC.

Significant research focused on modeling infection propagation, detection, and application profiling of malware in the context of wired networks [48–52]. Those results do not model mobile malware that spreads directly from device to device by using short-range communication. Therefore, we focus on investigating the propagation model of mobile malware amongst humans carrying smartphones and design countermeasures to mitigate the propagation of mobile malware under a practical scenario.

1.2 Thesis Contributions

In this thesis, we contribute towards providing specification compliance, robustness, and security of emerging wireless networks through (a) checking compliance of protocol implementations with their specifications, (b) performing adversarial testing on the implementations to find vulnerabilities prior to deployment, and (c) mitigating infection in case of epidemic outbreak in the network after deployment. We summarize our key contributions as follows:

- We present a framework, CHIRON, that can check a network protocol source for compliance with standards and requirements collected from RFCs, academic papers, and documentation. We develop a technique that automatically extracts the FSM from the source code of a stateful, event-driven protocol with minimal developer assistance. A two-step validation process to rule out false non-compliance protocol executions is also developed. We show the applicability of CHIRON by testing 5 real protocol implementations from two different network stacks—uIP of Contiki [53], FNET [54]—against 18 protocol requirements

and uncover 10 instances of non-compliances, several of which have security implications.

- We develop Turret-W, a platform for automated adversarial testing of wireless routing protocols. Turret-W can test not only general attacks against routing, but also wireless specific attacks such as blackhole and wormhole attacks. Demonstrating Turret-W on publicly available implementations of five representative routing protocols, we (re-)discovered 37 attacks and 3 bugs. To the best of our knowledge, all these bugs and 5 of the total attacks were not previously reported.

- We model the propagation of mobile malware amongst humans carrying smartphones using epidemiology theory and study the problem as a function of the underlying mobility models. We define the optimal approach to heal an infected system with the help of a set of static healers (nodes that distribute patches) as the T-Cover problem and show that it is NP-Complete. We then propose three families of healer protocols that allow for a trade-off between the recovery time and the energy consumed for deploying patches.

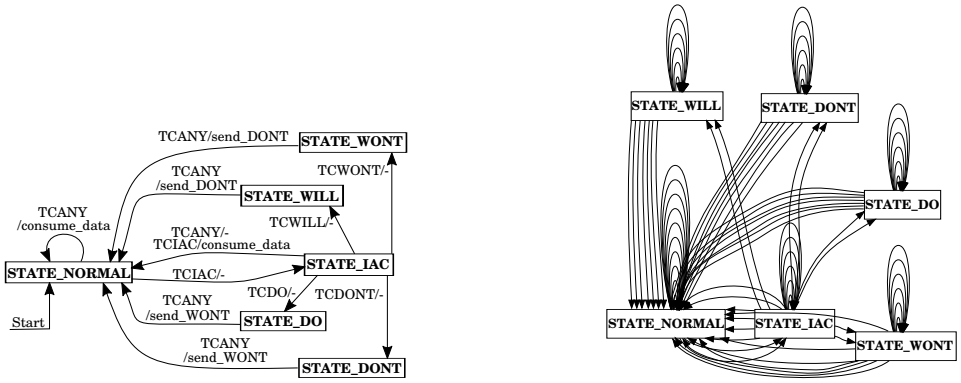
1.3 Thesis Organization

The rest of the thesis is organized as follows. We present our compliance checking approach in Chapter 2. Our adversarial testing tool is described in Chapter 3. We next describe how we mitigate infection propagation in the network in Chapter 4. We present the related work in Chapter 5 and conclude the thesis in Chapter 6.

2 COMPLIANCE CHECKING OF NETWORK PROTOCOL IMPLEMENTATIONS

Stateful network protocols are often specified using finite state machines (FSMs), which identify the protocols' internal states and indicate under what conditions (*e.g.*, occurrence of an event) the protocols change their internal states. We call such a specification FSM of a protocol an **S-FSM**. Fig. 2.1(a) shows the **S-FSM** of the Telnet server protocol (*i.e.*, Telnetd). Such (stateful) network protocols are often required to comply with some requirements (also known as *properties*) according to their specifications (*e.g.*, RFCs). An example requirement specific to the Telnetd implementation for the Contiki [11] operating system is: "*when the server has an on-going connected session with a client, any further connection requests should be rejected by the server*". Failing to adhere to the desired properties can result in inconsistencies in the internal states, interoperabilities, incorrect behaviors, or even security vulnerabilities. In the above example, if the Telnetd accepts another client connection when there is already an ongoing client connection, the protocol can misbehave by sending one client's (partial) command output to another client, thereby affecting confidentiality and integrity. This is a real non-compliance which we have detected using our approach and is also confirmed by a bug report filed by a developer in the Contiki forum [12]. The code snippet shown in Fig. 2.2 illustrates the *bug* in the implemented Telnetd for Contiki. At line 8 of the code snippet, the protocol moves to the normal state whenever its gets a new connection without additionally checking whether there is already an established connection, and this causes the non-compliance.

Checking protocol implementations for non-compliances with their specifications is challenging as it often requires an intricate sequence of network events to manifest such non-compliances using traditional testing approaches. Some previous works [13–18] explored the problem of checking whether a program complies with the given



(a) **S-FSM** of the Telnetd protocol

(b) **E-FSM** of the Telnetd protocol implemented for Contiki-2.4

Figure 2.1.: FSMs of the Telnetd protocol. (a) We follow the *event/action* convention when labeling transitions. The character '-' denotes an empty action. The prefix TC indicates a received Telnet command. TCANY signifies any character other than (DO, DONT, WILL, WONT). *consume_data* means the application consumes what has been received as normal data. (b) For ease of exposition, we do not show the transition labels of the **E-FSM**.

```
1        ... /* Omitted */
2  /* Protocol event dispatcher function */
3  void telnetd_appcall(void *ts)
4  {
5    if(uip_connected() /* Got a new connection? */ ) {
6      /* An FSM Bug can manifest here */
7      s.bufptr = 0;
8      s.state = STATE_NORMAL;
9      ... /* initialize buffer pointers */
10     ... /* start shell */
11   }
12   ...
13 }
```

Figure 2.2.: Code snippet from Telnetd implementation for Contiki-2.4

invariants. However, their reliance on syntactic approaches, their restricted form of properties, or their intention of focusing on low-level programming errors make them limited to be applied to detect logical programming errors (called functional properties) introduced while implementing the **S-FSM**s described in specifications.

In this work, we present CHIRON[1], a protocol compliance checking framework that allows a user to check whether a stateful, event-driven network protocol imple-

[1]In Greek mythology, Chiron was considered to be the wisest centaur amongst his brethren.

mentation in C complies with its desired properties derived manually from the RFCs, research papers, and documentation. CHIRON, in spirit, follows the high level approach of *counterexample-guided abstraction refinement* (CEGAR) [55]. CHIRON reasons about a protocol's compliance without making any restrictive assumptions about the underlying network stack or the behavior of the other protocol participants. The heart of CHIRON is an *FSM Extractor* that takes as input a couple of configuration files provided by the developer and the C source code of the implemented protocol. By leveraging symbolic execution [56], the FSM Extractor automatically extracts an approximated FSM of the implemented protocol. We refer to the extracted FSM from the protocol source as the **E-FSM**.

Automatically extracting an **E-FSM** that is suitable for compliance checking is challenging. Due to the many possible states and transitions, manually deriving the **E-FSM** from the source is an error-prone and time-consuming process. Fig. 2.1(b) shows the **E-FSM** (6 states and 84 transitions) automatically extracted by CHIRON from the Telnetd source for Contiki. Existing work has looked at inferring protocol FSMs— based on network traces [57–59], using program analysis [60, 61], or through model checking [14, 62]. While network trace-based approaches often suffer from incompleteness due to inadequate number of traces, others focus on extracting a sequence of messages valid in a session or the low-level program FSM instead of the protocol **E-FSM**. For compliance checking, however, it is required to have an **E-FSM** that precisely captures the relevant details.

Once we have the **E-FSM**, we use a symbolic model checker to check whether the **E-FSM** complies with the requirements given by the user. We consider requirements that are written as propositional linear temporal logic (pLTL) formulas [21]. If the **E-FSM** does not comply with a requirement, then the model checker outputs a counterexample (*i.e.*, an execution of the protocol) as evidence. Due to abstractions in our analysis, the provided counterexample (in short, CEX) may not be realizable in an actual execution of the protocol. Therefore, we use a two-step validation process to rule out unrealizable CEXs.

Since CHIRON does not make any assumption about the other protocol participants while extracting the **E-FSM**, one advantage of such an approach is that CHIRON can aid in composable reasoning. Specifically it can enable us to reason about the global properties of a two-party protocol by composing their individual **E-FSM**s. This is also very relevant in reasoning about protocols designed for distributed systems. In addition, due to the modular nature of our approach, different verification tools and techniques can be easily incorporated in CHIRON as pluggable components.

Our technique of extracting the **E-FSM** is of independent interest. Some existing work employs fuzz testing for finding vulnerabilities in protocol implementations. To overcome the inherent coverage problem of fuzz testing, several works like SNOOZE [63], KiF [64], and SNAKE [65] rely on user provided **S-FSM**s. Additionally, the extracted **E-FSM** can be visually checked against the **S-FSM** to spot missing or spurious transitions without requiring any verification.

CHIRON can also be used as a debugging tool for developers to find missing/unwanted state transitions in the **E-FSM**. Moreover, the **E-FSM** can be used to perform *counterexample driven model-based testing* [66,67] of a protocol implementation. Finally, our general analysis technique can easily be adopted in other contexts where the implementation is also written in an event-driven fashion, *e.g.*, Android UI testing.

We have implemented CHIRON and applied it to a total of 5 implementations of two different protocols– Telnet server protocol (*Telnetd*) and DHCP client protocol (*DHCPc*) – from two separate TCP/IP network stacks: uIP [53] (part of Contiki) and FNET [54]. Contiki is a widely used open source operating system that runs on Internet-of-Things (IoT) devices, *e.g.*, smart home appliances [68]. FNET is a network stack actively maintained by Freescale Semiconductor Inc., which supports various microcontroller units (MCUs) used in a wide range of applications including IoT devices, health-care, and vehicular control systems [69]. We use 11 representative properties for Telnetd and 7 for DHCPc derived from their RFCs, documentation, and/or bug reports. During compliance checking of these 5 implementations, we

discovered 10 non-compliance instances in total. One of these non-compliances has security implications while others can hinder interoperability and possibly impair performance. Although our technique is general enough to be applicable to other network protocols, in our evaluation, we particularly focus on protocol implementations for Contiki and FNET as they are widely used by IoT devices but have not been extensively studied. To summarize, this work makes the following contributions:

- We present a framework, CHIRON, that can check a network protocol source for compliance with standards and requirements collected from RFCs, academic papers, and documentation.
- We develop a technique that automatically extracts the **E-FSM** from the source code of a stateful, event-driven protocol with minimal developer assistance.
- We develop a two-step validation process to rule out false non-compliance protocol executions.
- We also present optimizations that make the **E-FSM** extraction and compliance checking efficient.
- We show the applicability of CHIRON by testing 5 real protocol implementations from two different network stacks— uIP of Contiki, FNET—against 18 protocol requirements. We demonstrate the efficacy of CHIRON experimentally, and in the process, we uncover 10 non-compliances, several of which have security implications.

2.1 Background

In this section, we briefly review the background materials necessary to understand our technical contributions.

Finite state machine (FSM). A finite state machine in our setting, denoted by M, is a tuple $\langle Q, Ev, A, \mathcal{V}_c, q_I, R \rangle$. Q represents a finite set of states $q_0, \ldots, q_n$ and $q_I \in Q$ represents the initial state of the finite state machine M. We use Ev to

denote a set of events (*e.g.*, receive) whereas we use A to denote a set of actions (*e.g.*, send_ack) the protocol can perform. We assume $Ev \cap A = \emptyset$. We also have a finite set of *conditional* variables $\mathcal{V}_c$ which is disjoint from both the sets Ev and A. The set R represents the transition relation and $R \subseteq Q \times Ev \times \mathcal{C} \times 2^A \times Q$ where the condition $\mathcal{C}$ represents a set of *transition conditions* and each element of which is a quantifier-free first-order logic formula over variables $\mathcal{V}_c$. If $\mathcal{V}_c = \{x, y\}$, then an example transition condition can be $x \geq 0 \wedge x + y \neq 10$ where each atomic formula of the transition condition (*e.g.*, $x \geq 0$) is called an *atom*. Given a transition $\langle q_a, \mathsf{receive}, \mathsf{uip\_len}[0, (2\mathsf{B})] \neq 0 \wedge \mathsf{payload}[0, (1\mathsf{B})] = 255, \{\mathsf{send\_ack}\}, q_b\rangle$, it signifies that if the FSM is currently at state q_a, the event receive is triggered, the receive buffer is not empty (*i.e.*, receive buffer length is not equal to zero), and payload's first byte is 255, then the FSM can move to a new state q_b and can take the action send_ack. We use FSMs to abstractly represent the high level operation of a protocol.

Propositional Linear temporal logic (pLTL). Propositional linear temporal logic (in short, pLTL) extends propositional logic with temporal operators [21]. We use pLTL to express the desired properties a protocol should have. There are two kinds of temporal operators: *past temporal operators* and *future temporal operators*. pLTL reasons about relative temporal ordering of states/events without considering the explicit time at which each event/state happens. The past temporal operators are: $\diamondminus$ (read "*once*", it means that the formula following the operator was true at some point of time in the past including the current time point), $\boxminus$ (read "*historically*", it means the formula following the operator has been true all along in the past including the current time point), $\ominus$ (read "*yesterday*" which signifies that the following formula was true in the immediate previous step), and $\mathcal{S}$ (read "*since*", is a binary operator and $\varphi_1 \mathcal{S} \varphi_2$ is true in the current state if φ_2 was once true in the past, possibly in the current state, and φ_1 has been true since then till the current state.). The future operators $\diamondsuit$ (read "*eventually*"), $\square$ (read "*henceforth*"), $\bigcirc$ (read "*next*"), and $\mathcal{U}$ (read "*until*") are duals of the past operators. The atomic elements of a

pLTL formula is an atomic proposition which is a drawn from a finite set $\mathcal{P}$. which in our case is $\mathcal{P} = Q \cup A \cup E \cup T$ where T is a set and each element of T is a proposition denoting the truth value of an atom. A pLTL formula φ can be inductively defined as: $\varphi ::= \mathsf{true} \mid p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \diamondminus\varphi \mid \boxminus\varphi \mid \ominus\varphi \mid \varphi_1 \mathcal{S} \varphi_2 \mid \Diamond\varphi \mid \Box\varphi \mid \bigcirc\varphi \mid \varphi_1 \mathcal{U} \varphi_2$ where $p \in \mathcal{P}$. pLTL formulas are interpreted over *infinite traces* generated by a kripke structure or in our case a finite state machine of form M, each element of which is a state that maps each proposition in $\mathcal{P}$ to either true or false. For example, the formula $\Box(\mathsf{receive} \wedge \mathsf{receivedPacketIsACK} \rightarrow \diamondminus \mathsf{send\_data})$ specifies that whenever we receive an acknowledgement of a packet receive it implies that we have send a packet at some point of time in the past. The formal semantics of pLTL is standard and can be found elsewhere (cf. [21]).

Event-driven protocol implementation. A protocol can be implemented in plethora of ways. As a first cut in our context, we consider protocols implemented only in the event-driven paradigm. In the event-driven style of protocol implementation, the protocol has internal states which is altered with respect to different network events. For each event, the protocol has specific handling code that performs the necessary protocol state transition when that event occurs. The logic behind the transition of states with respect to a given event is protocol dependent. Let us consider the very simple protocol skeleton in Figure 2.3. In this protocol, there are three possible events: CONN (referring to a connection attempt), RECV (referring to receiving of new data), and CLOSE (referring to the termination of the connection). The main function waits for an event to occur and calls the `dispatch_event` function which based on the different types of events, calls its appropriate handling code that performs necessary state transitions.

Symbolic execution (SE). Symbolic execution (*SE*) is a program analysis technique which is used to generate program inputs such that it is possible to test different execution paths to attain a high level of coverage during testing. SE as the name suggests considers program input variables to have symbolic values and then it

```
1  // List of events
2  enum{CONN = 0, RECV, CLOSE, ...};
3  void dispatch_event(int ev){
4    switch(ev){
5      case CONN:
6        /*connection handling code*/
7        ...
8        break;
9      case RECV:
10       /*data receiving code*/
11       ...
12       break;
13     case CLOSE:
14       /*connection termination code*/
15       ...
16       break;
17     case ...:
18       ...
19     default:
20       /* Unknown event */
21       ...
22   }
23 }
24 int main(){
25   init();
26   while(1){
27         event = getEvent();
28         dispatch_event(event);
29   }return 0;
30 }
```

Figure 2.3.: A simple protocol implemented in event-driven paradigm

symbolically executes the program with those value. Special care is given in handling conditional branches (*i.e.*, if-else, loops). While symbolically executing the program whenever a conditional branch is encountered, SE first checks see to whether both the condition and its negation are satisfiable, if this the case, SE explores both paths in the program due to the branch but adds the branch condition (resp., its negation) as the constraint of that path. This is called the *path constraint* or *path condition*. When new branches are encountered, they are added to the current path constraint with conjunction. When a desired program location or the end of the program is reached, SE consults a SMT solver to solve the path constraint and obtain concrete values for the input variables for which the path is taken.

Consider the simple function `foo` presented in Figure 2.4. Let us assume x and y are the function's input variables. Let us also assume that they have the symbolic value α_x and α_y. After executing line 5, variables x, y, and z have values $\alpha_x + \alpha_y + 1$,

α_y, and 0, respectively. In line 6, the condition $x \geq 5$ is encountered, in which case SE consults the SMT solver to check whether the constraints $\alpha_x + \alpha_y + 1 \geq 5$ and $\alpha_x + \alpha_y + 1 < 5$ are both satisfiable. If this is the case, SE adds the constraint $\alpha_x + \alpha_y + 1 \geq 5$ to the path condition of the execution where the `if` branch is taken and conversely adds $\alpha_x + \alpha_y + 1 < 5$ to the path condition of the execution which takes the `else` branch. When the execution terminates, SE again consults the SMT solver to obtain a concrete value for each symbolic variable. For the execution that took the `if` branch, SE consults the SMT solver to obtain concrete values for α_x and α_y such that the path condition $\alpha_x + \alpha_y + 1 \geq 5$ is true. The output of the SMT solver (*i.e.*, the concrete values) can be used as the program input, which will drive the execution to take the `if` branch. This process is carried out by SE for each of the execution paths of the program.

```
1  void foo(){
2    int x = input();
3    int y = input();
4    int z = 0;
5    x = x + y + 1;
6    if(x >= 5 ){
7      z = 1;
8    }
9    else{
10     z = -1;
11   }
12 }
```

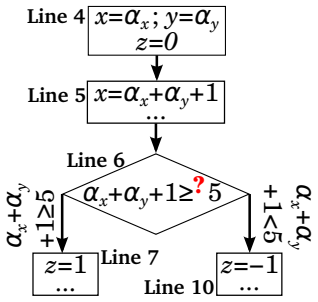


Figure 2.4.: Example of symbolic execution

2.2 Chiron Design

In this section, we first present the problem definition of protocol compliance checking. Next we present the assumptions and scope of CHIRON. Finally, we give an overview of CHIRON's design and then provide the detailed description of each of its components.

2.2.1 Problem Definition and High Level Approach

The problem of checking compliance of protocol implementation is formally defined as follows. *Given an event-driven protocol implementation* I_p *written in C and some desired property* PROP *that the protocol must comply to, is it the case that* I_p *satisfies the property* PROP? The property in question can express some desired functionalities of the protocol or some guarantees the protocol should provide. We want to emphasize that not all properties can be checked by our approach, and it mainly relies on the granularity of the program analysis. This will be made clear in later discussion.

At a high level, our approach has the following four steps:

- We first extract an FSM M, which abstractly captures the high-level operations of the protocol from I_p using a static program analysis technique. The extracted FSM has similarity with Input-Output automata [70]. The network events can be viewed as the FSM's inputs whereas the actions performed by the protocol can be viewed as FSM's outputs. In our extracted FSM, each transition additionally has a condition over some program variables (*i.e.*, *conditional variables*).
- We then manually extract a desired property PROP from the protocol documents (*e.g.*, RFCs, documentation) and express PROP as a pLTL formula φ.
- We then use a symbolic model checker to check whether M satisfies the formula φ, *i.e.*, $M \models^{?} \varphi$.
- If a counterexample (CEX) is generated due to the violation of φ by M, *i.e.*, $M \not\models \varphi$, we further scrutinize the CEX to ensure that the CEX is realizable in an actual execution of the protocol.

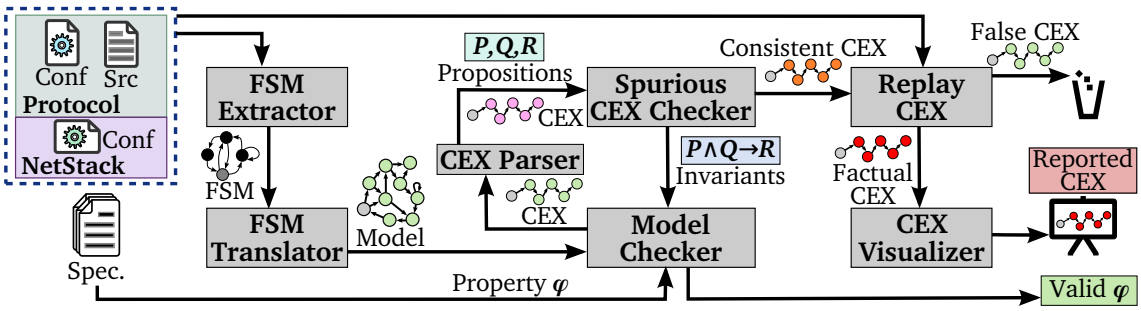


Figure 2.5.: The architecture of CHIRON

2.2.2 Assumptions and Scope

We assume that the source code of the protocol is available for our analysis. Extracting **E-FSM** from the implementation presents numerous challenges, the major ones being the state explosion and the precision of the FSM extraction technique. We focus specifically on network protocols written in C using an *event-driven* paradigm. We make the assumption that the implementation has an explicit representation of the protocol state machine, *i.e.*, protocol states are explicitly realizable through program variables. In addition, we assume that the possible values of the state variables are drawn from a small, finite domain. We also assume that all the event-handling code has a common entry point (often referred to as *event dispatcher* function) in the source code.

Non-compliances of the given properties can occur in an implementation due to *logical programming errors* such as wrong/missing state transitions and/or protocol actions when responding to an event; ultimately, not following the high level protocol design. We can only check properties which impose constraints over the events, state variables, and conditional variables. Such properties can be manually extracted from specifications such as RFCs, documents or bug reports.

We do not focus in detecting low-level memory errors (*e.g.*, null dereferencing, memory corruption, segmentation faults) in the implementation. There are complementary tools and techniques for detecting such errors [71–74].

2.2.3 Design Overview

The architecture of CHIRON is presented in Fig. 2.5. It consists of the following main components: *FSM extractor*, *FSM translator*, *model checker*, CEX *parser*, *spurious* CEX *checker*, CEX *replayer*, and CEX *visualizer*.

The FSM extractor is the key component of CHIRON. Its main goal is to extract the abstract FSM representation of the protocol using a static program analysis based on symbolic execution [56]. To achieve this goal it takes as input the source code of the protocol that is to be checked for compliance and two types of configurations: (a) Configuration specific to the network stack, which the protocol relies on, and (b) Configuration specific to the protocol implementation. The network stack configuration contains information about the possible network events. Whereas the protocol specific configuration consists of the program variables that comprise the FSM state of the protocol, the event dispatcher function, the list of actions performed by the protocol when responding to network events, and the set of conditional variables. These conditional variables will be included in the FSM transitions, and they can be referred to in the property being checked. We describe the FSM extraction algorithm in Section 2.2.4 in details.

For compliance checking, CHIRON uses a symbolic model checker that takes as input an FSM M represented in a high level modeling language and a pLTL formula φ, which is a desired property the protocol must comply with and checks to see whether the property φ is satisfied by M. In case φ is violated, the model checker generates a counterexample (CEX) as evidence. The property that can be checked depends on the granularity of the analysis, *i.e.*, the variables that were included in the FSM extraction. More precisely, if some information is not captured by the FSM M due to a coarse-grained analysis, it cannot be used in the property.

To bridge the gap between the output format of the FSM extractor and the format required by the different model checkers, we introduce an FSM translator. The FSM translator takes as input the **E-FSM** represented in our XML-based intermediate

language and translates it to the high level modeling language expected by the model checker. Note that the conditions associated with the transitions of the **E-FSM** are quantifier-free first order logic (QF-FOL) formulas over conditional program variables, *e.g.*, $x > 0 \wedge (x + y = 5) \wedge y < 0$. However, model checkers generally do not support QF-FOL formulas and expects the transition conditions to be boolean variables, *i.e.*, *propositions*. The FSM translator maps each unique atom (*e.g.*, $x + y = 5$) of the transition conditions to a unique propositional variable.

Due to abstraction during FSM extraction and also due to abstracting atoms with propositions, the counterexample (CEX) generated by the model checker may just be a false CEX (*i.e.*, *not factual* CEX). More precisely, the CEX generated by the model checker may not be realizable during an actual execution of the protocol. Hence, to rule out false CEXs, we use a two-step CEX verification process.

A CEX generated by the model checker contains the boolean (*i.e.*, true/false) assignment to each proposition. As the model checker is oblivious to the semantics of the atom (*e.g.*, $x > 0$) associated with each proposition, it will check all possible boolean values of the propositions during verification. It could happen that one such boolean assignment to the propositions in a CEX is not satisfiable considering their corresponding concrete atoms. To rule out such cases, we use a *Spurious* CEX *checker* that consults an SMT solver to check whether each transition condition in the CEX is satisfiable according to the truth assignment given by the model checker. If all transition conditions of a CEX are satisfiable, we refer to the CEX as a *consistent* CEX. However, in case of an inconsistent CEX, *i.e.*, at least one transition condition is not satisfiable, the Spurious CEX checker automatically generates invariants that notify the model checker to rule out the unsatisfiable transition in the future verification steps.

Due to the abstractions used during the extraction of the **E-FSM**, a consistent CEX still may not be realizable in an actual execution of the protocol, especially, while checking liveness properties [75]. Hence, to rule out such false CEXs, we use a CEX *replayer* that generates concrete program inputs from the consistent CEX with the

help of the SMT solver, and concretely executes the protocol implementation using those inputs. It also monitors the execution to check whether the state transitions and the associated actions in the implementation and in the CEX agree. If a consistent CEX is realizable during the replay, we call it a *factual* CEX.

Two auxiliary modules are used by CHIRON in the process of compliance checking. The first is a parser (CEX *parser*) that converts the CEXs generated by the model checker to the input of Spurious CEX Checker. Specifically, it takes as input a proposition-atom map file generated by the FSM translator and the CEX generated by the model checker, then replaces the propositions with their respective atoms in the CEX. For instance, if the map file contains the proposition-atom mapping $P \mapsto \{y < 5\}$ and the CEX maps P to false, then the CEX parser replaces $P \mapsto$ false with $(y < 5) \mapsto$ false. The second is a visualizer (CEX visualizer) that graphically presents the factual CEX to the user so that the CEX can be easily understood.

2.2.4 FSM Extraction Algorithm

The FSM extraction algorithm takes protocol source code as input and two configuration files– one related to the protocol implementation and the other related to the network stack – and extracts the FSM of the protocol. The protocol configuration file contains the following information: a list of program variables that form the protocol FSM state, an entry point to the event handling code (*i.e.*, event dispatcher function), the granularity of the analysis by selecting program variables (*i.e.*, conditional variables) that should be marked as symbolic (*e.g.*, the packet header or the packet buffer), and the list of actions along with their signatures (*e.g.*, send_data function with argument 255 means the action sending_command_byte). The network stack configuration contains a list of events and how to trigger these events in the code (*e.g.*, possibly by setting a bit or by assigning a particular value to a variable).

At a high level, our algorithm has the structure of a graph search algorithm in which we choose one of the possible events that can happen in the current state and

symbolically execute the protocol code from the entry point of the event handling function until it encounters new states or transitions. In that sense, it can also be viewed as a fixed point iteration algorithm. Specifically, the algorithm follows the structure of the breadth-first search algorithm (BFS). However, other graph search algorithms (*e.g.*, depth-first search, iterative deepening search, A*) can be easily adopted for our purpose. Although BFS is less memory efficient than DFS, we use BFS as it can be easily parallelizable and it can find states in shorter time. We keep track of the values of the protocol state variables and also the path conditions that are encountered during symbolic analysis. Whenever the value of one of the protocol state variables are changed, we check to see whether we have seen that state. If this is the first time the state has occurred then we mark it as a new state, which will be used for analysis later. We also check to see whether we have seen the transition between the current state and new state. If the transition is new then we add it to the FSM. We use the path conditions gathered during the symbolic execution as the condition of the transition. We continue the analysis until we do not see any new state or transition.

A pseudocode of our algorithm is presented below (c.f., Algorithm 1). The algorithm starts off by constructing the initial execution state e_0 of the program. The initial state of the program contains initialized values for both protocol state variables and other non-symbolic global variables. It then extracts the initial protocol state q_0 from e_0 using the function ExtractFsmState. This function basically takes a projection of the protocol state variables and their values from a given program state. We then mark q_0 as seen and add it to the FSM M. We then add the execution state e_0 to a working queue W_e. Next we process one execution state e_i at a time in FIFO manner until W_e is empty. We first extract the associated FSM state q_i from e_i. Then we try all possible events τ that are feasible in that state. For each such event τ, we run symbolic analysis from the protocol event handling entry point by simulating the occurrence of τ. The symbolic execution returns all possible paths and their associated execution states, path conditions, and associated actions taken by the protocol

(*e.g.*, sending an acknowledgement, retransmitting a packet). For each of the possible paths, we take the execution state e_j, the path condition on symbolic variables c_j, and the actions a performed by the protocol. We then extract the FSM state q_j from e_j. We insert q_j to the FSM M if we have not seen q_j and mark it as seen. In that case, we also add e_j to W_e. We then check whether we have seen the transition $q_i \xrightarrow{\tau,c,a} q_j$. If not, we add it to M and mark it as seen.

Algorithm 1: FSM Extraction Algorithm

Input: The protocol source S, analysis configuration C, network-stack configuration N
Output: An abstract FSM M of the protocol

```
1  Queue W_e ← ∅; FSM M ← ∅;
2  Create initial program state e_0;
   // q_0 is the initial protocol FSM state
3  q_0 ← ExtractFsmState(e_0);
4  W_e.enqueue(e_0);
5  M.Q ← {q_0};
   // M.Q is the set of states
6  M.q_I ← q_0;
   // M.q_I is the initial state
7  M.R ← ∅;
   // M.R is the set of transitions
8  Mark state q_0 as old;
9  while W_e ≠ ∅ do
10     e_i ← W_e.dequeue();
11     q_i ← ExtractFsmState(e_i);
12     foreach Event τ ∈ PossibleEvents do
13        S_q ← SymbolicExecution(S, C, N, e_i, τ);
14        foreach ⟨e_j, c, a⟩ ∈ S_q do
15           q_j ← ExtractFsmState(e_j);
16           if q_j is not old then
17              W_e.enqueue(e_j);
18              M.Q ← M.Q ∪ {q_j};
19              Mark state q_j as old ;
20           if Transition ⟨q_i, τ, c, a, q_j⟩ is not old then
21              M.R ← M.R ∪ {⟨q_i, τ, c, a, q_j⟩};
22              Mark transition ⟨q_i, τ, c, a, q_j⟩ as old ;
```

2.2.5 FSM Translation, Property Extraction and Verification

Once an abstract FSM is received from the FSM extractor, the FSM translator *automatically* carries out the following two functions: (i) It generates a mapping between each unique atom in the path conditions with a unique proposition; (ii) It then translates the abstract FSM description to the input modeling language of the model checker. As the model checker generally works with propositions whereas we have quantifier-free first-order logic formulas as the path conditions (*e.g.*, $(x > 0) \wedge (x + y = 5) \wedge y < 5$), the FSM translator replaces each unique atom (*e.g.*, $(x > 0)$) with a propositional variable. Function (ii) of FSM translator is dependent on the input modeling language of the model checker. This function needs to be adapted to handle different model checkers.

The properties against which to check the protocol implementation can come from exploring RFC documents, research papers proposing the protocol, properties from the developer, and possible bug reports. For specifying the properties in pLTL, the **E-FSM** and the atom-proposition mapping file is required.

Once the FSM is translated to the high level modeling language, the model checker takes that model and the property to check and exhaustively searches the state space to look for an execution of the model in which the property is violated. Once such an example is found, it is returned as a CEX. Each state in the CEX is a truth assignment to each of the different propositions used in the **E-FSM**.

2.2.6 Spurious CEX Checking

Since the model checker we use is oblivious towards the real semantic meaning of the propositions, we can have a CEX which is spurious. More precisely, the truth assignment given by the model checker in the CEX might not be satisfiable.

For example, let us assume we have the following three proposition-atom mappings from the FSM translator: $p \mapsto \{x > 0\}, q \mapsto \{x + y = 5\}, r \mapsto \{y < 5\}$. Let us also assume that there is a transition in the FSM whose condition is $p \wedge q \wedge r$

and the counterexample assigns the propositions p, q, and r the following values: false, true, true, respectively. Now, if $y < 5$ and $x + y = 5$, then it is apparent that $x > 0$ cannot be false. Hence, $\neg(x > 0) \wedge (x + y = 5) \wedge y < 5$ is not satisfiable. The purpose of the Spurious CEX checker is to rule out such cases by consulting a SMT solver. When the CEX fails the spurious check, *i.e.*, there is a transition condition which is not satisfiable according to the truth assignment given by the model checker, then the Spurious CEX checker forwards an appropriate invariant to the model checker to rule out this case and continue verification. For instance, in our example, the invariant will be $(q \wedge r \rightarrow p)$ (if q and r are true, then it implies that p is true).

2.2.7 Replaying CEX

Due to abstractions (*i.e.*, when we extract the **E-FSM** state from the program state, we ignore some of the program variables), fixed number of loop iterations during symbolic execution, and the granularity of the analysis process, we could have a CEX which is not realizable during the real execution of the protocol even though the CEX passes the spurious checking. This is specifically relevant while verifying liveness properties.

For instance, let us assume that the user selects a coarse-grained analysis in which the user does not take into consideration a specific timer. Now let us assume that the property we want to verify is that: "*Whenever the protocol is in state S_1 then it implies that the protocol will eventually move to state S_2*" (formally, $\Box((\text{state} = S_1) \rightarrow \Diamond(\text{state} = S_2))$). Let us assume that there is a self-loop in S_1 that is conditioned by a timeout of the timer, where the timer is not considered in the analysis. Hence, the condition of the self-loop in the **E-FSM** does not include the timeout. The model checker can easily find a counterexample in which the protocol always stays in state S_1 using the self-loop whereas in the real code, this will not happen as the timeout will happen and the program will move to possibly a new state.

To rule out cases like above, we replay the CEX and follow along the execution of the program. The replay execution is guided by the CEX which instructs the replay mechanism what branches to take via the truth assignment of the propositions. We monitor the different actions and state changes during the execution and whether it matches the CEX. If the protocol execution agrees with the CEX, then we report the CEX by visualizing the *factual* CEX to the user.

2.3 Implementation

In this section, we present the details related to the implementation of CHIRON. We also describe how we address the various implementation challenges and point out some adopted optimizations to speed up the compliance checking process of CHIRON.

2.3.1 Preparation for Analysis

Before the protocol source becomes amenable to our framework we require some steps to prepare the source. Recall that our framework only requires the source files that implement the protocol analyzes it in isolation. To make the source self-contained, we carry out the following steps in order. Among the following steps, only the code harnessing step requires manual efforts.

Preprocessing done by compiler Since we want to analyze only the protocol source code, not any underlying network stack code, we first preprocess the necessary protocol source files by using a C preprocessor. We basically use `gcc -E -P` to stop the compilation process right after preprocessing stage. After this step, we obtain a preprocessed source file containing all the declarations from the included header files and having all the macros expanded.

Slicing the source Like any other program analysis, our analysis could be adversely affected by the size of the source. From the user-provided configuration file,

we slice the protocol source using the state variables as the slicing criteria. This step generates a smaller compilable source from the original source that only contains the necessary program information required for the analysis. For slicing the source, we use `Frama-C` (Fluorine version) [76]. We would like to emphasize that this step is not mandatory but is useful to speed up the analysis.

Code harnessing In this step, we make the source completely self-contained by wrapping it in a test harness if necessary. To do so, we first add a `main()` function to the protocol source code, where we mark all the extern variables of the source as symbolic (*e.g.*, `uip_appdata`, the network stack payload buffer in Contiki) and then call the event dispatcher function (*e.g.*, `telnetd_appcall()`). An example of a harnessed `main()` function for Telnetd protocol is shown in Fig. 2.6. In § 2.3.2, we will explain the reason for calling the dispatcher function once in the `main()` function. Next, we add empty stub functions for the external functions, for instance, `uip_send()` function of the TCP/IP stack for Contiki, which the application calls to transmit a message over the network.

If necessary, we provide simple implementations of basic library functions such as `strlen()`, `memcpy()`. In case a library function returns a value, for example, `is_timer_expired()`, we provide a small stub implementation that returns a symbolic value. Hence, our analysis can capture all the additional transitions created due to the branching conditions on the returned symbolic value. In addition, we unroll any loop code blocks for a fixed number of iterations. If required, we can even unroll a loop block for a variable number of iterations by marking the loop counter as symbolic, which results in a large number of extra, but futile, transitions due to the symbolic loop counter.

Generating LLVM bitcode In the very last step of preparation, we use the `llvm-gcc` (version 4.2 for LLVM 2.9) compiler front end to generate LLVM bitcode of the protocol source. The output is used by the FSM extractor to carry out the analysis for FSM generation.

```
1  int main(){
2    void *ts;
3    /* Network stack payload buffer */
4    mark_symbolic(uip_appdata);
5    /* Network stack payload length */
6    mark_symbolic(uip_len);
7    /* Network connection struct */
8    mark_symblic(uip_conn);
9    /* An opaque pointer used by the process */
10   mark_symblic(ts);
11   /* Event dispatcher function */
12   telnetd_appcall(ts);
13   return 0;
14 }
```

Figure 2.6.: A sample of a harnessed main function for the same Telnetd protocol source shown in Fig. 2.2

2.3.2 Symbolic Execution for Deriving FSM

Once we have the protocol source (to be precise, LLVM bitcode) and the configuration files, we extract the **E-FSM** out of the source using our FSM extractor. We then use the KLEE symbolic execution engine [77] to symbolically execute the LLVM bitcode (LLVM-2.9 version) of the given program. We have implemented and integrated our FSM extractor in KLEE as a library containing more than 4 KLOC of C++ code to the original KLEE code base.

The FSM extractor starts by loading the LLVM bitcode of the protocol implementation as an LLVM module. It then creates the initial program state (e_0) and extracts the corresponding initial FSM state (q_0) from the LLVM module by reading out the values of the protocol state variables. For each possible network event (τ_i), the extractor injects the event into the appropriate program variable and initiates symbolic execution of the implementation against the injected event. The execution may fork depending on the branch conditions on any symbolically marked variables (*e.g.*, `uip_appdata`) along with the execution path. Such conditions become the path-constraint of the execution path the code took responding to the event τ_i. Once the symbolic execution finishes, the FSM extractor analyzes each of the execution paths to extract the corresponding FSM state (q_j) by reading out the values of the state

variables and to create the corresponding transition from q_0 to q_j based on the constraints of the execution path. Next, the extractor again runs the symbolic execution for e_0 but against the event τ_{i+1} and so on. Once it finishes executions for e_0, the extractor repeats the same steps for the next program state e_j in the queue. Alongside, the extractor builds the **E-FSM** containing only distinct FSM states and FSM transitions observed during each round of the symbolic executions.

Note that the harnessed `main()` function in Fig. 2.6 calls the event dispatcher function once rather than calling the dispatcher function inside a infinite loop. We do this to ensure the termination of the symbolic execution of the program. However, to mimic the real execution that is calling the dispatcher function every time an event occurs, the FSM extractor starts the symbolic execution of the protocol implementation from the `main()` function against each event. This may seem counterintuitive as the extractor always starts the execution from the `main()` function. But, in reality, the extractor records the current program state[2] information (*i.e.*, the current values of all the non-symbolic global variables) before KLEE removes the current program state and uses this information to overwrite the memory of the program state to be used in the next round of the symbolic execution just before executing the `main()` function as if the program were executing from where it was left off. Thus we ensure the termination of symbolic execution, otherwise the symbolic execution would not terminate if we had an infinite loop in the `main()` function.

Recall that for an event-driven network protocol implementation, the dispatcher function is invoked each time a network event occurs. However, one execution of the dispatcher function is completely independent of the other. Therefore the implementation often retains the effects caused by a network event by modifying only the global, most likely the state variables. Similarly, the FSM extractor runs the symbolic execution multiple times, which invokes the event dispatcher function once for each event. To resemble the independent execution of the event dispatcher function, the

[2]The program state is different from the protocol state. A program state consists of the current values of all the global variables, whereas a protocol state is signified by the state (possibly global) variable(s).

extractor does not carry over the path constraints from the previous round of symbolic execution. However, by recording the program state information (as explained above), the extractor retains the effects on the protocol caused by the event.

2.3.3 FSM Translation and Model Checker

The FSM extractor outputs the **E-FSM** in a generic XML format that needs to be translated to the language specific to the model checker of the user's choice. For pLTL model checker, one could possibly choose (a) an explicit-state model checker such as SPIN [13] or (b) symbolic model checker such as NuSMV [78]. We choose NuSMV 2.5.4 as symbolic model checkers tend to support models with large state space.

We implemented the FSM translator in C++ (2 KLOC), which performs the following four steps: (i) Parses the XML-based intermediate representation of the FSM; (ii) Translates the constraints generated during symbolic execution, from the KQuery language [79] to a human readable version; (iii) Generates the atom-proposition mapping and stores it in a file; (iv) Translates the FSM into the SMV modeling language of NuSMV 2.5.4. Step (i), (iii), and (iv) are straightforward and hence we do not discuss the details here. The reader might wonder why would one need the step (ii). This is due to the fact that LLVM converts all the integers to bitvectors. KLEE uses the STP SMT solver [80] with ARRAY and BITVECTOR theory as the underlying theories. Hence, even a simple constraint such as $x + y + c \leq 5$ expressed in bitvector representation can possibly be unreadable by a human. Recall that, these constraints are referred to in the properties. It is thus reasonable to have a level of comprehensibility to the end user who is carrying out the verification process. For example, see Fig. 2.7 for a KQuery and its associated human-readable constraint in our language. Although a more intuitive notation would have been desired, it is very difficult to achieve it due to a whole different kind of possibility in the bitvector representation. In the example in Fig. 2.7, `uip_len[0,(2B)]` means the 2 byte value

which is read from the 0th byte position of the variable `uip_len`. In the example, `(4B)0x0 uip_conn[18,(2B)]` signifies that the 2 byte value read from the 18th byte position of `uip_conn` is zero-extended to obtain a 4 byte value.

```
array payload[1] : w32 -> w8 = symbolic
array uip_len[2] : w32 -> w8 = symbolic
array uip_conn[48] : w32 -> w8 = symbolic
(query
  [(Ule N0:(ReadLSB w16 0 uip_len)
        1)
   (Eq false (Eq 0 N0))
   (Eq 255 (Read w8 0 payload))
   (Slt 0
        (ZExt w32 (ReadLSB w16 18 uip_conn)))]
  false)
```

(a)

```
{{uip_len[0,(2B)]} <= 1}
&&
{{{uip_len[0,(2B)]} = 0} = FALSE}
&&
{{payload[0,(1B)]} = 255}
&&
{0 < {(4B)0x0 {uip_conn[18,(2B)]}}}
```

(b)

Figure 2.7.: Representation of a path constraint obtained from the Telnetd protocol shown in Fig. 2.2: (a) in KQuery language and (b) in our human readable version

2.3.4 Property Extraction and Verification

One of the most demanding parts of this analysis is coming up with desired properties that capture the state machine inconsistencies the implementation should not have. To obtain such properties, we explore the RFCs of the different protocols, academic papers and Internet blog posts talking about the protocol, bug report filed by the developers or third parties. One of the main problems we faced is that the properties one can derive are generally described at a very high level of abstraction.

We then need to specify them in the implementation level details that are captured by the extracted FSM.

To write the property, one would require the atom-proposition mapping file and the **E-FSM**. We want to acknowledge that writing properties in pLTL needs a significant amount of effort. However, one can use property patterns to write the desired properties in pLTL [81]. The property pattern gives a mapping between different requirements in natural language and their pLTL counterpart. There are also automatic tools that convert restricted natural language properties into pLTL formulas [81].

Once the property has been specified in pLTL, we add it to the input file along with the model specification given by the FSM translator and run the model checking. If the property has been violated by the model and a CEX is found, we forward the CEX to the spurious CEX checker for further scrutiny. If the CEX is found to be inconsistent, the spurious checker forwards an invariant which is added to the specification for the next iteration of verification.

2.3.5 Spurious CEX Checker

Once we have received the parsed CEX from the NuSMV model checker, we check consistency of each state and the transition in the CEX. For this we use the atom-proposition map file generated by the FSM translator. We automatically generate an SMT query from the CEX and the map file. For instance, for any state s in the CEX, we take each proposition and its truth assignment of the form $p_i \mapsto \mathsf{TV}_i$ where $\mathsf{TV}_i \in \{\mathsf{true}, \mathsf{false}\}$, replace p_i with its associated atom a_i and construct a quantifier first order formula of form $\bigwedge_i (a_i = \mathsf{TV}_i)$. We then call the STP SMT solver [80] with the query and ask for a satisfiable substitution for the free variables. If the query is satisfiable, we pass the CEX to the replay mechanism. However, if the query is not satisfiable, we forward the following invariant "$\bigwedge_{\mathsf{TV}_i=\mathsf{true}} p_i \wedge \bigwedge_{\mathsf{TV}_j=\mathsf{false}} \neg p_j \rightarrow \mathsf{false}$" to NuSMV using the `INVAR` keyword. This tells NuSMV that the states which do not

satisfy the invariant is an inconsistent state and should not be explored during state exploration.

2.3.6 Replay CEX

The replay mechanism takes a consistent CEX and try to concretely execute the CEX to check whether it is a realizable CEX. One possibility to execute the CEX is to solve the constraints $\bigwedge_i (a_i = \mathsf{TV}_i)$ in each state of the CEX using the STP SMT solver to get concrete values for the symbolic values then feed it to the replay mechanism of KLEE in an input file. Recall that, to verify a plausible CEX we also have to monitor that the state transitions in the CEX matches up with the state change in the code. This will require instrumenting the source to have assertions, which checks to see whether the state transitions and their corresponding actions match up. However, for a different CEX, we would have to heavily instrument the source code again. To avoid instrumenting the code to add assertions, we simulate the execution using KLEE's symbolic execution engine with concrete values. More precisely, at each step of the CEX, we solve the constraints of that step using STP SMT solver to get concrete values of each symbolic variables and overwrite the memory of each symbolic variable with their respective concrete value. This has the advantage that we do not require instrumenting the code at all. Additionally, we do not have to worry about timeout events. The program runs with concrete values and we monitor the program execution state to check whether the state transitions and actions of the implementation match up with the state transitions and actions in the CEX. If this is the case, we assume the CEX is a plausible one. When there is a loop in the CEX, we unroll the loop for a fixed number of iterations and execute it accordingly.

2.3.7 Optimizations

We now describe two optimizations that we introduce in our CHIRON implementation– one for preemptively ruling out spurious transitions generated by the FSM extraction

algorithm, and the other for speeding up the process of finding a consistent CEX during verification.

Modeling Network Stack Events

Since our FSM extraction is focused on network protocol implementations, network events play a critical role in extracting the **E-FSM**. An implementation of a network protocol relies on the underlying network stack protocols. For instance, the Telnet server protocol runs on top of TCP. The underlying network stack protocols can impose restrictions on the feasible order of network events. For example, in case of the Telnet server protocol, receiving a data packet from a client without an established connection with the client is not feasible. As our FSM extraction algorithm is oblivious to such restricted ordering of events, it will generate spurious transition when a restricted ordering of events needs to be respected. Equipped with this intuition, one can contemplate the following two approaches to ignore the spurious transitions during compliance checking.

(a) During the FSM extraction We allow the user to provide a restricted event model that regulates the order in which the network events can occur during an actual protocol execution. Our FSM extractor can exploit this restricted event model to generate the list possible events feasible in a particular FSM state of the **E-FSM** (See line 12 of Algorithm 1).

(b) During the verification In this approach, the FSM extractor attempts all possible events during the extraction of the **E-FSM**. However, we allow the user to provide a pLTL formula Ψ that captures the feasible order of network events in which they can occur in an actual execution. To verify the compliance of the implementation against the property φ, CHIRON checks conformance against $\Psi \rightarrow \varphi$ instead of φ.

Although both approaches rely on the user-provided input, which is based on domain knowledge, there are some major differences. The first approach has the

advantage that the **E-FSM** is closer to the **S-FSM**. Whereas the second approach yields both a larger FSM M and a larger formula $\Psi \rightarrow \varphi$ against which M is model checked. As the runtime complexity of model checking is dependent on both the model size and the formula size, the second approach incurs a high overhead during verification. Therefore, we select the first approach as it enables us to obtain a reduced FSM and also speeds up the verification process.

Initial Invariants

When an inconsistent CEX is encountered by the Spurious CEX checker, it forwards an invariant to the model checker to rule out the unsatisfiable transition for future verification. Let us consider that we have the following atoms: $\mathsf{payload}[0, (1\mathsf{B})] = 255$, $\mathsf{payload}[0, (1\mathsf{B})] = 254$, and $\mathsf{payload}[0, (1\mathsf{B})] = 253$. The atoms signify that the first byte of the payload buffer is 255, 254, and 253, respectively. Let us also assume that they are respectively mapped to propositions p, q, and r. According to the semantics of the propositions, the valid assignments are the ones where all of p, q, and r are false or only one of them is true at a time. Since the model checker is oblivious to the relationships between the atoms, it can require up to 4 steps (because 4 out 8 possible truth assignments are correct) before the Spurious CEX checker rules out all the spurious cases. However, with the following set of initial invariants we can easily rule out all the 4 unwanted truth assignments: $p \rightarrow \neg q \wedge \neg r$, $q \rightarrow \neg p \wedge \neg r$, and $r \rightarrow \neg p \wedge \neg q$.

With the increasing complexity of the implemented protocol, the number of propositions is likely to increase, which can consequently require exponential number of steps to find a consistent CEX.

2.4 Evaluation

In this section we assess the effectiveness and practicality of CHIRON. Specifically, we intend to answer the following research questions: (a) Is CHIRON applicable to

Table 2.1.: List of implementations evaluated using CHIRON

Protocol	Type	Implementation	Protocol Notation
Telnet	Server	Contiki 2.4	Telnet_C24
		Contiki 2.7	Telnet_C27
		FNET 2.7.2	Telnet_F
DHCP	Client	Contiki 2.7	DHCP_C
		FNET 2.7.2	DHCP_F

real protocol implementations in the wild? (b) Is CHIRON effective in finding non-compliances? (c) How much improvement can we gain by applying the proposed optimizations? (d) Is it possible to run compliance checking by CHIRON in a reasonable amount of time?

2.4.1 Setup

We first demonstrate the efficacy of CHIRON by using it to evaluate various implementations of 2 protocols: Telnet and DHCP. Telnet is a byte-oriented bidirectional communication protocol and often used as a mean to provide a command line interface for interacting with a (possibly remote) device. Despite being an old protocol, Telnet is still being used in the wild, for instance, by Android and other embedded systems' developers, and also by Cisco network administrators. DHCP is a binary protocol that assigns IP addresses to devices on a network. Because it greatly simplifies the network administration, it is widely used in both home and enterprise settings.

We have obtained a total of 5 implementations of these protocols from different TCP/IP network stacks: uIP (part of the Contiki OS) and FNET. In particular, we focus on the Telnet server and DHCP client implementations from the source trees of Contiki 2.4, Contiki 2.7, and FNET 2.7.2. We use Contiki 2.7 and FNET 2.7.2 because these were the latest releases at the time of evaluation. Contiki 2.4 came to our attention because of the bug reported on its Telnetd implementation [12]. A

summary of the evaluated implementations can be found in Table 2.1. For brevity, we will use the notation defined in Table 2.1 to identify each of the implementations in the rest of our discussion.

We have configured both the Contiki and the FNET based on their default configuration to enable TCP/IP support for IPv4, Telnet server, and DHCP client service. By default, the Telnet server for Contiki supports only one active client session. Therefore, to be consistent, we have configured the Telnet server for FNET to also handle at most 1 active session.

We run our experiments on a commodity machine equipped with an Intel Core i7-2620M CPU and 8GB of RAM, running Ubuntu 14.04 LTS with Linux kernel version 3.13.

2.4.2 Property Verification

We have obtained 11 representative properties for the Telnet server protocol and 7 for the DHCP client protocol to demonstrate the effectiveness of CHIRON. The list of properties as well as the verification results are shown in Table 2.2 and Table 2.3.

The properties (DP1 – DP7) for a DHCP client (see Table 2.3) are all extracted from the RFC [82]. They govern how a DHCP client implementation must react to various network events. However, for the Telnet server, we select the properties from various sources (see Table 2.2). The TP1 property is specific to implementations that support only one active client session at a time. TP2 – TP4 are obtained from the Telnet RFC [83]. They describe how an implementation must interpret incoming data and react accordingly. Properties like TP5 – TP7, not originated from specifications, are used to demonstrate how a developer can use CHIRON to reason whether the implementation transit correctly between states as desired. One important aspect of the Telnet protocol is the Network Virtual Terminal (NVT), which is an abstraction the Telnet protocol uses to overcome platform compatibility issues by starting with all options disabled. Many modest Telnet servers implemented for resource-constrained

Table 2.2.: Telnet server properties and verification results

Property	Property Description	Telnet_C24			Telnet_C27			Telnet_F		
		Valid	False	Factual	Valid	False	Factual	Valid	False	Factual
TP1	The server must not accept any new connections during an on-going session			✓	✓			✓		
TP2	If receive WILL after IAC, must send DO or DONT			✓			✓	✓		
TP3	If receive DO after IAC, must send back WILL or WONT			✓			✓	✓		
TP4	If receive IAC IAC, must consume the 2nd IAC as regular data	✓			✓			✓		
TP5	If receive IAC in NORMAL state, must go to IAC state and eventually go back to NORMAL state	✓			✓			✓		
TP6	If receive DO after IAC, must go to DO state	✓			✓			✓		
TP7	If receive WILL after IAC, must go to WILL state	✓			✓			✓		
TP8	For NVT, if receive DONT after IAC, must NOT send WONT			✓			✓	✓		
TP9	For NVT, if receive WONT after IAC, must NOT send DONT			✓			✓	✓		
TP10	For NVT, never send DONT request	✓			✓			✓		
TP11	For NVT, never send WONT request	✓			✓			✓		
	Total:	6	0	5	7	0	4	11	0	0

embedded systems, including the three we have evaluated, tend to remain as NVTs and not to provide support for any sophisticated options. Therefore, we have derived

Table 2.3.: DHCP client properties and verification results

Property	Property Description	DHCP_C			DHCP_F		
		Valid	False	Factual	Valid	False	Factual
DP1	If receive DHCPNAK in REQUESTING state, must immediately start over DHCP negotiation			✓	✓		
DP2	If receive DHCPOFFER in SELECTING state, must immediately send out DHCPREQ and move to REQUESTING state	✓			✓		
DP3	If receive no DHCPOFFER in SELECTING state and response timer expired, must resend DHCPDISCOVER	✓			✓		
DP4	If receive DHCPOFFER in REQUESTING state, must discard, not change state, take no actions	✓			✓		
DP5	If receive DHCPACK in REQUESTING state, must immediately move to BOUND state	✓			✓		
DP6	If receive no DHCPACK in REQUESTING state and response timer expired, resend DHCPREQUEST	✓			✓		
DP7	If receive no DHCPACK in REQUESTING state and state timer expired, start over DHCP negotiation	✓			✓		
	Total:	6	0	1	7	0	0

four additional properties (TP8 – TP11) from the Telnet RFC specifically targeting the implementations that intend to remain as NVTs. Failing to comply with such properties could lead to endless acknowledgment loops as pointed out in the Telnet RFC.

We have discovered a total of 10 non-compliance instances: 5 in Telnet_C24, 4 in Telnet_C27, and 1 in DHCP_C. We now describe the non-compliances discovered by CHIRON in details; however, we group the similar non-compliance instances together.

Non-compliance 1 (Accepting multiple client connections simultaneously) According to the Telnet server implementation for Contiki, the server is expected to have only one active session at a time. In other words, the server must not accept any new connection from a Telnet client during an on-going session, which we denote as the property (TP1). In our experiment, CHIRON generates a factual CEX for Telnet_C24 demonstrating that the Telnet server accepts a new connection from a client even if there is an on-going session. In fact, this is due to a state machine bug that can manifest upon receiving any additional connection. This bug was, however, already reported [12] and fixed in the later release of Contiki-2.5.

After a close inspection, we have identified that this state machine bug can be damaging as it has several implications: (a) inconsistent protocol behaviors as it causes the protocol to end up in an unexpected (correct, but not in this context) FSM state by taking an undesirable FSM transition and may change the program variables, possibly by re-initialization, and (b) security issues as it can affect confidentiality and integrity by sending the data intended for one connection to the other connection. Such a state machine bug can often remain undetected during the concrete execution of the implementation because of its nuances. Moreover, it also depends on the developer's ability to imagine such a scenario to manifest the bug using traditional testing approaches (*e.g.*, black-box testing), whereas we have discovered this non-compliance using CHIRON with a very little effort.

Non-compliance 2 (Failed to reply appropriate Telnet command) Both the Telnetd implementations from Contiki (Telnet_C24 and Telnet_C27) violate the two properties (TP2 and TP3) that require the Telnet server must reply back the appropriate Telnet command if it receives WILL (in case of TP2) or DO (in case of TP3) from the connected Telnet client. The factual CEX generated by CHIRON demonstrates that there exists an execution path in the real implementation where the Telnet server fails to send back its response to the received Telnet command if the buffer (*i.e.*, `telnetd_buf`) is full. For both the implementations, the Telnet server

uses this buffer to temporarily store all outgoing data including the Telnet command responses and sends the data over the network from time to time.

A careful inspection of the source reveals that the Telnet server (to be precise, `sendopt` function) does not check if it has failed to append the Telnet command response to the buffer. Moreover, the Telnet command response is stored in an array local to `sendopt`, which is lost right after the function returns. Therefore if the buffer is full, the Telnet server would not ever notice that the response to the received Telnet command has not been sent, which causes an interoperability issue since the Telnet client keeps on waiting for the reply from the server. Such a non-compliance attests to the effectiveness of CHIRON in finding subtle interoperability bugs.

Non-compliance 3 (Potential endless acknowledgment loops) Like the previous non-compliance, both the Telnetd implementation from Contiki (Telnet_C24 and Telnet_C27) violate the two properties TP10 and TP11. According to the Telnet RFC [83], the protocol must acknowledge a DONT (resp., WONT) command by sending out a WONT (resp., DONT) only if the received DONT (resp., WONT) command causes a change in the current mode; otherwise, it must not acknowledge. This is necessary to prevent potential endless acknowledgment loops– each party considers the incoming commands as new commands rather than the acknowledgments. Since both Telnet_C24 and Telnet_C27 are the basic implementation of the Telnet protocol (*i.e.*, as an NVT), a request to disable any option cannot make any change in the mode of the terminal, and therefore they must not acknowledge any DONT/WONT command requests. For both the implementations, CHIRON generates the corresponding factual CEX, which demonstrates the Telnet server actually replies back WONT (resp., DONT) when it receives a DONT (resp., WONT) command request from the client.

One can argue that a Telnet client never sends a DONT/WONT command when connected to an NVT Telnet server (like Telnet_C24 and Telnet_C27) since the client would not be successful to enable any option in the first place. However, there can

be two possible scenarios where such endless acknowledgment loops are feasible: (a) The first case happens when the Telnet client allows multiple new requests about an option that is currently under negotiation, and this is not explicitly prohibited in the Telnet RFC [84]; (b) Second case happens if the Contiki Telnet server connects with a (possibly faulty) Telnet client that initiates a DONT/WONT request and also acknowledges the received DONT and WONT commands. Consequently, such loops can impair the performance of the IoT devices running either Telnet_C24 or Telnet_C27 implementation. This non-compliance exhibits how CHIRON can be used to check if an implementation complies with its RFC specifications.

Non-compliance 4 (Failed to immediately start over DHCP configuration) According to the RFC [82] of the DHCP protocol, a DHCP client receiving a DHCPNAK message from the DHCP server as a response to its previously sent DHCPREQUEST message must restart the DHCP configuration process by sending a new DHCPDISCOVER message, which we denote as the property DP1. In our analysis of the DHCP client implementation for Contiki (DHCP_C), CHIRON generates a factual CEX demonstrating an execution path of the implementation that violates the property DP1.

A close inspection of the source reveals that DHCP_C does not handle the reception of a DHCPNAK message; instead, DHCP_C keeps on re-transmitting its DHCPREQUEST upon timeout for multiple times before giving up and restarting the configuration process. Such a reaction of DHCP_C to DHCPNAK messages, however, does not lead to any inconsistency in the protocol state/behavior. Nevertheless, this implementation hinders the performance of the DHCP protocol as it waits for a long time before starting over the configuration process. Uncovering such a non-compliance shows that CHIRON can help developers find some bugs capable of impairing the protocol performance.

Table 2.4.: Impact of various event models on FSM extraction. EM1 corresponds to the restricted model described in section 2.3.7, whereas EM2 considers all possible events with no specific order.

Protocol Notation	Event Model 1 (EM1)			Event Model 2 (EM2)		
	States	Transitions	Propositions	States	Transitions	Propositions
Telnet_C24	6	84	19	6	114	19
Telnet_C27	12	162	21	12	306	21
Telnet_F	7	18	11	7	34	11
DHCP_C	4	46	17	4	47	17
DHCP_F	8	80	45	8	140	45

2.4.3 Impact of Network Event Models on FSM Extraction

Table 2.4 shows the comparison between the extracted FSMs (*i.e.*, **E-FSM**s) using two different event models. Event Model 1 (EM1) corresponds to the user-provided restricted event model that regulates the order of the network events in which they can occur in an actual execution of the protocol. Whereas Event Model 2 (EM2) represents the less restrictive event model where any event can occur from the set of all possible network events. For both the event models, the **E-FSM**s contain the same number of FSM states. However, in case of EM2, the **E-FSM** has more transitions as expected. Most of these transitions are spurious since in reality such transitions can never occur. This result empirically supports our claim about the advantage of having a restricted event model as pointed out in section 2.3.7. Note that the number of propositions stays in the same in both the cases.

2.4.4 Execution Time of CHIRON

To evaluate the feasibility of CHIRON being a practical compliance checking framework, we report the execution time incurred by the major components of CHIRON as shown in Table 2.5. In this set of experiments, we have considered both optimizations: EM1 (the restricted event model) during FSM extraction and the initial invariants during verification. Each reported execution time is an average of ten

Table 2.5.: Run Time of CHIRON components (unit: seconds). For each protocol implementation, the value for **Property Verification** is the aggregate time for verifying all applicable properties. '–' denotes CHIRON found no consistent CEXs to replay.

Protocol Notation	**FSM Extraction**	**Property Verification**	**CEX replay**	**Total Run Time**
Telnet_C24	0.98	8.38	1.05	10.41
Telnet_C27	6.29	14.95	1.12	22.37
Telnet_F	0.16	1.64	–	1.80
DHCP_C	7.01	1.45	0.24	8.70
DHCP_F	15.09	3043.80	–	3058.89

independent runs. Note that, once the **E-FSM** is extracted, we can then use it for checking compliance of an arbitrary number of properties. For property verification, we report the total required time to check all the 11 properties for Telnetd and 7 properties for DHCP client. CEX replay is only applicable if CHIRON has found a consistent CEX.

Among the three Telnetd implementations, CHIRON requires the longest time to extract the **E-FSM** of Telnet_C27, which has a relatively larger **E-FSM** size (see Table 2.4). The same trend is observed in case of the two DHCP client implementations. Note that both the DHCP client implementations take longer time than the Telnetd implementations. This is due to the fact that, for each receive event, a DHCP client implementation handles a symbolic packet of size at most 552 bytes as opposed to a Telnetd implementation that handles 1 byte at a time.

CHIRON spends the majority of its execution time in verifying properties. The required time spent in the verification phase is influenced by the **E-FSM** size, the length of the properties, and the number of propositions. However, the verification time for DHCP_C is much smaller than its FSM extraction time, and this is because of having a relatively smaller **E-FSM** with a small number of propositions compared to other implementations. On the contrary, DHCP_F incurs a roughly 50-minute verification time because it has 1.7 times as many transitions and more than 2.5 times

as many propositions as its Contiki counterpart, DHCP_C. The high verification time is due to the increased number of propositions, which actually causes an exponential growth of the state-space of the model checker.

We also demonstrate the usefulness of adding initial invariants to speed up the convergence of finding a consistent CEX. For the purpose of comparison, we repeat the verification phase against the aforementioned properties. Without adding the initial invariants, the time required to finish the verification step for Telnet_F elevates to 333.4 seconds. Whereas none of the other implementations finished the verification for even one property within a time limit of 60 minutes. We also observe the benefit of the Spurious CEX checker during the verification of Telnet_F against the 11 properties as it was able to filter out a total of 1341 CEXs at an earlier stage.

2.5 Discussion

In this section, we briefly discuss how CHIRON can possibly be extended and also discuss threats to validity of our evaluation.

What if FSM states are not realizable explicitly through program variables Our FSM extraction technique requires the protocol FSM state to be explicitly realizable through program variables. However, one can implement a protocol **S-FSM** either by representing the states as *goto* labels in the program or by not even having any FSM representations at all. Even though it is possible to lift our analysis to handle state machines implemented using goto labels, how to capture implicit state information remains an open research question.

Which program variables to mark as symbolic In our analysis, only the constraints over symbolic variables are added to the transition of the **E-FSM**. Hence, if one desires to reason about certain variables in the program during analysis, then it is crucial that those variables are marked as symbolic. Now one obvious question the readers might ask is that in a network protocol implementation what variables

should be added as symbolic. One rule of thumb for network protocols is that any external or environmental variables (*e.g.*, packet buffer) which can influence the state transition of the protocol should be marked as symbolic.

Accuracy of compliance checking Since the **E-FSM** is an approximation of the FSM that is implemented in the source, CHIRON may provide incorrect verdicts on compliance checking of the protocol implementation. We have a two-step process for ruling out false non-compliance verdicts, nevertheless, it would require further refinement of the abstraction during FSM extraction to rule out false compliance verdicts. However, we want to emphasize that during our evaluation all compliance and non-compliance verdicts have been manually verified for further assurance.

Predicate abstraction To guarantee the termination of our analysis, we require that each program variable that explicitly constitutes the protocol state takes value from a small, finite domain. Obviously, this is restrictive and might not be satisfied by some stateful, event-driven protocol implementations. One possibility is to introduce predicates over the state variables that can take values from a large domain. Such abstractions (*i.e.*, predicate abstractions) will combine multiple concrete-valued variables into a single abstract state where the predicates' values are either true or false. Even though we will lose precision due to abstraction, it will enable us to handle large, possibly infinite, states of the protocol.

Alternative execution semantics CHIRON FSM extraction technique only considers non-concurrent, asynchronous C programs. For handling programs with alternative execution semantics, for instance, event-driven concurrent programs for TinyOS [85] or multi-threaded C programs will require revamping the symbolic execution engine [60, 86].

Threats to validity There are a couple of threats to validity in our evaluation. (1) KLEE has a high degree of non-determinism which may cause the different execution

times reported here to be not reproducible. (2) We used KLEE and STP SMT solver with their default configuration, and our reported results on execution time may not be the same for other configurations.

2.6 Summary

In this work, we have developed a framework CHIRON for checking whether an event-driven protocol implementation in C complies with some desired properties written as pLTL formulas. For checking compliance, we first extract the approximate FSM of the protocol (*i.e.*, **E-FSM**) automatically from the implementation with minimal developer input. Once we have extracted the **E-FSM** from the protocol source, we use a symbolic model checker to check the satisfaction of each of the desired pLTL properties against the **E-FSM**. When the property in question is violated, a counterexample (CEX) is generated by the model checker as evidence. We then use a two-step validation process to rule out the false CEXs. We have implemented CHIRON on top of KLEE and empirically showed the efficacy of it by uncovering several non-compliances of 5 protocol implementations from different network stacks.

3 ADVERSARIAL TESTING OF NETWORK PROTOCOL IMPLEMENTATIONS

Mobile ad-hoc networks allow a set of wireless nodes to communicate with each other without any central infrastructure. As traditional routing protocols do not perform well in a constrained environment such as wireless networks, significant work has been put into designing routing protocols for wireless networks. Examples include proactive protocols such as DSDV [22], and OLSR [23], reactive protocols such as AODV [24] and DSR [25], and hybrid protocols such as DST [88]. Additionally, there have also been efforts to improve the performance of the routing protocols by operating at the data link layer instead of the network layer, a representative example being the BATMAN [89] protocol. Given the increased threats that exist in wireless networks, several secure routing protocols have been designed. Examples include SAODV [90], ODSBR [91], ARAN [26], and Ariadne [92]. Many of the protocols mentioned above, such as AODV, ARAN, OLSR, DSDV, and BATMAN, were implemented and are available from public repositories [93–97].

Given the importance of routing as a fundamental component of wireless networks, many protocols have been subjected to model checking the design [27] and to testing the simulator-based implementation [28, 29]. For example, several model checking tools [27, 98, 99] were used to verify wireless routing protocols, and several simulators [28, 29] were used to demonstrate and test wireless routing protocols [22–25, 88, 100]. While model checking helps to verify the validity of the design, it does not provide a guarantee that the real-world implementation is free of bugs and vulnerabilities, since implementations contain optimizations not captured by the model, sometimes diverge from the design, and often introduce new bugs. In addition, while simulators

Some of the contents of this chapter is based on the joint work with Hyojeong Lee, Rahul Potharaju, Charles Killian, and Cristina Nita-Rotaru [87]

provide easier and simpler ways to describe a protocol, they sacrifice some aspects of realism such as the interaction of the protocol with the operating system components.

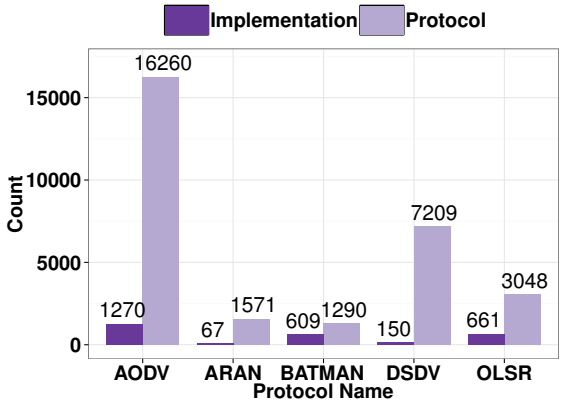


Figure 3.1.: Comparison of the routing protocols based on popularity (computed by searching on Google Scholar). *Protocol* counts indicate the total number of citations to the original research paper; *Implementation* counts indicate citations to the implementations and the URL of the software.

Fig. 3.1 shows the popularity of some wireless protocols in the academic community (obtained from Google Scholar) — it is evident that hundreds of researchers use the publicly available implementations for performance comparison across protocols [33–35], or to investigate properties of the network stack such as performance of TCP in multihop ad hoc networks [33, 36]. Thus, it is important to ensure that these implementations are robust and do not include faults and security vulnerabilities that may lead them to enter an unsafe state or exhibit degraded performance.

Some recent works like Gatling [30] showed the importance of performing adversarial testing for message-passing distributed systems. By testing systems implementations beyond just basic functionality (i.e. examining edge cases, boundary conditions, and ultimately conducting destructive testing), Gatling discovered vulnerabilities, many of which were not captured by model checking the design or by simulator-based testing. However, Gatling requires the target protocol to be implemented in the MACE language [101]. On the other hand, Max [31] focuses on two-party network protocols to find attacks that can manipulate the victim's execution

control flow by relying on the user specified information about a known vulnerability of the implementation to limit the search space and thereby catering itself as more suitable for corner cases.

In this work, we focus on adversarial testing of implementations of wireless routing protocols. We consider attacks and failures that are created through manipulation of protocol messages and are specific to wireless routing protocols, having a global impact on the protocol performance. We leverage the design of Turret [102]—an automated adversarial testing platform for distributed systems—to create an adversarial testing platform for wireless routing protocols. Turret uses a network emulator to create reproducible network conditions and virtualization to run unmodified binaries of systems' implementations. The platform requires the user to provide a description of the protocol messages and corresponding performance metrics. Turret's design is a good starting point for a cost-effective wireless testing environment because it allows a binary to run in its native operating system while limiting the impact of noise and interference on the performance of the system. Our contributions in this work are:

- We present Turret-W, a platform 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 heterogenous 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 costs required by the approach in [35]. In addition, our approach does not pose any restriction on the implementation language like Gatling [30], nor relies on a priori knowledge of any vulnerability like Max [31].

- We demonstrate attack discovery with Turret-W using detailed case studies on five representative wireless routing protocols: a reactive protocol (AODV), a secure reactive protocol (ARAN), and three proactive protocols (OLSR, DSDV, and BATMAN), whose implementations we obtained from public repositories. 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.

- We show that Turret-W also can find bugs, as it provides a testing environment that is realistic and controllable. Unlike attacks, bugs 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 AODV code and the operating system kernel.

3.1 Platform Overview

Our goal is to test wireless routing implementations, where the network conditions can be reproducible and also isolated from outside world interference. In our previous work [102] we created Turret, a platform for adversarial testing of message passing distributed systems. The design of Turret makes it an appealing choice for testing wireless network protocols because the emulation of the network ensures reproducible performance and limits the noise and interference, while the virtualized approach allows binaries to run in their native environments. However, Turret cannot be directly applied to wireless networks or routing protocols. Below, we first give an overview of Turret, the platform that we built on, and then explain what functionalities we added to support wireless routing protocols. We refer to Turret with our extension as Turret-W.

Table 3.1.: Message delivery actions supported by Turret

Action	**Action Description**	**Parameter**
Drop	Drops a message	Drop probability
Delaying	Injects a delay before it sends a message	Delay amount
Duplicating	Sends the same message several times instead of sending only one copy	Number of duplicated copies
Diverting	Sends the message to a random node instead of its intended destination	None

3.1.1 Overview of Turret

Turret is a platform for performance-related attack discovery in unmodified distributed system binaries. Turret uses virtualization (i.e. KVM [103]) to run arbitrary operating systems and applications, and network emulation (i.e. NS-3 [104]) to connect these virtualized hosts in a realistic network setting. Turret requires a description of the message formats that the system relies on, and a set of metrics that capture the performance of the system.

A controller bootstraps the system by starting NS-3 and running application binaries inside the virtual machines. Each of these virtual machines (VMs) acts as an individual node of the distributed system. The VMs communicate with each other with the help of the NS-3 emulator. Specifically, each VM is mapped to a node inside NS3, called a *shadow node*, through a *Tap Bridge* connection (available in NS-3), which connects the inputs and outputs of an NS-3 network device to the inputs and outputs of the VM's network interface (i.e., the corresponding TAP device of the VM) as if the NS-3 network device is a local device to the VM. The controller lets each shadow node know if it will act as a benign node or as a malicious node. A shadow node instructed to act as malicious will activate the *malicious proxy*, a component implemented by Turret on top of the Tap Bridge, to intercept messages generated by the application running inside the virtual machine and modify them according to

Table 3.2.: Message lying actions supported by Turret

Action	**Action Description**	**Parameter**
LieValue	Changes the value of the field with a specified value	The new value
LieAdd	Adds some amount to the value of the field	The amount to add
LieSub	Subtracts some amount from the value of the field	The amount to subtract
LieMult	Multiplies some amount to the value of the field	The amount to multiply
LieRandom	Modifies the value with a random value in the valid range of the type of the field	None

an *attack strategy*. An attack strategy may consist of two types of malicious actions: *Message Delivery Actions* that affect when and where a message is delivered (see Table 3.1) and *Message Lying Actions* that affect the contents of a message (see Table 3.2). In the case of message lying actions, different fields inside a message can be automatically modified based on the selected attack strategy and the user-provided message formats.

3.1.2 Limitations of Turret for Wireless Routing

Turret cannot be directly applied to wireless networks or routing protocols because of several limitations.

Distinguishing between control plane and data plane: While Turret can inject attacks and faults into any message-oriented protocol, it does not differentiate data messages from routing messages. In case of routing, many attacks on the data plane including degradation in the application performance can be amplified if the routing mechanism is disrupted. Thus, a platform intended for routing needs to control independently both the control (routing) plane and the data plane so that it can inject fine-grained attacks based on the type of the control plane messages and coarse-

grained attacks based on the service type of the data plane messages. For wireless networks, the separation is also needed to support basic attacks such as *blackhole* in which an attacker will drop all data messages but participate in the routing algorithm correctly.

Parsing homogeneous and heterogeneous packets: Turret expects an intercepted packet to contain only one message pertaining to the target protocol. Whereas routing protocols are typically designed to follow either *homogeneous packet format* (i.e. the routing protocol packs one type of routing message(s) into a single datagram) or *heterogeneous packet format* (i.e. the routing protocol packs different types of routing messages into a single datagram). In both cases, the length of the packet can be fixed or variable. Routing protocols designed for wireless networks generally adopt either packet formats, as communication is expensive in wireless networks.

Supporting non-IP packets: Turret assumes that the target protocol runs on top of Internet Protocol (IP) at the network layer. Thus, the malicious proxy processes each intercepted packet as an IP packet. However, not all existing wireless routing protocols use IP as the packet forwarding protocol at the network layer. For example, some protocols use geographic forwarding [105,106] where packets are forwarded based on physical proximity. Others such as BATMAN [89] or HWMP [107] operate at layer 2 (data link layer), instead of layer 3 (network layer), use MAC addresses for routing instead of IP addresses and transport routing information encapsulated into raw Ethernet frames. Therefore, it is important to support both non-IP and layer 2 routing packets to enable adversarial testing of such protocols.

Replaying packets: Turret does not provide the functionality to replay packets. Replaying packets is particularly interesting in case of wireless networks since it is a very low cost attack that can easily be launched. Note that packet replaying is different from packet duplication. In a replay attack, an attacker records another node's valid packets and resends them (without modification) later to other benign nodes via legitimate channels only if the packets contain the target control message(s).

This causes other nodes to add incorrect routes to their routing table. Such attacks can be used to impersonate a specific node or simply to disrupt the routing plane.

Establishing wormhole tunnels: Turret does not support colluding attacks. However, an attack specific to wireless networks that requires coordination between two attackers and is shown to be very detrimental is the wormhole attack where two colluding adversaries cooperate by tunneling packets between each other to create a shortcut in the network. As wormhole attacks are feasible (basic attack requires only two colluding nodes), it is important to be able to test the impact of wormhole attacks on the routing protocol.

3.1.3 Turret-W Description

We modified Turret to address the above limitations. The new platform, Turret-W, is shown in Fig. 3.2. The controller component coordinates the testing. It generates a topology file for the network emulator using a configuration file provided by the user. The configuration file specifies parameters such as the network topology, number of nodes, and number of malicious nodes. The controller then starts the virtual machines and binds each of them to the underlying network emulation layer. It then loads the routing service at the routing layer and instantiates the application at the application layer. It accepts the list of attack strategies created by the strategy generator and injects them into the malicious proxy. Finally, it collects log messages used to estimate the performance of the application running on top of the routing protocol.

Wireless network emulation: Like in Turret, the virtual machines operate on top of a network emulation layer provided by NS-3[1]. We configure NS-3 to emulate WiFi links. We leverage the Tap Bridge connection (available in NS-3) to connect a VM with its corresponding shadow node so that it enables a NS-3 net device to appear as

[1]Note that Emulab [108], MobiNet [109], Orbit [110] could also conceptually replace NS3. Emulab with fixed wireless provides more realism. However, the approach provides less reproducible results because of unwanted interference on the wireless channel and requires a separate implementation of the malicious version of the target routing protocol for each malicious node.

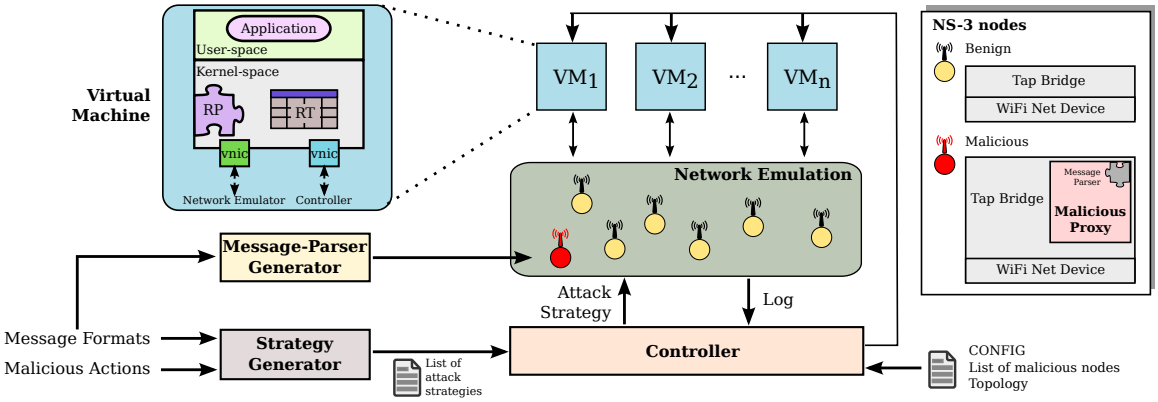


Figure 3.2.: Turret-W platform (RP denotes Routing Protocol, RT denotes Routing Table and VNIC denotes Virtual Network Interface Card)

a local device inside the VM thereby allowing the VM to use this local net device for WiFi transmission. The network emulation layer creates a virtual multi-hop wireless environment to transmit packets from a source to a destination virtual machine.

Attacks specific to wireless routing: We modified the Turret's malicious proxy (implemented on top of the Tap Bridge) to differentiate between messages originating from the routing layer and the application layer based on the port number. Differentiating data messages from routing messages allows Turret-W to implement a blackhole attack wherein a malicious node acts benign at the routing layer but selectively/entirely drops messages originating from the application layer.

We also provide support for a wormhole attack as follows: a wormhole tunnel is implemented as part of the malicious proxy inside NS-3 connecting any two colluding adversaries (precisely, shadow nodes). However, the routing code running in the virtual machines are oblivious of this tunnel, which introduces a new challenge to deal with. If we just forward data messages between the end nodes forming the wormhole, one side effect is that the routing will believe there is no direct link between the two end points of the wormhole. Therefore, to convince the routing services of the end nodes, forming the wormhole tunnel, that they are direct neighbors of each other, we allow these end nodes to exchange their own beacon messages (e.g., HELLO) over the wormhole tunnel. At the same time, the beacon messages forwarded between the ends

of the wormhole should be restricted only to those generated by the end nodes that form the wormhole and not their neighbors since that will results in incorrect updates of routing tables. All other routing protocol messages are forwarded by the colluding nodes over the wormhole tunnel so that they can perform the wormhole attack in the route discovery process. As a result, Turret-W supports all the malicious actions presented in Tables 3.1, 3.2, and 3.3.

Homogeneous and heterogenous packets: To inject a malicious action, the malicious proxy needs to be able to parse messages in order to act on different message types and to lie on a particular field of a message. The message-parser reads a message format description and outputs necessary source code that feeds into the malicious proxy. This source code contains a set of API calls (e.g., getMessageType(), getMessageSize() etc.) that expose properties of the message to the malicious proxy. An example message format description (a route request for AODV) is given below:

```
AodvRreq {
  uint8_t type = 1;
  uint32_t dest_addr;
  uint32_t dest_seqno;
  uint32_t orig_addr;
  uint32_t orig_seqno;
  ...
}
```

Routing protocols can follow either homogeneous packet format or heterogeneous packet format. For instance, AODV sends a route request message in a single UDP packet and thus, can be said to follow the homogeneous packet format. In contrast, OLSR allows individual messages be piggybacked and transmitted together in one transmission such as a topology control message and a HELLO message can be sent together in a single UDP packet. We modified the message-parser generator so that it can handle both homogeneous and heterogeneous packet formats and thus, enable testing of a wider variety of routing protocols.

Table 3.3.: Malicious actions added by Turret-W

Action	**Action Description**	**Parameter**
Replay	Records valid control messages from a node and resends them to other benign neighbors	None
Blackhole	Drops all data packets but participates in the routing algorithm correctly	None
Wormhole	Creates a wormhole between two colluding nodes and tunnels packets between each other	None
Wormhole with blackhole	Creates a wormhole between two colluding nodes and tunnels routing packets between each other, but drops all data packets	None

Packet forwarding protocols: Typically routing protocol implementations use Internet Protocol (IP) as the packet forwarding protocol at the network layer. However, developers are free to choose other packet forwarding protocols more suitable for the target network such as geographic forwarding for wireless ad hoc networks. The DSDV implementation [111] for the Click Modular router [112] is using such a protocol. Instead of IP, it is built on top of the Grid service [113] that is based on geographic forwarding. We modify the malicious proxy so that it handles routing messages packed into either IP or non-IP packets, and thus, we enabled the testing of routing protocols that are built on top of non-IP protocols.

Routing at layer 2: Traditionally routing protocols operate at layer 3 (the network layer) on top of IP (or some other packet forwarding protocols). However, several recently developed routing protocols (e.g., BATMAN) operate at layer 2 (the data link layer) where the nodes are attached to a unique Ethernet broadcast domain and are agnostic to the network topology. Moreover, routing in such protocols relies on MAC addresses instead of IP addresses. To enable adversarial testing of routing protocols like BATMAN, our malicious proxy supports injecting malicious actions into

routing messages even when they are encapsulated and forwarded as raw Ethernet frames.

Attack strategy generation: The strategy generator is responsible for generating a list of attack strategies that the target protocol should be tested against. For example, consider the following strategies in case of AODV where the malicious proxy is being instructed to duplicate each route request (AodvRreq) message 50 times and drop all the route error (AodvRerr) messages (i.e. 100%):

```
DUP AodvRreq 50
DROP AodvRerr 100
...
```

Given the message format description of the protocol under test, these attack strategies are generated based on the malicious actions listed in Tables 3.1 and 3.2 along with a value that decides the severity of that action. This attack strategy generation is inspired from prior work [**?**, 31, 32]. To support the additional wireless specific attacks listed in Table 3.3, we extended Turret's basic set of malicious actions with replay, blackhole, and wormhole attacks.

Support for multiple interfaces: Though Turret-W currently supports routing protocols that rely on a single network interface out-of-box, the platform can easily be extended to support routing protocols that leverage multiple network interfaces [114, 115]. In our current setup, each VM is equipped with only two network interfaces — one dedicated for the target routing protocol and another for other purposes (e.g., controlling the VM). Therefore, to enable testing of routing protocols that leverage multiple interfaces, we could equip the VMs with the necessary number of interfaces and configure the network emulator to detect these interfaces.

3.2 Methodology

We demonstrate our platform on real-world implementations of five representative wireless routing protocols: AODV [24], ARAN [26], OLSR [23], DSDV [22], and

BATMAN [89]. AODV is a well-known reactive (routes are determined on-demand) routing protocol whereas ARAN is not only reactive but also a secure routing protocol. On the other hand, both OLSR, DSDV, and BATMAN are proactive (routes are determined in advance) routing protocols. For AODV, ARAN, OLSR, and BATMAN we obtained the implementations from their public repositories [93–95, 97], while for DSDV, we obtained the implementation available in the Click modular router source [96]. It is noteworthy that the DSDV implementation runs on geographic forwarding and the BATMAN implementation operates at layer-2 (the data link layer).Next, we describe the attacker model, our experimental setup and the selection of system parameters.

3.2.1 Attacker Model

We focus on performance attacks mounted by malicious participants to disrupt the routing service thereby impairing the protocol performance, which is expressed by a *performance metric* that is when evaluated gives an indication of the progress the protocol has made towards completing its goals. To find such attacks, we measure the protocol performance, using the given performance metric, during each execution of the protocol in the presence of malicious participants in the network. The achieved performance is compared against a baseline performance obtained from an execution where all nodes are benign. We define an attack as follows:

Definition 1 - Performance Attack*: When the performance difference between a malicious execution and a benign execution is greater than a threshold, δ, we say that the attack strategy has resulted in a successful attack.*

Here, δ is a system parameter that depends on the protocol under test.

By directly testing real implementations running in their target operating systems, our platform captures the intricate interactions between the protocol being tested and the operating system components. In addition, the isolation and the reproducibility offered by the emulated and virtualization-based environment help us discover bugs

that impair the performance of the protocol even in a benign environment. Such bugs cannot be found in a simulation environment. We define a bug as follows:

Definition 2 - Performance Bug*: A performance bug is an implementation-level error that limits the practical utility of the protocol in a benign execution by causing 100% loss of application packets sent by the source.*

3.2.2 Experimental Setup

All our experiments are performed on a Dual-Quad core Intel(R) Xeon(R) CPU E5410@2.33GHz with 8 GB RAM host machine. We use Ubuntu 10.04.4 LTS to serve as the host OS. In all the experiments, we use 12 VMs, each allocated 128 MB RAM. For AODV, we use Debian 6.0.5 with Linux Kernel 2.6.32 as the guest OS. One of the advantages of our platform is that it allows us to execute binaries to run on their target operating systems. For instance, since ARAN requires an older kernel, we use Fedora Core 1 with Linux kernel 2.4.22 as the guest OS.

Our emulated network is a multihop wireless adhoc network. For the 802.11 MAC layer, we use 802.11a with a bit rate of 6 Mbps and a propagation loss model (called RangePropagationLossModel, available in NS-3) with a range of 100 meters for each link. We perform our experiments using a static grid topology. As an application on the VMs, we run *iperf* [116], a network benchmarking tool. In all the experiments, the performance of the application we report is averaged over ten runs.

We obtain a performance baseline using *benign testing*, where we randomly select pairs of source and destination nodes and transfer a stream of UDP packets between them for 30 seconds. Since we do not intend to stress the protocol implementation, we use a lower data rate of 128 Kbps so that the impact of attacks can be easily observed – a low packet delivery ratio implies an attack [91, 117].

As a performance metric, we use *packet delivery ratio* (PDR), i.e., a ratio of the total number of packets (in our case, application packets) received by the destination to the total number of packets sent by the source. PDR is easy to measure irre-

spective of the underlying routing protocol as it can be computed from the results produced by the application (i.e. iperf). Moreover, this metric does not require any instrumentation to the routing protocol implementation, which supports our goal of testing unmodified routing implementations. For each protocol, we capture the PDR achieved in each malicious execution and compare it with the baseline PDR. Given that we look for attacks that significantly degrade the performance, we argue that the measured baseline PDR can be used as a ground truth since it is always closed to the maximum (i.e., 100%) as per our experimental observation (see § 3.3- 3.7).

We select malicious node(s) randomly and inject malicious strategies during the entire experiment. We vary the total number of adversaries from 1 to 4 (out of the total 12 nodes) exhibiting a homogeneous behavior, i.e., we inject the same attack strategy to each malicious node. For every attack strategy applied to the routing messages, a malicious node drops application packets with a probability of p (a system parameter) to affect the performance of the application.

To demonstrate the effect of blackhole attacks and wormhole attacks, we perform experiments with three different configurations of adversaries: blackhole with one adversary, blackhole with two adversaries, combination of wormhole and blackhole with two colluding adversaries. When a blackhole attack strategy is injected, an adversary participates benignly in the routing protocol but drops 100% of application packets. The effect of a wormhole is noticeable in terms of application performance when combined with a blackhole attack. Remember that except for blackhole and/or wormhole attacks, we use the packet dropping probability p to drop application packets in all other malicious executions.

The threshold δ, a system parameter, is dependent on the protocol under test. The user can specify the threshold indicating the amount of performance loss he is willing to tolerate. Alternatively, it can be determined from ground truth by recording the observed performances for different attack strategies and select the threshold value that will detect the attack manifested by the weakest adversary from the set of the known attacks where a higher threshold means a more aggressive attacker. We relied

on the second approach. We consider blackhole with one attacker as the weakest adversary where the adversary drops all data messages but participates benignly in the routing protocol. Moreover, we know all the protocols we are testing are susceptible to blackhole attacks. Hence, we decide to choose 0.2 (i.e., 20%) as our threshold so that our tool can detect the blackhole attack. Intuitively, any successful attack strategy manifested by a relatively stronger adversary (attacks both the routing and the data messages) worsens the performance. Hence, the chosen δ would also be able to detect such attack strategies.

Overhead of Turret-W. Routing protocols usually use timeouts to prevent the use of stale information or provide reliability of transmission. When these timeouts expire, routing protocols take necessary measures such as removing stale entries from routing tables, restarting new route discovery, or entering recovery state. Turret-W can cause two different types of delays that will not be observed in real environment. First, it can cause a processing delay when the network flow is heavier than the network emulator capacity. Second, a malicious proxy can add delays while injecting malicious actions. The first type of delay is due to the nature of emulation based testing and can be prevented by over-provisioning. However, the impact of the second type of delay needs to be measured. To evaluate the amount of delay introduced by the malicious proxy, we performed experiments with AODV and OLSR protocols for the malicious attacks listed in Table 3.4. We observed that the delay is in the order of tens of μsec with a median of 40 μsec. Whereas the route expiration timeout used in AODV and OLSR are 5 sec and 6 sec, respectively. This result demonstrates that the computation of the malicious proxy of Turret-W does not have any significant impact on the routing protocols due to the low overhead.

Scalability of Turret-W: The scalability of Turret-W depends on (a) the scalability provided by the underlying emulator, and (b) the scalability of the routing protocol under test. Turret-W leverages the emulation environment of NS-3 and hence is subject to its limitations such as not being able to support large network sizes in the emulation due to the overhead related to the management of the large number of

threads in the NS-3 process [118]. As NS-3 is one of the most widely used network emulators and the performance of network emulation is not within the scope of our work, we choose a reasonable size of network consisting of 12 nodes and focus on networks that can still operate correctly under a reasonable number of malicious nodes (up to 30% of the total nodes).

3.3 Case Study 1: AODV

We now describe how we used Turret-W to test AODV [24]. All discovered attacks and bugs are shown in Table 3.4.

3.3.1 Protocol Description

AODV establishes a path on-demand. Specifically, when a source desires to send a message to a destination to which it does not have a valid route, it starts a *route discovery* process by broadcasting a route request (RREQ) message to its neighbors. Each node then forwards the first received RREQ by re-broadcasting it to its neighbors. This process continues until the RREQ reaches the destination or an intermediate node that has a valid route to the destination. In addition to forwarding the RREQ, each intermediate node records in its routing table (*i.e.*, *precursor list*) the address of the neighbor from which it receives the first RREQ, forming a reverse path. Once the RREQ reaches the destination node or an intermediate node with a valid route, the node responds to the RREQ by unicasting a route response (RREP) message to its precursor neighbor, *i.e.*, its neighbor on the reverse path, which in turn relays the RREP via precursor nodes back to the source node. From then on, the source node keeps unicasting the data to the next hop neighbor as long as the route is valid.

A node maintains connectivity with its neighbors by periodically broadcasting beacon messages (HELLO). Whenever the next hop becomes unreachable, the upstream node of the broken link propagates a route error (RERR) message to each of its upstream neighbors. Following the reverse path, the RERR finally reaches each

source node that contains the broken link on the route to its destination. A source then re-initiates the route discovery if a route to the destination is still desired.

Implementation used: We use AODV-UU-0.9.6 implementation publicly available from [93], which is RFC 3561 [119] compliant. The AODV-UU consists of two components — a loadable kernel module (`kaodv`) and a user space daemon process (`aodvd`). The kernel module intercepts and handles network packets by registering hooks (callbacks) with the Linux kernel's network stack. To register such hooks, `kaodv` uses the *Netfilter* framework [120]. The daemon (`aodvd`) uses netlink socket to communicate with `kaodv` and `NETLINK_ROUTE` protocol to communicate with the kernel routing table. We configure the protocol using the default values presented in [93].

3.3.2 Discovered Bugs

During the benign testing of AODV-UU, we discovered two unknown implementation bugs caused by a subtle interplay between the AODV-UU code and the kernel.

Bug 1. Kernel interaction order. In an attempt to measure TCP streaming performance between a source and a destination that are multiple hops away from each other, we observed that packets were not being delivered in the benign case. By design, whenever an application sends a packet for a destination to which the route is either invalid or unavailable, `kaodv` should hold the packet and notify `aodvd` to perform a route discovery. After finishing the route discovery, `aodvd` should notify the kernel to update the routing table and the `koadv` module to release the withheld packet. Our investigation revealed that in the AODV-UU implementation, the order of notification upon completion of a route discovery was incorrect, i.e., in the reverse order.

This bug could not have been discovered if we had not attempted to measure TCP performance where the first packet, *i.e.*, SYN packet is crucial to establish the connection. We also observed packet loss when initially using UDP, but like others,

we attributed this to the lossy behavior of UDP inside the wireless channel. We fix the bug by reversing the order of the two notifications.

Bug 2. Route packets harder. In the process of obtaining a baseline using iperf, we observed performance degradation over time despite the route being available and valid in the routing table. When the kernel transport layer hands-over any locally generated packet to the IP layer, `kaodv` receives the control of the packet via a hook registered with Netfilter. Thus, `kaodv` is responsible for returning a value to Netfilter so that Netfilter can decide what to do – accept/drop/ignore the packet or call the hook again.

When `kaodv` receives the control for a packet and already has a valid route, `kaodv` notifies Netfilter to continue processing the packet by returning `NF_ACCEPT`. On receiving `NF_ACCEPT`, Netfilter sends the packet down the network stack without performing any further iptables tests [121]. As a result, Netfilter does not send the packet to the correct next hop node on the route to the destination. We fix this bug by invoking `ip_route_me_harder()` inside `kaodv` before returning `NF_ACCEPT`.

3.3.3 Discovered Attacks

Attack caused crashing. We discovered an implementation attack that can cause all neighbors of a malicious node to crash. When a malicious proxy modifies an RREQ message to be an RREP by changing the type of the RREQ message, a recipient processes this altered RREQ message as an RREP message. The base RREP message (i.e., 20 bytes) is smaller in length than a base RREQ message (i.e., 24 bytes) [119]. Therefore, a recipient of the malformed RREQ message processes the message as if it were an RREP with extensions [119], and this causes AODV-UU of the receiver to crash with a segmentation fault. Our inspection reveals that the root cause is an integer overflow vulnerability in the AODV-UU code.

We show the related code snippet in Fig. 3.3. `extlen` is defined as an unsigned integer (line 2) and there is no checking if the extension length matches the actual

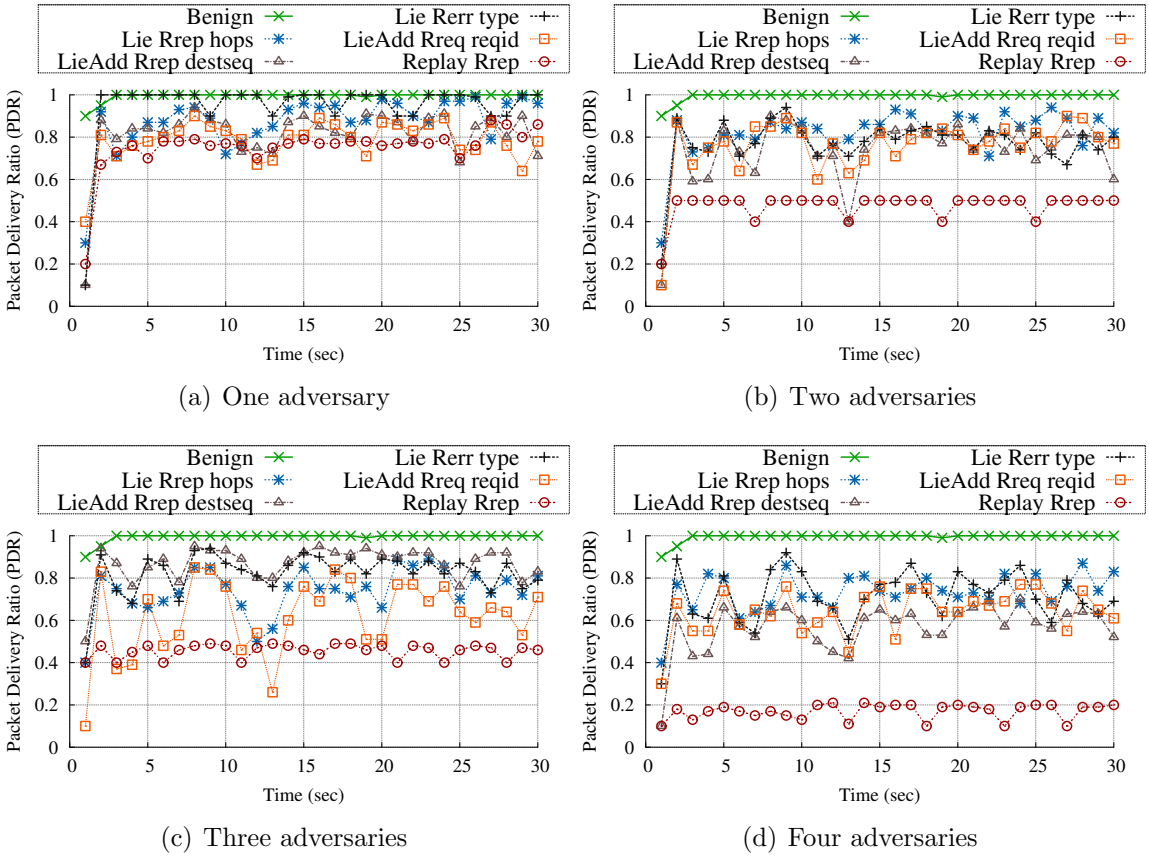
```
1.void NS_CLASS rrep_process(..., int rreplen, ...){
2.    unsigned int extlen = 0;
3.    AodvExtension *ext = rrep + RREP_SIZE;
      ...
4.    while ((rreplen - extlen) > RREP_SIZE) {
          // RREP_SIZE is 20
          ...
          // process extention according to the type
          ...
5.        /* read ext length from packet */
6.        extlen += EXT_HDR_SIZE + ext->length;
            // EXT_HDR_SIZE is 2
7.        ext = ext + EXT_HDR_SIZE + ext->length;
8.    }
      ...
9.  }
```

Figure 3.3.: Code snippet from AODV-UU showing the discovered integer overflow vulnerability

message size. In this case, the received buffer length (`rreplen`) is 24 bytes. Therefore, when the RREQ's originator seq number field value becomes 21 or bigger, this code will assume that the message has two extensions, one with 0 length and the other with length 21 or larger. At line 6, it will first increase `extlen` to be 2, which is the header size, then at the second iteration, it will add 2+21, and thus, `extlen` becomes 25. This results in an integer overflow on the left hand expression of the "while" condition at line 4, and therefore, the loop continues iterating. Later, the code crashes with a segmentation fault. This vulnerability can be fixed by enforcing careful type safety and boundary checking.

Attacks caused by malicious actions. We rediscovered several attacks on AODV-UU based on message delivery and lying actions that decrease the PDR below the accepted threshold. By design, AODV is known to be susceptible to these attacks [26, 90]. In case of our benign experiments, we observe a 98% PDR. Fig. 3.4(a)-3.4(d) show the temporal impact of the attacks on PDR as a function of the number of adversaries in the network. The impact of an attack increases as more nodes become malicious in the network.

Replay RREP. By replaying an RREP message received from a node, an adversary can fool its benign neighbors to believe that the originator is their one-hop neighbor. The benign neighbors that are at least two hops away from the actual originator believe the adversary is the originator node as they never receive RREP messages directly from the originator. This attack is more damaging than others because replaying the periodic HELLO messages causes these pseudo-links never to expire. We observe the PDR drops as low as 17% as the number of adversaries increases.



(a) One adversary

(b) Two adversaries

(c) Three adversaries

(d) Four adversaries

Figure 3.4.: Packet delivery ratio for the discovered attacks against routing messages of AODV-UU

LieAdd RREP destsq. Whenever a node receives a control packet from another node with the destination sequence number higher than what it has in its routing table, the node selects the route via this other node. A malicious node adds a positive

value with the destination sequence number of an RREP message, and this causes the recipient to select the route through the malicious node. In the case of 4 adversaries the PDR drops to 56%.

LieAdd RREQ reqid. Each RREQ message is uniquely identified by the request identifier in conjunction with the originator's IP. For each new route request, the request identifier is incremented by one. No node ever responds to an older RREQ message. A malicious node tricks the destination to respond to an RREQ with a future request identifier so that the source will be left with only one available route, i.e., through the malicious node. We observe that this attack causes the PDR to drop as low as 62% as the number of adversaries increases.

Lie RERR type and *Lie RREP hops.* Modifying the type of an RERR to RREQ causes the recipient to discard the packet. We find that adversaries can reduce the performance to 71% by performing this attack. Similarly, when a malicious node sets the hop count of an RREP to 0, the recipient selects the route through the malicious node as the recipient thinks that it can reach the destination by 1 (=0+1) hop. We observe that this attack causes the PDR to drop up to 73%.

Blackhole/wormhole attacks. We first tested AODV-UU against blackhole attackers (malicious nodes that drop all the data packets). We then introduce an additional blackhole node that colludes with the other blackhole node via a private channel to perform a wormhole attack. The PDR drops to 50% with the increase in blackhole nodes, whereas the PDR drops to 40% in case of the wormhole attack.

3.4 Case Study 2: ARAN

We now describe how we used Turret-W to test the implementation of ARAN presented in [122]. We summarize all discovered attacks and bugs in Table 3.4.

3.4.1 Protocol Description

ARAN [26, 122] is a secure reactive wireless routing protocol. ARAN introduces authentication, message integrity and non-repudiation by utilizing digital signatures on messages. Each node receives a certificate from a trusted certification authority (CA). The protocol consists of a route discovery process utilizing three types of routing messages: route discovery (RDP), route reply (REP), and route error (ERR). In essence, the route discovery process of ARAN is similar to that of AODV. In addition, ARAN guarantees end-to-end authentication. The routing messages are digitally authenticated at every hop, which ensures that only authorized nodes participate at each hop between the source and the destination.

Implementation used: We rely on the implementation *arand*-0.3.2 (referred below as ARAND), publicly available from [95]. This user space routing daemon built for Linux kernel 2.4 relies on the Ad hoc Support Library (ASL) [123] that provides an interface to the kernel functionalities required by any on-demand ad hoc routing protocol. ASL takes care of adding/deleting routes in the kernel routing table and notifying ARAND to initiate a route discovery for a destination in case of an unavailable route. The ARAND daemon also utilizes the functionality provided by the `route_check` kernel module of ASL to delete stale routes. For the cryptographic functionalities, it uses OpenSSL [124]. We use the default values for parameters as used in [95].

3.4.2 Discovered Bug

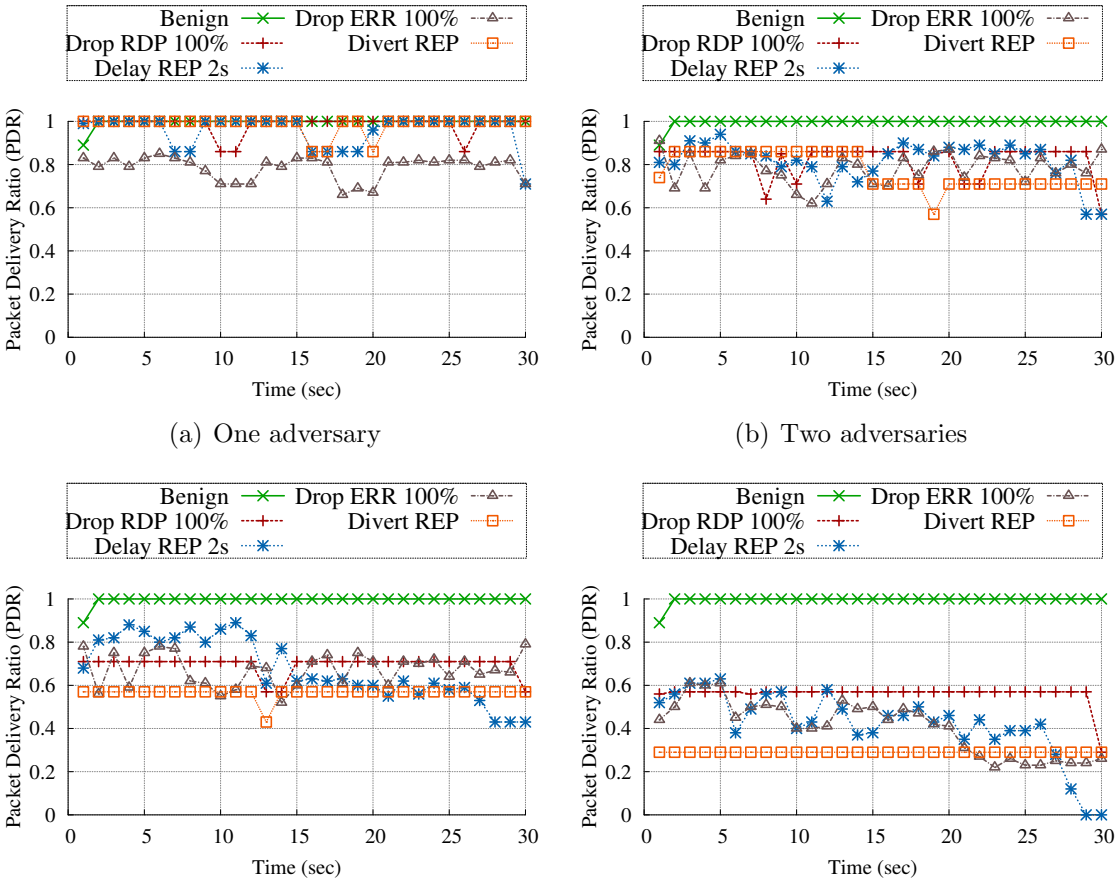
Bug. Wrong postal address. We discovered an implementation bug during the benign experiments in the setting of a multi-hop wireless network. By design, a route discovery request should be flooded via broadcast and the response should be delivered via unicast following the reverse path. However, in the implementation, upon receiving a response, an intermediate node attempts to forward the response directly to the source node (i.e., the originator of the route discovery) instead of the

correct next hop node that is on the reverse route to the source. If the intermediate node is more than one hop away from the source node, this response message cannot be delivered to the source, and thus, the route discovery fails. We fix this bug by letting the intermediate node use the correct next hop address to forward the route response. This bug is due to an implementation mistake that exists inside the `aran_processREP()` function defined in `aran.c` and manifests in topologies having nodes that are at least 3 hops away from each other.

3.4.3 Discovered Attacks

Attacks caused by message forwarding actions. We rediscovered several attacks on ARAND based on malicious delivery that have a significant impact on the performance. By design, ARAN is known to be susceptible to these attacks [91, 125]. We observe a 99% PDR when no attacks take place. We then measure the changes in the PDR achieved by ARAND as a function of the number of adversaries. Fig. 3.5(a)-3.5(d) show the temporal changes in the PDR achieved by *arand* as a function of the number of adversaries. The damage created by each attack increases with the number of adversaries.

Divert REP, *Drop ERR* and *Delay REP*: By diverting a route reply (REP) message and by dropping a route error (ERR) message, a malicious node can cause the most damage among these attacks. Both these messages are sent via unicast by design, and therefore, if an intermediate malicious node drops or diverts these messages, the upstream nodes on the route remain unaware of the on-going attack. Diverting REP messages disrupts the completion of route discovery whereas dropping ERR messages keeps the source unaware of the broken link and thus, prevents the source from re-initiating a route discovery for the destination. Four malicious nodes can drop the PDR to below 30% by diverting REP messages and to 40% by dropping ERR messages. On the other hand, delaying a REP message at an intermediate



(a) One adversary

(b) Two adversaries

(c) Three adversaries

(d) Four adversaries

Figure 3.5.: Packet delivery ratio for the discovered attacks against routing messages of ARAND

malicious node can reduce the PDR, but the impact is less significant as compared to diverting REP messages.

Drop RDP. An intermediate malicious node can drop a route discovery (RDP) message instead of re-broadcasting. This attack causes a slow decrease in PDR because every intermediate node re-broadcasts the RDP packet and therefore, even if a malicious node does not forward the RDP, the destination eventually receives the RDP message from other benign node(s).

Blackhole/wormhole attacks. We evaluate ARAND in the presence of blackhole/-wormhole attackers in the network. In the presence of one blackhole attacker, the

PDR drops to 80%. Adding another blackhole node drops the PDR to 42%. However, when two blackhole nodes collude with each other to perform a wormhole attack, the PDR drops to 28%.

3.5 Case Study 3: OLSR

We now describe how we used Turret-W to test OLSR [23]. All discovered attacks are shown in Table 3.4.

3.5.1 Protocol Description

OLSR [23] is a proactive routing protocol based on the traditional link-state algorithm where each node maintains topology information about the network by periodically exchanging link-state messages. OLSR minimizes the size of each control message and the number of rebroadcasting nodes during each route update by employing a multipoint relaying strategy. During every topology update, each node in the network selects a set of neighboring nodes, called *multipoint relays*, to retransmit its packets. To select the multipoint relays, each node periodically broadcasts a list of its one hop neighbors using HELLO messages. From the list of nodes in the HELLO messages, each node selects a subset of its one hop neighbors, which cover all of its two hop neighbors. Each node, then, disseminates information about the subset, i.e., the set of multipoint relays, using *topology control* (TC) messages that are retransmitted only by the multipoint relays of the node. Other nodes receiving these TC messages process them but do not retransmit. Each node eventually determines an optimal route (e.g., with minimum hops) to every known destination using the topology information and updates its routing table. During data transmission, this routing table is leveraged to determine route to a destination.

Implementation used: We use olsrd-0.6.3 (referred below as OLSRD) publicly available from [94], which is RFC 3626 [126] complaint. This implementation is a routing daemon that employs the `ioctl()` system call to communicate with the

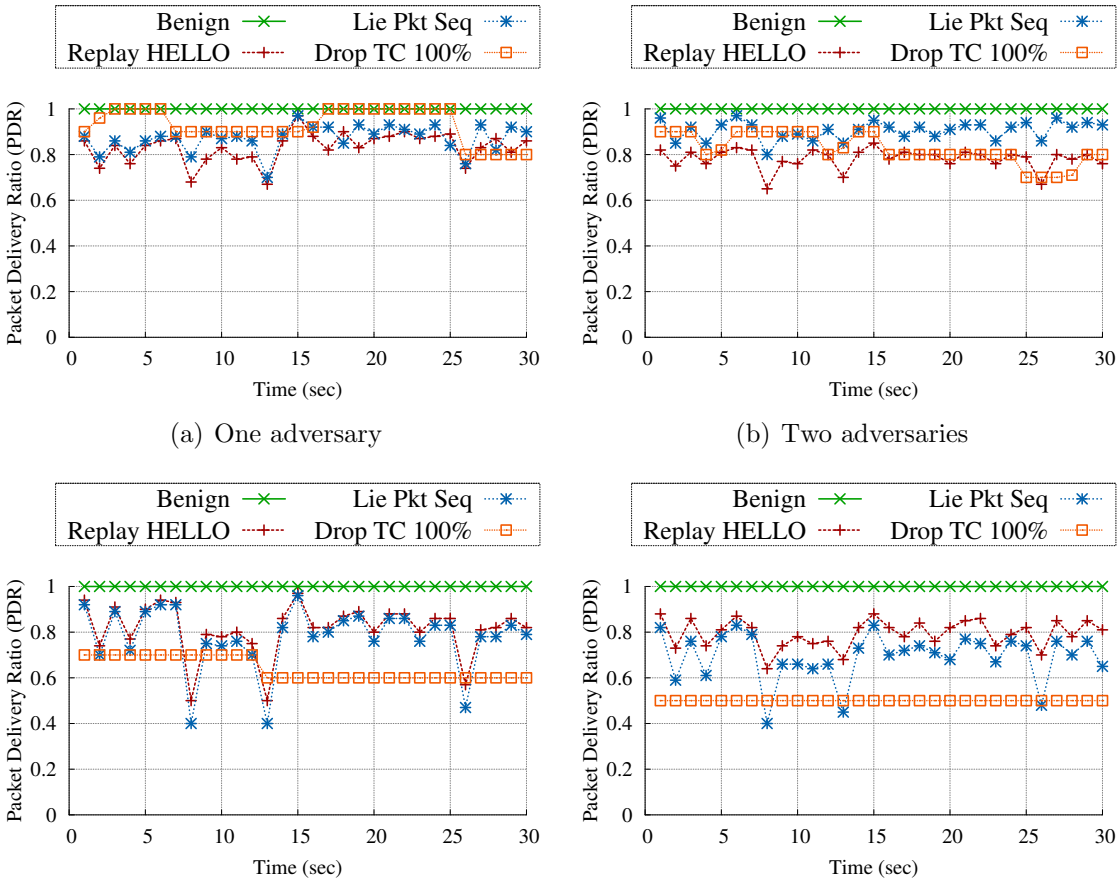
kernel and utilizes the NETLINK_ROUTE protocol to manipulate the kernel routing table. Unlike the above reactive protocols, it does not have any kernel module that intercepts the network packets from the network subsystem. The daemon communicates with other nodes over UDP and interacts with the kernel only when necessary, e.g., to add/delete a route to/from the kernel routing table, to enable IP forwarding, etc. We use the default values for parameters as used in [127].

3.5.2 Discovered Attacks

Attacks caused by malicious actions. We rediscovered several attacks in OLSRD based on message delivery and lying actions that have a significant impact on the application performance. By design, OLSR is known to be susceptible to these attacks [128–130]. We observe a 100% PDR in a benign scenario. We measure the impact of the attacks on PDR as a function of the number of adversaries in the network. Fig. 3.6(a)-3.6(d) show the temporal impact of the attacks on PDR as a function of the number of adversaries in the network.

Replay HELLO. When a node receives a HELLO message from another node, it adds the node to its neighbor list and starts broadcasting a new HELLO message. Based on the HELLO messages, nodes learn about their one hop neighborhood and select their multipoint relays that forward TC messages. By replaying a HELLO received from a neighbor, a malicious node can disrupt the routing service of its benign neighbors that are not direct neighbors of the originator of the HELLO. We observe the PDR to be around 80% on average, regardless of the number of attackers in the network.

Drop TC 100%. A TC message traverses the entire network via multipoint relays. TC messages are important because a node considers all the received TC messages to infer the network topology and thus, establishes a route to every other node. Therefore, an attack on TC messages is more damaging in that it will lead to inconsistencies in routing table of benign nodes. We observe that dropping TC messages results in at



(a) One adversary

(b) Two adversaries

(c) Three adversaries

(d) Four adversaries

Figure 3.6.: Packet delivery ratio for the discovered attacks against routing messages of OLSRD

most 50% drop in PDR. Note that while selecting malicious nodes randomly in our experiments, we do not add any constraints on the selection procedure.

Lie Pkt Seq. By design, OLSR follows the heterogeneous packet format where each packet is ordered by a sequence number. Before sending out a packet, a malicious proxy can replace the sequence number of the packet with a fake value (e.g. 0). This malformed packet causes disruption in route calculation. Four malicious nodes can drop the PDR of OLSRD up to 69%.

Blackhole/wormhole attacks. We measured the PDR obtained by OLSRD at the presence of three different configurations of blackhole and wormhole attackers: one

blackhole attacker, two independent blackhole attackers, a colluding pair blackhole attackers connected through a private channel. With the increase in blackhole nodes the PDR decreases as low as 50%. The combination of the wormhole and blackhole attackers makes the attack more significant as the PDR drops to around 30%.

3.6 Case Study 4: DSDV

We now describe how we used Turret-W to test DSDV [22]. All discovered attacks are shown in Table 3.4.

3.6.1 Protocol Description

The DSDV (destination-sequenced distance-vector) routing protocol is based on the Bellman-Ford family of algorithms that utilize distance vectors to calculate paths, between any two nodes in the network, along which data can be exchanged. DSDV is a table-driven proactive routing protocol, and therefore, each node maintains a routing table consisting of entries for every possible destination (not just the neighbors) along with the cost to reach the destination. As a cost metric, the protocol uses hop-count that is the number of hops a packet has to travel to reach its destination.

Each node periodically advertises its own routing table to its neighbors using HELLO messages. In addition, any changes to the routing table are propagated to other nodes as quickly as possible. These updates may lead to routing loops within the network. To avoid routing loops, each routing update from the node is tagged with a sequence number. Each node is free to choose an even number as the starting sequence number for the routing updates where the node is listed as the destination, but the node increments the sequence number by 2 for each periodic update. A sequence number defines the freshness of the route to the destination. Note that one node cannot change the sequence number tagged with such routing updates made by others. However, in case of a broken/expired link to one of its neighbor, the node can increment the sequence number by 1 and trigger an update mechanism. The

nodes receiving this update check the sequence number and if it is an odd number, they remove the corresponding entry from their routing table. Moreover, DSDV uses settling time to dampen the route fluctuations due to node mobility.

Implementation used: We use the DSDV implementation presented in [111] that is developed as part of the Grid project, which is built on the Click modular router [112] and written in the click configuration language. The code is publicly available from [96]. We refer to this implementation as DSDV-Click. All the states of the routing protocol are maintained inside Click elements and are accessed through Click. This implementation of DSDV can run either at the user-space using the Click user-space process or the kernel-space using the Click Linux kernel module. We chose the former due to its nature of high portability and easy debugging. At user-space, the Click process loads a network tunnel (`tun`) device, which the process considers as a file descriptor (e.g., `/dev/tun0`) and the operating system considers as a network interface (e.g., `tun0`). The Click process exchanges packets with the operating system's network stack using this tunnel device. We use the default values for parameters as used in [96].

3.6.2 Discovered Attacks

Attacks caused crashing. We discovered 4 implementation dependent attacks in DSDV-Click that cause all the neighbors of a malicious node to crash.

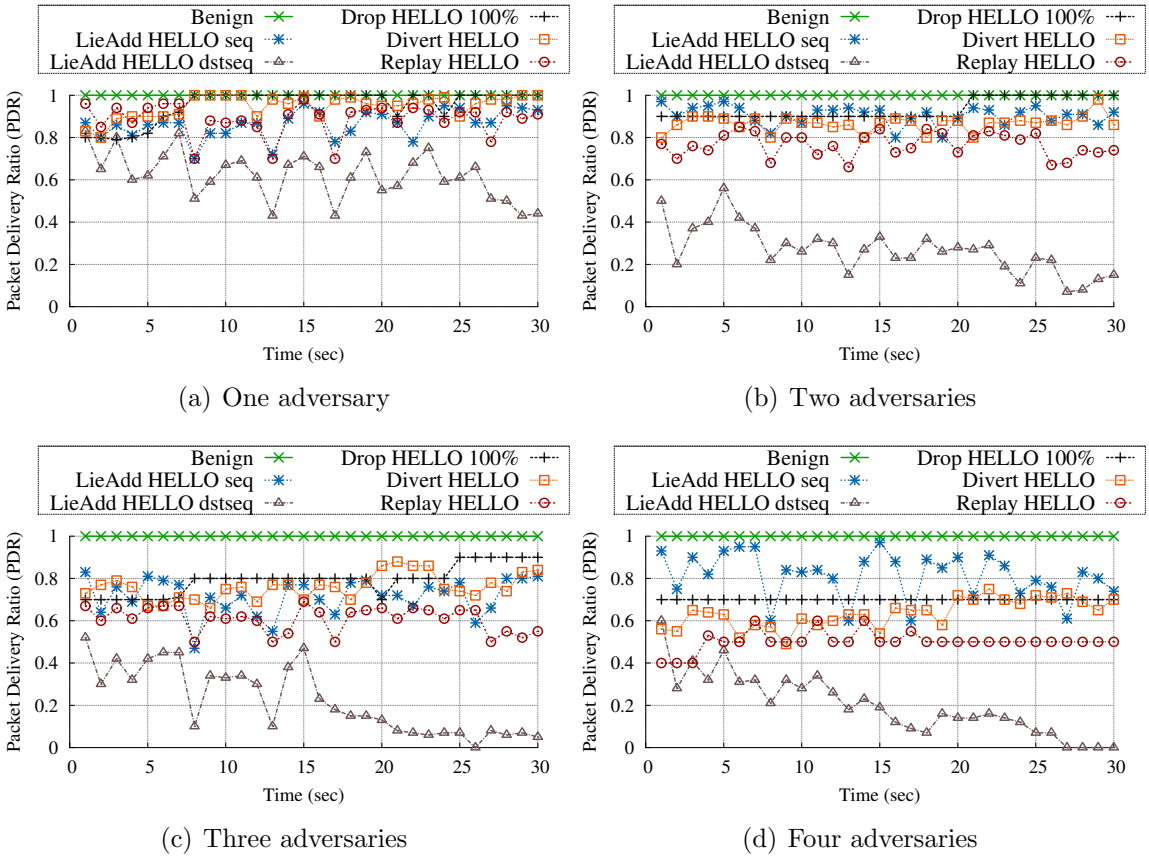
Lie HELLO seq or *dstseq with odd values*: We found that there can be multiple sequence numbers in a HELLO message. A node places its own sequence number (we refer to it as *seq*) as well as the sequence number of each destination (we refer to it as *dstseq*) that it is aware of into its HELLO messages. Whenever a node receives such a HELLO message, it checks if each advertised route is active. If so, each of the received sequence numbers must be an even number. Therefore, by simply lying on one or more of these sequence numbers, *i.e.*, by setting a positive odd number, a malicious node can cause each of its neighbors to fail an assertion check and crash.

Lie HELLO hopcount with 255: While advertising routes to other destinations, the originator node also includes *hopcount* (i.e., the number of hops to reach each of them from the originator) into its HELLO messages. We found an attack where an adversary can exploit the integer overflow vulnerability associated with the *hopcount* field, which is one byte in length. The adversary maliciously advertises routes with a value of 255 as the *hopcount*. Whenever one of the adversary's neighbors receives such advertisements and decides to update its routing table, the node adds 1 to the received *hopcount*. This addition overflows the field causing the node itself to crash due to an assertion failure.

Lie HELLO dstseq with even values: Turret-W helped us discover another crashing attack that is very subtle and delicate in terms of its execution. In this attack, the malicious node always modifies the route advertisements with a positive even number as the destination sequence number (*dstseq*), which apparently looks correct according to the protocol. However, a positive even number as *dstseq* is not correct for an advertisement of an expired route. Therefore, whenever the malicious node sends advertisements about the recently expired routes with a positive even number as *dstseq*, an assertion check on the neighbors causes them to crash.

Attacks caused by malicious actions. Like other protocols, we also found several attacks in DSDV-Click that impair the application performance. By design, DSDV [22] is known to be susceptible to these attacks [92, 131, 132]. We measure the changes in PDR achieved by the application as a function of the number of adversaries in the network where each node employs the DSDV-Click as the underlying routing protocol. Fig. 3.7(a)-3.7(d) show the temporal changes in PDR achieved by the application as a function of the number of adversaries in the network where each node employs the DSDV-Click as the underlying routing protocol. In the benign case, we observe a 100% PDR.

LieAdd HELLO seq and *LieAdd HELLO dstseq*. Recall that, in DSDV, each node maintains a routing table consisting of entries for all possible destinations (not only neighbors) and periodically advertises its routing table to its neighbors using beacon



(a) One adversary

(b) Two adversaries

(c) Three adversaries

(d) Four adversaries

Figure 3.7.: Packet delivery ratio for the discovered attacks against routing messages of DSDV-Click

messages (i.e., HELLO). Each of these messages contains the sequence number (*seq*) of the node itself along with zero or more entries for other destinations that the node is aware of at that very moment. Each additional entry includes the received sequence number (*dstseq*) of the corresponding destination. A sequence number tagged with a route defines the freshness of the route— a higher sequence number indicates a more recent route. Therefore, whenever a node receives a HELLO message from another node with the destination sequence number higher than what it is aware of, the node selects this new route. A malicious node can exploit this fact and add a positive even number to the destination sequence number contained in a HELLO message, and this causes the receiving nodes to select the route through the malicious node. Note

that instead of a positive even number, if the adversary chooses to add a positive odd number, the attack will cause the neighbors to crash (as explained earlier) since DSDV expects the sequence numbers defined by the originators to be positive even numbers.

According to our experimental results, adding positive even numbers to the *dstseq* field is more damaging than performing the same attack on the *seq* field. We can attribute this to the fact that by modifying the *seq* field the adversary just offers a more recent route to itself whereas by modifying the *dstseq* fields the adversary offers more recent (but not legitimate) routes to other destinations containing itself on these paths. Our experiment results show that the achieved PDR can drop from 62% to 20% with the increase in the number of adversaries when the adversaries perform such attacks on the *dstseq* fields. However, in case of such attacks on the *seq* field, we observe the PDR to drop from 86% to 72%.

Drop HELLO and *Divert HELLO*. The DSDV protocol requires nodes to exchange only HELLO messages as control packets pertaining to the routing service to establish a routing table. Therefore, when a malicious node drops all of its own HELLO messages, no other nodes within the network will ever be aware that the malicious node is active. As a result, the source node selects a path longer than the shortest one if the malicious node is on that shortest path. Similarly, when a malicious node sends its own HELLO messages to randomly selected nodes instead of broadcasting the messages, only a few nodes will know about the existence of this node. However, every node eventually learns the route to the malicious node due to the route advertisement mechanism of the DSDV protocol. The cost metric of these routes may not be the real optimum value. Consequently, the source may end up using a longer path than the original shorter one. In both the cases, we observe the PDR drops roughly from 95% to 65% with the increase in the number of adversaries.

Replay HELLO. In this attack, an adversary re-broadcasts the HELLO messages received from the neighboring nodes without any modification. As a result, any two benign neighbors of the adversary that are multiple hops away from each other (in

reality) consider themselves as 1-hop neighbors. Moreover, these false links never expire as long as the attack continues. In this case, we observe the PDR changes from 88% to 50% as the number of adversaries increases.

Blackhole/wormhole attacks. To test DSDV-Click in the presence of blackhole/-wormhole attacks, we followed the same approach as for the other protocols. In the presence of one blackhole attacker, we observe a PDR of 80% whereas the PDR drops to 63% when we introduced another blackhole adversary. Note that in case of the wormhole attack, the PDR drops to 49%.

3.7 Case Study 5: BATMAN

We now describe how we used Turret-W to test BATMAN [89]. All discovered attacks are shown in Table 3.4.

3.7.1 Protocol Description

BATMAN is a proactive routing protocol for multi-hop wireless adhoc networks. Unlike link-state protocols, BATMAN does not determine the whole path to the destination, nor does it requires the global view of the network topology to route packets. Instead, it requires each node to maintain only the best next hop to every other node in the network using collective intelligence, similar to a distance-vector protocol. Therefore, information about any topological change in the network does not need to be instantly spread throughout the network.

Each node periodically broadcasts an originator message (OGM) to inform its existence to its neighbors. The neighbors then rebroadcast the message to their neighbors and so on and so forth. Therefore, every node is aware of the existence of every other node in the network but records only the list of direct neighbors that it has received such messages from. The best next-hop to each destination is selected based on a metric called Transmit Quality (TQ), which measures the probability of a successful transmission of a packet on the link between the node and the next-hop.

As a result, each node only knows who to handover the data (encapsulated in Unicast messages) destined to a node that is multiple hops away. The data is handed over to the best next-hop neighbor, which in turn repeats the mechanism until reaches its destination.

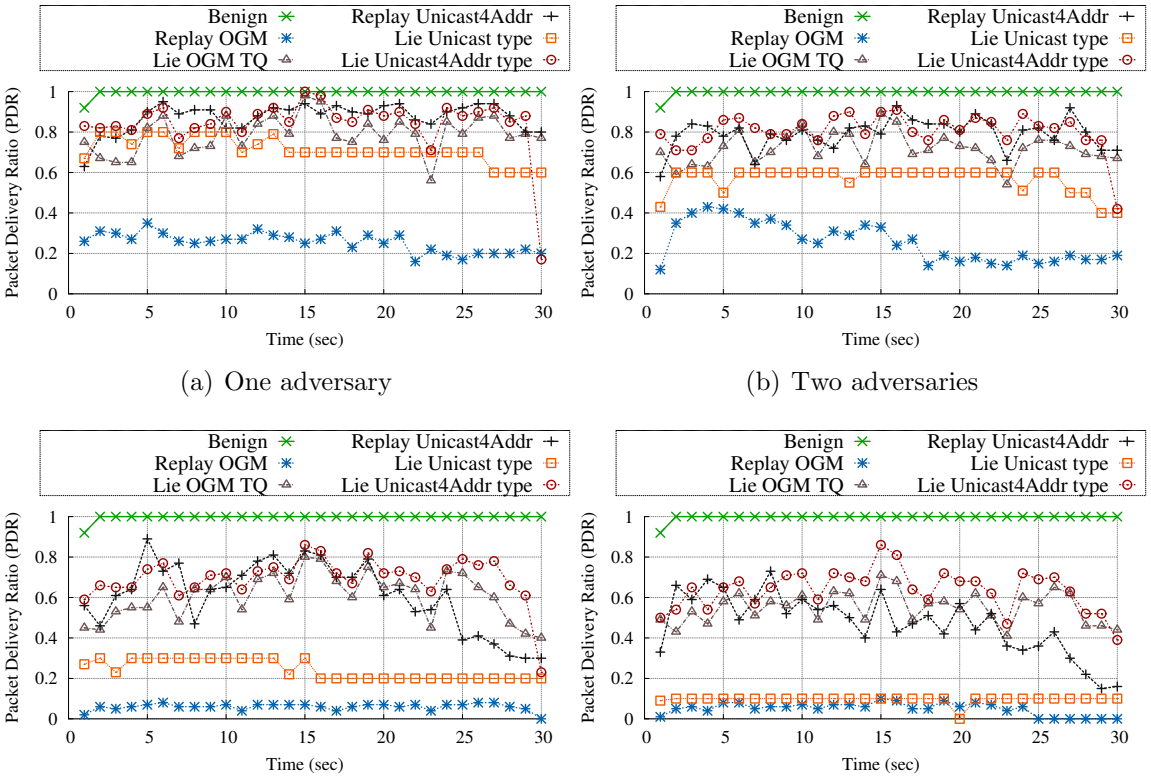
BATMAN utilizes a distributed ARP table (DAT) to enable nodes to perform faster ARP lookup operations. In essence, DAT mechanism creates an ARP cache distributed across the nodes by storing ARP entries as the ARP requests/responses travel through the network. Unlike traditional ARP requests, given an IPv4 address, a node can identify the group of nodes that may contain the related ARP entry by utilizing a distributed hash function. Instead of broadcasting, requests are sent as unicast messages (Unicast4Addr). If there is no response to the request, the requester node can fallback to the traditional ARP mechanism and broadcast the ARP request.

Implementation used. We use Batman-adv-2014.1.0 (referred below as Batman-adv) implementation publicly available from [97]. This implementation is a kernel-space implementation running at the data link layer where both the routing information and the data traffic are encapsulated and forwarded as raw Ethernet frames. Hence, the network communication does not depend on IP. The protocol emulates a virtual network switch connecting all the nodes as if the nodes are link local, and therefore unaware of the network topology. To reduce the packet processing overhead incurred by a user-space routing daemon, this version of the routing protocol is implemented as a Linux kernel module.

3.7.2 Discovered Attacks

Attacks caused by malicious actions. We rediscovered several attacks on Batman-adv based on message delivery and lying actions that decrease the PDR below the accepted threshold. By design, the BATMAN protocol is known to be susceptible to these attacks [89, 133]. In case of our benign experiments, we observe a 97% PDR. We then measure the impact of the attacks on PDR as a function of the

number of adversaries in the network. Fig. 3.8(a)-3.8(d) show the temporal impact of the attacks on PDR as a function of the number of adversaries in the network. The impact of an attack increases as more nodes become malicious in the network.



(a) One adversary

(b) Two adversaries

(c) Three adversaries

(d) Four adversaries

Figure 3.8.: Packet delivery ratio for the discovered attacks against routing messages of Batman-adv

Reply OGM. By replaying the originator messages (OGMs) received from a node, an adversary can induce its benign neighbors to consider the originator as a direct neighbor since OGMs are used to announce the existence of nodes in the network. This disrupts the routing service substantially since these replayed OGMs propagate through the network thereby affecting the best next-hop selection at the nodes that are closer to the attacker than to the originator. This attack is more damaging than others since replaying OGMs causes these pseudo-links never to expire. We observe that the PDR decreases from 24% to 6% with increasing adversaries.

Lie OGM TQ. When sending an OGM, the originator initializes the transmit quality (TQ) field with its maximum value of 255. Prior to re-broadcasting an OGM, the forwarding node sets the TQ field with a value that is the TQ of the received OGM times its measured TQ towards the last hop node via which it received the OGM. As a result, the TQ field of an OGM indicates the probability of successful transmission of a packet towards the originator along the path the OGM has traversed. A malicious node exploits this fact by setting the TQ field of all outgoing OGM to 255 thereby enticing the neighbors to select itself as the best next-hop neighbor towards the originator. Our experiment results show that the PDR drops from 77% to 54% as the number of adversaries increases.

Replay Unicast4Addr. When the source has to retrieve the MAC address of the destination, it computes the group of nodes that may contain the related ARP entry and sends Unicast4Addr messages. In this attack, an adversary replays all the Unicast4Addr messages containing either ARP request or response. Though this attack cannot directly disrupt the routing table, it can overload the network with Unicast4Addr packets when the number of adversaries increases in the network because the adversaries collectively create a ripple effect by replaying each received Unicast4Addr message. Moreover a Unicast4Addr message is quite smaller in length compared to a message carrying data traffic. As a result, this ripple effect affects the forwarding of the data traffic through the network. In our experiments, we observe the PDR drops from 77% to 47% as the number of adversaries increases.

Lie Unicast type and *Lie Unicast4Addr type.* The source encapsulates the data traffic in Unicast messages and hands over to the best next-hop neighbor and so does the next-hop neighbor until the data reaches the destination. By lying on the type field of a Unicast message, the adversary disrupts the data forwarding as the modified Unicast message is not interpreted as the data message. We observe the PDR drops from 70% to 9% with the increase in the number of adversaries. On the other hand, when the adversary modifies the type field of a Unicast4Addr message, it can disrupt the ARP request for a while. However, after a timeout, the requester falls back

to traditional ARP mechanism and broadcasts the ARP request, which eventually reaches the destination or some intermediate nodes that can reply with the related ARP entry. Therefore, in case of this attack, we observe that the attack is only effective when the number of adversaries in the network is larger than 2 causing the PDR to drop to 63%.

Blackhole/wormhole attacks. We test Batman-adv against blackhole/wormhole attacks in the same way as we did for other protocols. We observe that the PDR drops from 60% to 47% as the number of blackhole attacker increases from 1 to 2. When these two attackers collude to create a wormhole, the PDR drops to 42%.

Table 3.4.: Attacks and bugs (re-)discovered by Turret-W. Attacks/bugs with (*) means newly discovered.

Protocol Impl.	**Discovery Type**	**Name**	**Description**
AODV-UU 0.9.6 [93], Reactive, Updated: Apr 13, 2011	Attack*	Lie RREQ type 2	Lie about RREQ message type by setting to 2 (RREP) (causes crashing)
	Attack [26]	Lie RERR type 1	Lie about RERR message type by setting to 1 (RREQ)
	Attack [26,90]	Lie RREP hop 0	Lie about the hop count in route response to be 0
	Attack [26]	LieAdd RREQ reqid 10	Increment the route request id of route request by 10
	Attack [26,90]	LieAdd RREP destsq 10	Increment the destination sequence number of route response by 10
	Attack [26,90]	Replay RREP	Replay both route response and hello messages
	Attack [90]	Blackhole	Drop all data packets
	Attack [26,90]	Wormhole + Blackhole	Colluding malicious nodes drop all data packets
	Bug*	Kernel interaction order	Notifies the two components about the route discovery in a wrong order
	Bug*	Route packets harder	Returning `NF_ACCEPT` from hooks causes Netfilter not to check iptables
ARAND 0.3.2 [95], Reactive, Updated: Jan 31, 2003	Attack [125]	Drop RDP 100%	Drop each route request message
	Attack [125]	Delay REP 2s	Delay forwarding of route response message by 2 seconds
	Attack [125]	Divert REP	Divert route response message
	Attack [125]	Drop ERR 100%	Drop route error message
	Attack [91]	Blackhole	Drop all data packets
	Attack [91]	Wormhole + Blackhole	Colluding malicious nodes drop all data packets

Continued on next page

Table 3.4.: *Continued*

Protocol Impl.	**Discovery Type**	**Name**	**Description**
	Bug*	Wrong postal address	Intermediate nodes forward REP to the source instead of the next hop
OLSRD 0.6.3 [94], Proactive, Updated: Jun 5, 2011	Attack [128–130]	Replay HELLO	Replay a HELLO message received from a neighbor
	Attack [128–130]	Drop TC 100%	Drop all topology control messages
	Attack [128–130]	Lie Pkt Seq 0	Lie about the sequence number in olsr pkt to be 0
	Attack [129, 130]	Blackhole	Drop all data packets
	Attack [129, 130]	Wormhole + Blackhole	Colluding malicious nodes drop all data packets
DSDV [96], Proactive, Updated: Sep 24, 2011	Attack*	Lie HELLO seq 255	Lie about own sequence in HELLO messages with 255 (cause crashing)
	Attack*	Lie HELLO dstseq 255	Lie about the dest. sequences in HELLO messages with 255 (cause crashing)
	Attack*	Lie HELLO hopcount 255	Lie about the hopcount in HELLO messages with 255 (cause crashing)
	Attack*	Lie HELLO dstseq 254	Lie about the dest. sequences in HELLO messages with 254 (cause crashing)
	Attack [131]	Replay HELLO	Replay all HELLO messages received from neighbors
	Attack [131]	Drop HELLO 100%	Drop all HELLO messages
	Attack [131]	Divert HELLO	Divert own HELLO messages
	Attack [132]	LieAdd HELLO seq 10	Increment the own sequence number of HELLO messages by 10

Continued on next page

Table 3.4.: *Continued*

Protocol Impl.	**Discovery Type**	**Name**	**Description**
	Attack [132]	LieAdd HELLO dstseq 10	Increment each destination sequence number in HELLO messages by 10
	Attack [92, 131]	Blackhole	Drop all data packets
	Attack [131]	Wormhole + Blackhole	Colluding malicious nodes drop all data packets
Batman-adv 2014.1.0 [97], Proactive, Updated: Mar 13, 2014	Attack [89, 133]	Replay OGM	Replay an OGM message received from a neighbor
	Attack [89]	Lie OGM TQ 255	Lie about the transmit quality in OGM to be 255
	Attack [89]	Replay Unicast4Addr	Replay an Unicast4Addr message received from a neighbor
	Attack [89]	Lie Unicast type 0	Lie about the type of an Unicast message to be 0
	Attack [89]	Lie Unicast4Addr type 0	Lie about the type of an Unicast4Addr message to be 0
	Attack [133]	Blackhole	Drop all data packets
	Attack [133]	Wormhole + Blackhole	Colluding malicious nodes drop all data packets

3.8 Summary

Given the importance of routing in wireless networks, it is critical to subject their implementations to adversarial testing before deployment. To aid developers in this task, we develop Turret-W, an adversarial testing platform for wireless routing protocol implementations with minimal physical resources. We demonstrate our system by evaluating actual implementations of AODV, ARAN, OLSR, DSDV, and BATMAN. In total, we (re-)discovered 37 adversarial attacks capable of either crashing the benign nodes or reducing their performance by disrupting the routing service and 3 implementation bugs that impair the protocol performance in benign environment.

4 INFECTION MITIGATION IN EMERGING NETWORKS

With the proliferation of smartphones, Internet-of-Things, the number of wireless devices with complex capabilities has rapidly increased. While the openness of such wireless devices—supported by various open source operating systems like Google's Android [37], Contiki [11], FreeRTOS [38], Raspbian [136], and their development platforms—induces developers' motivation, it also introduces new propogation vectors for mobile malware. Recent reports show a surge of malware incidents targeting smartphones [39–41] and IoT devices [42–44].

Significant research focused on propagation modeling, detection, and application profiling of malware in the context of wired networks [48–52]. Those results do not model mobile malware which spreads directly from device to device by using short-range communication such as WiFi, Bluetooth or NFC [9, 45–47]. Mobile malware propagation has been studied using mean field compartmental models [137] which assume that each infected node will contact every neighbor once within one time step, *i.e.*, the infectivity is equal to the connectivity. Such models do not take into account that mobile malware does not spread at an even contact rate, as spreading requires devices to be within each other's proximity which in turn depends on user mobility. Most previous research on mobile malware has either not considered mobility [138–140] or has given limited considerations to it [141, 142]. Approaches that have considered mobility have used popular models like the random waypoint model which, as it has been shown, does not realistically mimic human mobility [143].

The content of this chapter is based on the joint work with R. Potharaju, C. Nita-Rotaru, S. Sarkar, and S.S. Venkatesh [134, 135]

While there has been work studying mobile malware propagation, the problem of *infection containment* in wireless networks was less studied. The work of [144] analytically studies containment of infection in a mobile network through countermeasures such as reducing communication range of nodes during an infection outbreak. The work does not consider realistic mobility models and does not propose concrete protocols to deploy and activate such countermeasures. The work in [145] introduces replicative and non-replicative patch disseminations assuming a network cost function and proves that the dynamic control strategies have a simple optimal structure. However, the impractical determination of the healer activation time and the lack of inclusion of the resource cost incurred by each patch dissemination make the techniques difficult to apply directly to energy constrained realistic scenarios.

In this work, we take the first step towards designing countermeasures for malware propagation under the presence of realistic mobility in a practical scenario. We investigate the dependence of infection spread on the underlying mobility model in order to systematize the design of countermeasures. We introduce the concept of *healers* to mimic the recovery process in a standard epidemic model and we focus on *static healers*, (*i.e.*, immobile) healers, which represent a realistic model because they can be directly mapped to real-world scenarios. For instance, static healers can be considered as cellular base stations (where no two stations cover the same cell in most cases) and the mobile nodes can be considered as users carrying mobile phones (moving with a certain mobility model). In contrast to the mechanism shown in [141], our static-healers are not white-worms and do not deactivate infected nodes. Our contributions are:

- We show that the infection spread in mobility models that mimic human behavior is slower than standard mobility models due to different contact rate and spatial distribution characteristics. We compare the Truncated Levy Walk

(TLW) and Random Waypoint (RWP) mobility models and show that the epidemic spread in TLW is *relatively slower* compared to RWP. This finding indicates that when designing countermeasure mechanisms, the time constraints are less tight than believed and that time-dependent assumptions can be relaxed to some extent, resulting in relatively lower consumption of energy.

- We model countermeasures to malware spread using static healer nodes. Static healers once placed in the area, act *independently* to deploy a patch when they sense nodes in their proximity. A healer-based solution optimizes: (i) the time it takes to heal the entire system by patching all the infected nodes and (ii) the total number of patches broadcasted. We formulate the optimal solution based on static healers as a T-Cover problem , which is NP-Complete.
- We use ORACLE, a $log(n)$ greedy approximation algorithm, that computes the optimal healing time knowing the placement of the static healers and *the future*, i.e. the exact time instances when the infected nodes arrive within each healer's proximity.
- We propose a novel healer placement strategy using blue-noise distribution generating Poisson Disk Sampling. We show that unlike random placement that results in many overlapping healers which cover the same area, our method allows healers to cover disjoint areas, thus enabling them to independently cover more infected nodes.
- We design three families of healer protocols: *randomized* (RH), *profile* (PH), and *prediction* (PDH), that allow for a trade-off between the energy consumed for sending patches and the time taken to recover the entire system. The intuition behind each protocol is as follows: (1) RH uses randomization to ensure simplicity in healers' functionality and achieves reasonable performance. (2) PH uses system feedback to optimize the energy consumed for sending patches, but

may result in a larger recovery time. (3) PDH predicts the cost of waiting for a suitable time instance to deploy a patch thereby achieving a smaller recovery time but has the side-effect of utilizing more patches. We compare our protocols with the ORACLE protocol and show through simulations that despite lacking knowledge of the future, our healers obtain a recovery time within a 7.4x∼10x bound of the ORACLE.

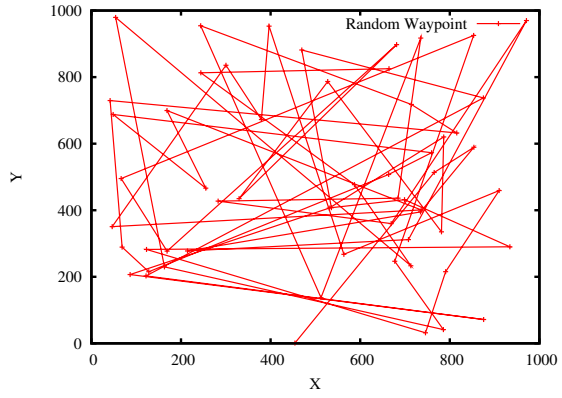
4.1 System Model

In this section, we construct a framework for analyzing the propagation of malware over a mobile ad hoc network that relies on epidemic theory to capture both the spatial interaction of nodes and the temporal dynamics of infection propagation.

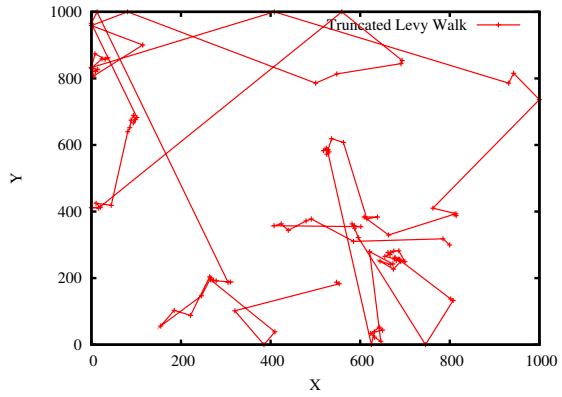
4.1.1 Mobility Models

Due to the difficulties in adapting real-trace data to long running simulations [140], we decided to use analytical models derived from real-trace data instead. Specifically, we use the *Random Waypoint* (RWP) and *Truncated Levy walk* (TLW) mobility models to generate synthetic mobility traces. We selected RWP because it is a typical mobility model used to study mobile malware propagation. We selected TLW because it provides more realistic representations of statistical patterns found in human mobility. Unless otherwise noted, we use a node velocity of 0.6 m/s to mimic low velocity realistic human mobility in both mobility models throughout this chapter.

Random Waypoint (RWP): RWP is a widely used mobility model [146–148] and includes pause times between changes in direction and/or speed [25]. A mobile node begins by staying in one location for a certain time period (pause time). Once this time elapses, the mobile node chooses a random destination in the simulation area



(a) RWP



(b) TLW

Figure 4.1.: Tracing the path of a single node: (a) for RWP and (b) for TLW

and a speed that is uniformly distributed between $[v_{min},\ v_{max}]$. The mobile node then travels toward the newly chosen destination at the selected speed. Upon arrival, the node pauses for a specified time period and starts the whole process again (see Fig. 4.1(a)). RWP is heavily used for mobile ad hoc network simulation [149] to simulate mobile nodes that can move randomly and freely in a mobility area without any restriction. This model is super-diffusive because of high-probability of long flights. On the contrary, human walks have heavy-tail flight distributions [150] that are not captured by common mobility models such as RWP.

The initial random distribution of mobile nodes is not representative of the manner in which nodes distribute themselves when moving as the instantaneous mobile node neighbor percentage possess high variability [151]. We use the approach suggested by [149] and discard the initial 1000 seconds of simulation time produced by RWP in each simulation trial.

Truncated Levy Walk (TLW): Based on the empirical studies performed on human mobility data collected through moblie devices carried by humans, Rhee *et. al.* [143] reported that human walks performed in outdoor settings of tens of kilometers resemble a truncated form of Levy walks commonly observed in animals such as spider monkeys, birds and jackals. A *Levy walk* is a type of random walk in which the increments are distributed according to a heavy-tailed probability distribution, *i.e.*, their tails are not exponentially bounded. The distribution used is a power law of the form $y = x^{-\alpha}$ where $1 < \alpha < 3$. TLW is a random equivalent mobility model for human walks in that it can describe some important characteristics of human walks (e.g. flight length, pause time and inter-contact time) despite being a random model. Inter-contact times are defined to be the time durations between two consecutive meeting events of the same two nodes. Human walks have long inter-contact times, which is intuitive in a sense that as humans do not move much, they will not meet each other very often. The distributions of these inter-contact times, which follows a power-law distribution with an exponential tail, are similar to those observed in case of Levy walks. Similarly, the heavy-tail distributions of flight length and pause time can be captured by Levy walkers moving in a confined area. Intuitively, Levy walks consist of many short flights and exceptionally long flights that nullify the effect of such short flights (see Fig. 4.1(b)).

Note that while there are other recent human mobility models similar to TLW such as the ones proposed by Lee *et al.* [152], Boldrini *et al.* [153] and Isaacman *et*

al. [154], our end goal is to advocate the use of one of these human mobility models while designing defenses against epidemic outbreaks.

4.1.2 Infection and Recovery Models

We adapt two classic epidemic models (SI and SIR) to take into account mobility. First we give a brief overview of the SI and SIR models, then describe how we use them to model malware propagation and node recovery in a mobile network.

SI Model. The SI-model is a two-state compartmental epidemic model, *i.e.*, a node can stay in one of two states: *susceptible* and *infected*. A susceptible node is vulnerable and can be exploited to be infected which in turn can infect other susceptible nodes. In this model, once a susceptible node is infected, it stays that way. The parameter that characterizes the model is the infection rate, β.

SIR Model. The SIR Kermack-McKendrick model [155] assumes that an infected node can be recovered. Specifically a node can be in one of the following states: *susceptible*, *infected*, and *recovered*. Nodes flow from the susceptible group to the infected group and then to the recovered group [156] as shown in Fig. 4.2. The model is characterized by two parameters, the infection rate β and the recovery rate α.

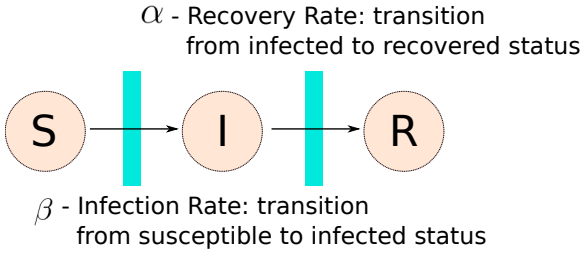


Figure 4.2.: SIR model: S, susceptible; I, infected; R, recovered

Mobile Infection Model. The SI model makes the unrealistic assumption that each infected node will contact every neighbor once within one time step, *i.e.*, the infectivity is equal to the connectivity. To take into account mobility, we assume the

nodes are moving according with a mobility model and we define infection spread as a function of a parameter c which we call the *probability of successful transmission*. At each time step, for every node X, we find the neighbors of X that are capable of infecting X. For each of these neighbors, we generate a random number from a uniform distribution between $[0, 1]$ and if this value is smaller than c, then X becomes infected.

Mobile Recovery Model. We adapt the SIR epidemic model as follows. Infection is modeled as in the mobile infection model above. We map node recovery through a healer that will change the state of an infected node to recovered through a healing mechanism. Once recovered, a node can no longer be infected, thus if no new nodes are added the infection will eventually disappear. The healing mechanism is distributed through a patch, a healer can send at most once during an interval of time called *epoch*, denoted as τ. We assume that healers are static, resource constrained, and act independently. Our assumptions also include that once an infected node receives a patch, the node instantaneously applies the patch and becomes completely recovered. We assume that there is no packet loss but note that it is straightforward to extend our model to a model having packet loss.

This model is characterized by the way the healers are placed and by the frequency with which they send patches. All healers are activated once the number of infected nodes in the system reached a system-wide parameter.

4.2 Infection Dynamics

In order to understand the infection dynamics of the two mobility models, we first describe our methodology and then explain the results that we observed.

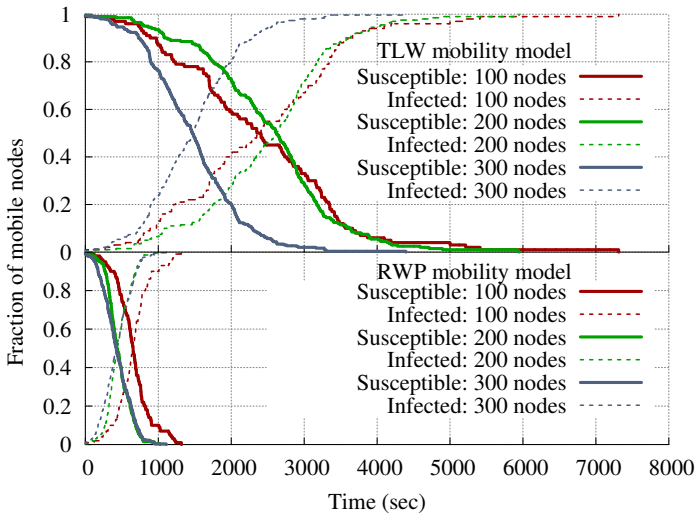
4.2.1 Methodology

We use the infection model described in previous section with the parameter that controls the infection rate, $c = 0.3$ [157] to mimic a more realistic infection scenario where infection spreads slowly. We generate RWP traces by using the methodology outlined in [158] and TLW traces by using the algorithm outlined in [143]. We perform our simulations using NS-3 [104]. We simulate the behavior of a system with 100, 200, and 300 nodes in a fixed area. All results have been averaged over ten simulation runs.

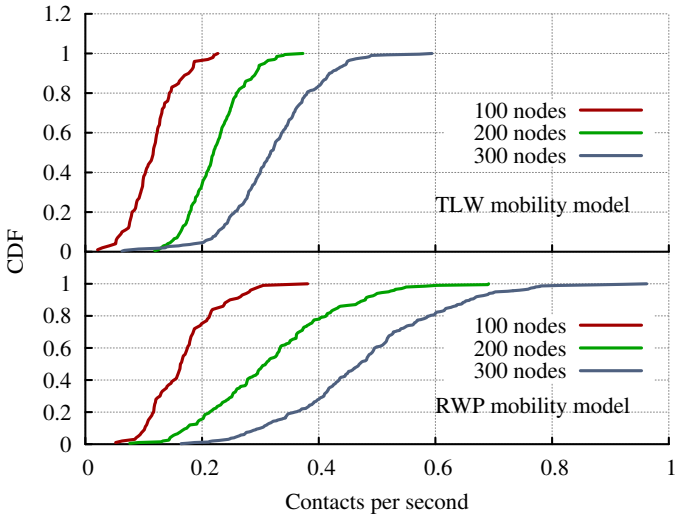
We define an *inversion point* to be the time instant where 50% of the population is infected. We use this metric to indicate the *first* point in time where the number of infected nodes surpasses the susceptible ones, thus *inverting* the scenario. Intuitively, an inversion point characterizes how fast the infection is propagating in an epidemic system.

4.2.2 Results

Figure 4.3 shows the infection dynamics in RWP and TLW mobility models. Observe that the inversion point for RWP occurs quite early in the simulation (Fig. 4.3(a) indicates a time around 500 seconds) in comparison with TLW (Fig. 4.3(a) indicates times between 1500-3000 seconds). This indicates that the time required to infect the system is far less in case of RWP differing almost by a factor of 3 from TLW. To the best of our knowledge, this phenomenon has not been observed before as most earlier research [141, 142] has studied these mobility models in isolation. As protocols are to be designed mostly for realistic mobility models (TLW in this case), this comes as a good news in that certain assumptions such as time-constrained-ness of a protocol can be relaxed to some extent, resulting in relatively lower consumption of energy.



(a)



(b)

Figure 4.3.: Infection dynamics: (a) Inversion point in RWP and TLW (the infection spread is slower in TLW) (b) Explaining the slow propagation (the contact rate in TLW is less than RWP)

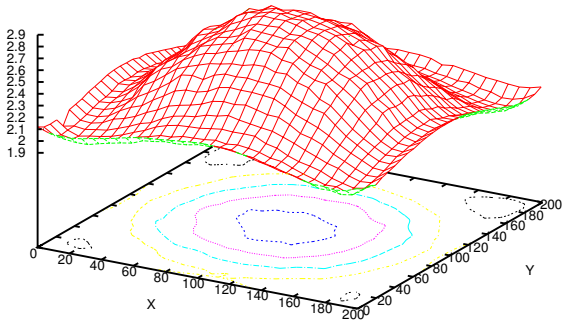
We gain insights into the reasons behind the slow infection propagation for TLW by using two metrics: (i) contact rate, and (ii) spatial distributions of node mobility.

1. Contact Rate: Contact rate is the average number of nodes encountered by any given node over the duration of simulation. We plot an empirical cumulative

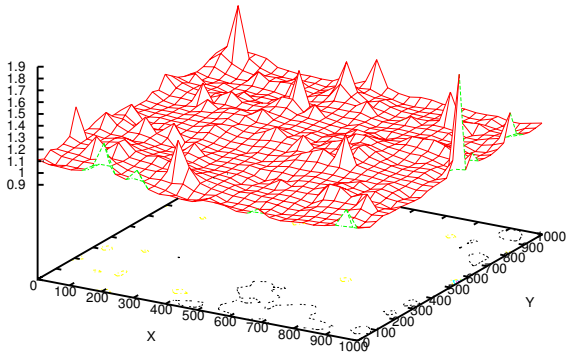
distribution curve (ECDF) of the contact rate in Fig. 4.3(b) for RWP and TLW. Observe that the median contact rate of nodes in case of RWP is almost always higher than that in TLW. The same effect can be observed for the 95^{th} percentile indicating that in RWP, a given node comes in contact with a relatively higher number of nodes thereby increasing its chances of infecting other nodes or getting infected by other infected nodes.

2. Spatial Distributions: The spatial distribution (*i.e.*, frequency of visits in the simulation area) of the mobility models reveals another reason behind the slow infection propagation. In order to evaluate the spatial distribution of infected nodes that move according to each of the models, we take an approach similar to [159]. Specifically, we divide the simulation area into small size cells (e.g., divide a 1000x1000m^2 into 20x20m^2 size cells) and characterize each one of them using a histogram that captures the duration of how long an infected node stays in a particular cell. We end the simulation after 50,000 seconds.

Fig. 4.4(a) shows the resulting spatial distribution and contour lines for a particular simulation run using RWP. We observe that the spatial distribution has a peak in the middle of the area, i.e., an infected node is most likely to be found in the central cells of the simulation area and the probability that a node is located at the border of the area goes to zero. Fig. 4.4(b) shows the spatial distribution and contour lines for TLW. Observe that the non-homogeneous behavior seen in the case of RWP is absent in the case of TLW, i.e., TLW exhibits a homogeneous spatial distribution. The reason for the non-homogeneous behavior in RWP is well known [159–161]. In short, RWP chooses a uniformly distributed destination point rather than a uniformly distributed angle. This means that nodes located at the border of the simulation area are very likely to move back toward the middle of the area. However, this is not the case as per the original definition [143] of TLW. Under a TLW, at the beginning of



(a)



(b)

Figure 4.4.: Spatial distribution of mobility models: (a) RWP: The non-homogeneous distribution of node mobility indicates the center to be the most frequented region. (b) TLW: The nearly-homogeneous distribution of node mobility indicates that all regions are equally frequented.

each step, an infected node chooses a direction randomly from a uniform distribution of angle within $[0, 2\pi]$, a finite flight time randomly based on some distribution, and its flight length and pause time from some chosen probability distributions. In the long run, the positions of the random walker (infected node in our case) has been shown to converge to another distribution, called the *Levy stable distribution*, which

leads to super-diffusive paths, thus making the infected nodes cover the area in a nearly homogeneous manner.

In summary, in case of RWP, depending on the origin of the infection, the spread can progress rapidly because most nodes have to pass through a common point in the center which also explains why the contact rate of the nodes is higher than that in TLW. In case of TLW, due to the underlying homogeneous behavior, the rate of infection propagation is nearly the same irrespective of the point of origin of the infection.

Impact on the design of countermeasures:

- **Static healers placement**: In case of RWP, positioning a few static healers somewhere near the center of the field in a non-overlapping manner should suffice because most nodes will traverse the central point in the field anyways. However, this is not the case for TLW, because the node distribution is uniform across the field, thus requiring a way to optimize healer placement such that they cover as much field as possible.
- **Healer patch dissemination**: In case of designing a healer for TLW, having a higher patch dissemination rate will result in a lot of patches being delivered to the same set of nodes since due to the low velocity (and thus low contact rate) many nodes may continue to stay within the proximity of the healer. Therefore, for a system optimizing energy, healer patch dissemination is a function of the contact rate (details in § 4.3.5).

4.3 Defense Protocols Based on Static Healers

In this section, we discuss defense protocols against mobile malware. We first present the problem definition, formally define the static optimal healer activation problem, and design a greedy approximation algorithm. We discuss strategies for

healer placement and present three families of static healers heuristics: *randomized*, *profile*, and *prediction*.

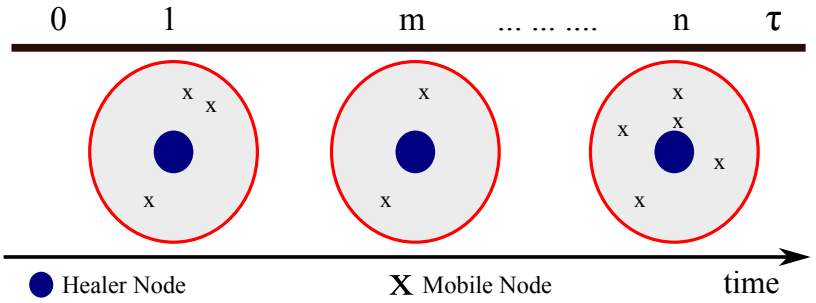


Figure 4.5.: Healer activation problem: Without information about its future states, predict when to broadcast a patch so that the healing time is optimized

4.3.1 Problem Definition

Healers have the ability to broadcast a patch periodically at every epoch τ. We consider the decision problem of when the healer node should be activated (i.e. switched on) within this time period to deliver a patch, to optimize along two dimensions: (i) the time it takes to heal the entire system, and (ii) the total number of patches broadcast.

We assume that healers are not susceptible to the infection. In addition to this, we also assume that healers can sense the number of neighbors surrounding them but cannot determine which of the nodes are infected/susceptible/recovered[1]. Note that this increases the complexity of the problem significantly. Consider the example in Fig. 4.5. At time slot 1, if the healer decides to utilize its patch, it will heal at most three nodes whereas at time slot m, it can heal at most two nodes and at time slot n, it can heal at most five nodes. An oracle that has access to the future will pick a

[1]Identifying the state of a node based on the interaction with the node is quite similar to the problem of detecting rootkits using the intrusion detection systems (IDSs) that rely on the system itself. In fact, when a system is compromised by rootkits, IDSs must not rely on the system [162]. Hence our healers treat each node equally.

time slot that will make an effective use of the patch to heal the maximum number of nodes (in this case, time slot n). However, in practice, the future is not available to healer nodes.

We ask the questions: *What is an effective strategy for positioning the static healers so that two healers will avoid healing the same set of infected nodes?* and *How does the healer decide whether it should deliver the patch or wait in anticipation of a higher number of nodes in the future?* Without loss of generality, we consider that the energy consumption in delivering the patch is much higher than any other communication activity initiated by a healer. Intuitively, we are solving the problem of effectively distributing a patch without knowing the arrival distribution of infected nodes.

4.3.2 Design of an Oracle Optimal Healer

In the following, we formally define the static optimal healer activation problem, and design a greedy approximation algorithm instead.

Let us call the task of designing a strategy for an optimal healer as the T-Cover problem.

Definition 1 (T-Cover *Problem*). *Let $\mathcal{I}$ be the set of all infected nodes in the network and $\mathcal{T} = \bigcup_i T_i$, where each T_i is the set of infected nodes seen by all the healers at time instance i. Furthermore, let no two healers exist within the range of each other, and a patch from a healer can heal all infected nodes within its range and will consume one time unit. The* T-Cover *problem of $I_t = (\mathcal{I}, \mathcal{T})$ is to find a set $W \subseteq \mathcal{T}$ such that it covers the entire set of infected nodes $\mathcal{I}$ (i.e. $\bigcup_{T_i \in W} T_i = \mathcal{I}$) and W has minimum cardinality.*

Here, the cost c_i associated with T_i is equivalent to the time instance value, i.e., i. Minimizing the total cost $\sum_{i \in W} c_i$ is equivalent to minimizing both the total time to recover the infected nodes and the required number of patches to do so. For example, let $\mathcal{I} = \{1, 2, 3, 4\}$ and $\mathcal{T} = \{T_1, T_2, T_3\}$ where $T_1 = \{1\}$, $T_2 = \{1, 2, 3\}$ and $T_3 = \{3, 4\}$ be the sets of infected nodes seen by the healer, then the T-Cover is $W = \{2, 3\}$ meaning that a patch should be deployed at time $t = 2$ and $t = 3$ for optimality. We can restate this optimization problem as a decision problem.

Definition 2 (T-Cover *Decision Problem*). *Given a system $I_t = (\mathcal{I}, \mathcal{T})$, the* T-Cover *decision problem is to determine whether I_t has a* T-Cover *of size at most k.*

In other words, we wish to determine whether there is a set $W \subseteq \mathcal{T}$ such that $|W| \leq k$ and $\bigcup_{T_i \in W} T_i = \mathcal{I}$. In essence, T-Cover problem is the same as the min set cover problem, which we define below for completeness.

Definition 3 (Min Set Cover (*MSC*) *Problem*). *Let $\mathcal{S} = \{S_1, S_2, ..., S_m\}$ be a collection of finite sets, where S_i's elements are drawn from a universal set $U = \bigcup_{i=1}^{m} S_i$. The MSC of $I_s = (U, \mathcal{S})$ is a set $C \subseteq \mathcal{S}$ such that $\bigcup_{S_i \in C} S_i = U$ and C has minimum cardinality.*

For example, assume $U = \{1, 2, 3, 4, 5\}$ and $\mathcal{S} = \{S_1, S_2, S_3, S_4\}$, where $S_1 = \{1, 2, 3\}$, $S_2 = \{2, 4\}$, $S_3 = \{3, 4\}$ and $S_4 = \{4, 5\}$. The minimum set cover is $C = \{S_1, S_4\}$. Similarly, we can restate this optimization problem as a decision problem.

Definition 4 (Min Set Cover *(MSC) Decision Problem*). *Given $I_s = (U, \mathcal{S})$, the MSC decision problem is to determine whether I_s have a set cover of size at most k.*

In other words, we wish to determine whether there is a set $C \subseteq \mathcal{S}$ such that $|C| \leq k$ and $U = \bigcup_{S_i \in C} S_i$. As the MSC decision problem is NP-Complete, it follows that the T-Cover decision problem is NP-Complete.

Algorithm 2: Greedy Approximation (ORACLE)

Input Let $\mathcal{I}$ be the list of all infected nodes, S_i be the set of infected nodes seen at each time i, w_i be the list of costs associated with each arrival at i

Initially:

R is the set of elements that are not covered as yet
C is the set of covered elements
w is the weight vector
$R = \mathcal{I}$ and $C = \phi$

repeat

let S_i be the set that minimizes $\frac{w_i}{|S_i \cap R|}$
$C = C \cup \{S_i\}$
$R = R - S_i$

until $R = \phi$

return C

According to the above theorem, we can employ any heuristic that solves the set cover problem to solve the T-Cover problem. Algorithm 2, based on the greedy set cover algorithm [163], gives a greedy approximation for the T-Cover. The algorithm takes as input the arrival times of the infected nodes. Here, S_i is the set of infected nodes seen at any one time instant and we equate the weight vector w_i to the time of arrival – cost of healing nodes at a later time is higher because it introduces delay. The main loop iterates for $O(n)$ time, where $|\mathcal{I}| = n$. The minimum W can be found in $O(log\ m)$ time, using a priority heap, where there are m sets in a set cover instance giving us a total time of $O(nlog(m))$. Fig. 4.6 shows that even in the presence of hundreds of thousands of node sets, we are able to compute the optimal solution in under 8 seconds.

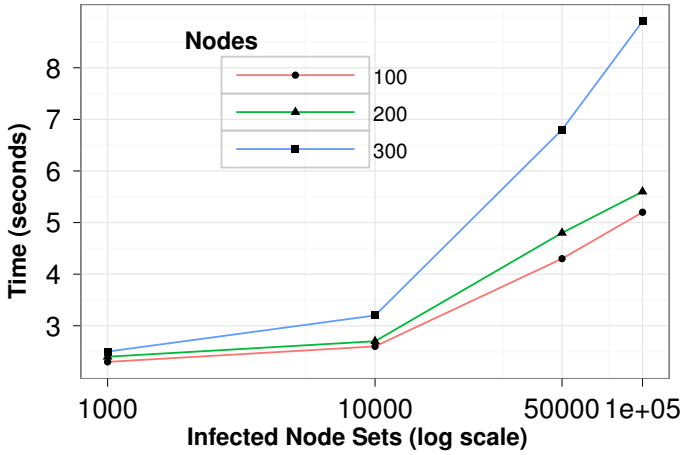


Figure 4.6.: Oracle Performance (The algorithm terminates in less than 8 seconds even in long simulation scenarios)

4.3.3 Effective Healer Placement

Since the healers are static, the healer placement has an impact on our defense protocols and thus their coverage area depends on their placement strategy. Our simulations showed that a naive placement using uniform random distribution resulted in a scenario where many healers ended up covering the same region thereby leaving a lot of uncovered area. Another naive approach is the grid placement of healers in which healers cover the entire arena and therefore each mobile node will always be in the range of at least one healer. This approach would require N number of healers to cover the entire arena which could be a very large number depending on the size of arena and the range of healers[2,3]. Note that the infection containment problem becomes trivial in case of grid placement. For instance, if healers were placed in grids, the defense protocol would require all the healers to broadcast one patch each at the same time instance t and thus, the entire system would be recovered in one second at

[2]For an arena of 500×500 (meter)^2 and 20 meter healer-range, N would be at least 157, whereas we used $N = 20$ healers for the same setup.
[3]N would no longer be a fixed number

the cost of N patches. However, in realistic environments, it is not practical to have so many static healers. We focus instead of scenarios using a much smaller number of static healers.

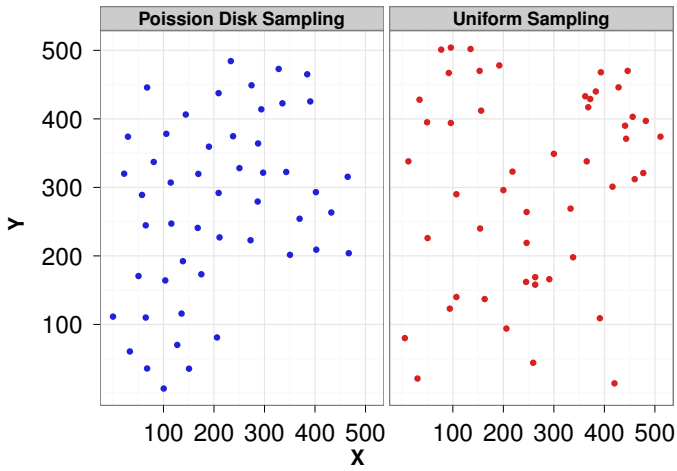


Figure 4.7.: Healer placement using Poisson Disk Sampling and Uniform Sampling (Poisson Disk Sampling approach increases the coverage of the simulation area)

For our healer placement strategy, what we need is a type of a constraint that rejects certain configurations that place healers very close to each other. This problem can be directly reduced to a problem from the field of computer vision which involves producing sampling patterns with a blue noise Fourier spectrum. Formally, the problem can be defined as the limit of a uniform sampling process with a minimum-distance rejection criterion. Successive points are independently drawn from the uniform distribution $[0, 1]$. If a point is at a distance of at least R from all points in the set of accepted points, it is added to that set. Otherwise, it is rejected. The choice of R controls the minimum allowable distance between points. This procedure called *Poisson Disk Sampling* [164] has been actively studied and many efficient algorithms exist. We adapted this algorithm by setting $R = 2r$, where r is the range of our each healer. Fig. 4.7 clearly highlights the merits of using this specific sampling process - healers are no longer close to each other and hence cover more of the simulation area.

4.3.4 Family of Randomized Healers

We first present a heuristic where a healer randomly decides at what time within an epoch to send a patch. Note that a healer will decide to send a patch regardless of the number of nodes in its vicinity. Fig. 4.8 depicts the state machine of the randomized healer (RH). It contains two states, an *initialization phase* where an *epoch timer* is started and an *execution phase* where the healer prepares to deliver a patch. The *epoch timer* fires a callback function that has two responsabilities: (i) pick a random time from the interval $[0, \tau]$, where τ is the *epoch length*, and use this random time to schedule a broadcast, called the *patch timer* and (ii) re-schedule the *epoch timer* to be fired for the next epoch. τ depends on the range of the healer and velocity of the mobile node. When the *patch timer* expires, the healer broadcasts a patch with a probability p, we call it the *patch deployment probability*.

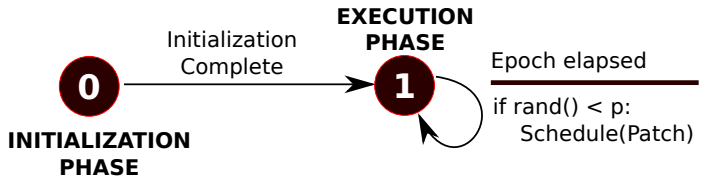


Figure 4.8.: State machine of a Randomized Healer

Algorithm 3 outlines the pseudo-code for the randomized healer. Varying p will generate a family of randomized healers. On one hand, setting $p = 1$ ($RH_{(p=1)}$) makes the healer broadcast a patch at every epoch and thus attempts to minimize the time it takes to heal the system. However, notice that the number of patches delivered would be equal to $\frac{D_{sim}}{\tau}$, where D_{sim} is the simulation duration. On the other hand, setting $p < 1$ makes the healer broadcast a patch only during certain epochs. The time taken to heal the system is inversely proportional to p whereas the number of patches delivered is directly proportional to it.

Algorithm 3: Randomized Healers (RH)

Input Epoch length τ and patch deployment probability p
Initially:
start $epoch\_timer(\tau)$
Upon the expiration of $epoch\_timer$:
select a duration t randomly from $(0, \tau)$
start $patch\_timer(t)$
start $epoch\_timer(\tau)$

Upon the expiration of $patch\_timer$:
Broadcast a patch with probability p

4.3.5 Family of Profile Healers

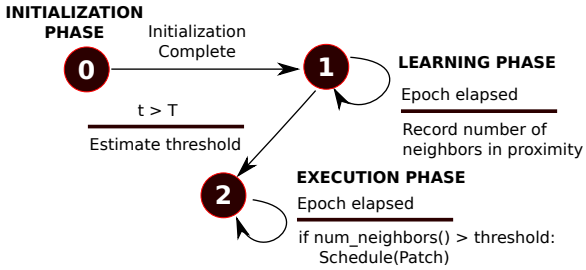


Figure 4.9.: State machine of a Profile Healer

One limitation of the RH approach is that healers may send more patches than needed since they decide to send patches regardless of how many infected nodes are present in their proximity. We propose a new approach, PH, where a healer attempts to learn the arrival distribution of nodes and subsequently determine whether or not it is cost effective to deliver a patch. The decision is made based on a threshold that captures the number of nodes in its vicinity.

Each healer can exist in one of three states as depicted in Fig. 4.9 - an *initialization phase* which prepares the healer, a *learning phase* where the healer passively records the number of neighbors it is observing during each epoch (in general), and an *exe-*

cution phase where the healer utilizes information that it learnt during the previous phase to decide whether or not to deliver a patch. Algorithm 4 describes this healer (PH) in detail. In the initialization phase, each healer sets its own state to `LEARNING` and starts the sensing timer (*sensing_timer*) with a duration of 1 second. Upon the expiration of *sensing_timer*, the healer checks whether the observation time T has elapsed yet. If not, the healer records the number of neighbors ($num\_of\_neighbors$) in its proximity and restarts *sensing_timer*. When the observation time T has elapsed, the healer first estimates the *threshold* from the recorded information and moves to the `EXECUTION` state. Now at every second, the healer checks whether the number of neighboring nodes exceeds the *threshold*. If so, the healer deploys a patch and starts a timer (*epoch_timer*), which expires at the end of the current epoch. Until then, the healer does no sensing at all. Upon the expiration of *epoch_timer*, the healer starts sensing again by setting *sensing_timer*.

The goal of *learning phase* is to learn the distribution of node arrivals specific to a healer's locality for a certain *observation time* T which is a multiple of τ. Specifically, the goal is to learn a threshold of nodes that will determine whether the healer should send a patch or not. We use two metrics described below.

- $MSD = Mean + 1.5 \times Standard\ Deviation$: MSD is well-known for normal distributions and makes the healer broadcast a patch only if the number of neighbors exceeds its estimate of the 95^{th} percentile.
- $M = Median$: M is the median of the observed distribution. Median is very robust to outliers – it handles cases where a healer observes a burst of infected nodes during an epoch.

During our simulations, we observed that relying solely on a *threshold* was leading to a wastage of patches - due to the low contact rate we observed in §4.2.2. Consider Fig. 4.10 which depicts the healing sequence of a set of five healers during the epochs

Algorithm 4: Profile healers (PH)

Input Epoch length τ, observation time T such that $T > 1$

Initially:
  $t \leftarrow 0$, $\Delta \leftarrow 1$, *state* $\leftarrow$ `LEARNING`, *next_epoch_time* $\leftarrow 0$
  Start *sensing_timer*(Δ) ▷ Start timer with duration Δ
Upon the expiration of *sensing_timer*:
  $t \leftarrow t + \Delta$
  if *state* = `LEARNING` **then**
    if $t < T$ **then**
      Record *num_of_neighbors* in proximity
    else
      Estimate *threshold* from the recorded *num_of_neighbors* at each Δ
      state $\leftarrow$ `EXECUTION`
      next_epoch_time $\leftarrow t + \tau$
    end if
    Start *sensing_timer*(Δ)
  else
    if current *num_of_neighbors* > *threshold* **then**
      Broadcast a patch
      Start *epoch_timer*(*next_epoch_time* $- t$)
    else
      Start *sensing_timer*(Δ)
    end if
  end if

Upon the expiration of *epoch_timer*:
  $t \leftarrow$ *next_epoch_time*
  next_epoch_time $\leftarrow t + \tau$
  Start *sensing_timer*(Δ)

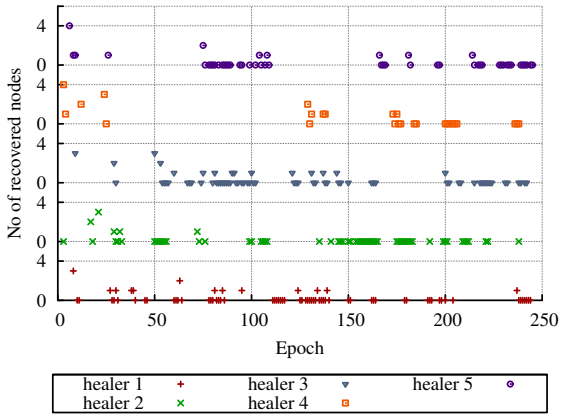


Figure 4.10.: Motivating backoff: Most consecutive patches do not heal infected nodes indicating that it is better to backoff after a patch delivery

of one simulation run. Points situated at 0 indicate that the healer deployed a patch as the number of neighbors was above the threshold but the patch *did not* heal any infected nodes. Any other number indicates the number of infected nodes healed with that patch. Observe that most patches are going to waste, *i.e.*, they are not healing any nodes. In the worst case, it takes at least $\frac{healer\ range}{node\ velocity}$ seconds for a node to go out of range of a healer. Therefore, for shorter epochs, consecutive patches are delivered to the same set of nodes. We address this issue by introducing a *random backoff*, i.e., once a patch has been broadcast, the healer selects a random backoff delay κ from the interval $(0, \eta)$, where η is the maximum backoff in epochs, and skips that many epochs. Algorithm 5 also describes the backoff algorithm in detail. We refer to this algorithm as PHB. This algorithm is similar to Algorithm 4 except that each healer now selects a random backoff delay $\kappa \in \{1, \ldots, \eta - 1\}$ to remain silent after the deploy of a patch. Upon the expiration of this remaining period, the healer starts sensing again.

Both PH and PHB have two shortcomings. First, both require to wait until the end of the learning phase (i.e., a certain observation time T to learn the distribution of

Algorithm 5: Profile healers with backoff (PHB)

Input Epoch time τ, observation time T such that $T > 1$ and maximum backoff η such that $\eta > 1$

Initially:
  $t \leftarrow 0$, $\Delta \leftarrow 1$, $state \leftarrow$ `LEARNING`, $next\_epoch\_time \leftarrow 0$
  Start $sensing\_timer(\Delta)$ ▷ Start timer with duration Δ

Upon the expiration of $sensing\_timer$:
  $t \leftarrow t + \Delta$
  if $state =$ `LEARNING` **then**
    if $t < T$ **then**
      Record $num\_of\_neighbors$ in proximity
    else
      Estimate $threshold$ from the recorded $num\_of\_neighbors$ at each Δ
      $state \leftarrow$ `EXECUTION`
      $next\_epoch\_time \leftarrow t + \tau$
    end if
    Start $sensing\_timer(\Delta)$
  else
    if current $num\_of\_neighbors > threshold$ **then**
      Broadcast a patch
      Randomly select κ between $(0, \eta)$
      Start $epoch\_timer(next\_epoch\_time - t + \kappa \times \tau)$
    else
      Start $sensing\_timer(\Delta)$
    end if
  end if

Upon the expiration of $epoch\_timer$:
  $t \leftarrow get\_current\_time()$
  $next\_epoch\_time \leftarrow t + \tau$
  Start $sensing\_timer(\Delta)$

node arrivals) to start healing the system. Second, the healers learn and estimate the threshold only once. This may not characterize the node arrival distribution of the system accurately. Note that any attempts to improve one shortcoming will worsen the other. For instance, on one hand, decreasing the observation duration T, to start healing early, introduces the possibility of inaccurately estimating the threshold (due to insufficient data points) and hence leads to consuming more patches. On the other hand, if T were to be increased (to better capture the node arrival distribution), it results in an increased system recovery time. To address these limitations, we adopt a hybrid approach where healers perform online learning and heal the system simultaneously. This approach is an extension of the PHB algorithm where each healer never stops learning. Moreover, at the end of every Γ epochs, each healer dynamically estimates a new decision threshold based on what it has learned in the last Γ epochs and uses the newly estimated threshold for the next Γ epochs. We refer to this algorithm as D-PHB (see Algorithm 6). Note that, unlike both PH and PHB, each healer can be either in `LEARN_EXEC` state when it both learns and heals or in `ONLY_LEARN` state when it only learns. However, each D-PHB healer uses random backoff mechanism like PHB healers.

4.3.6 Family of Prediction Healers

The optimal healer ORACLE (see Algorithm 2) has several advantages compared to the profile healers. Firstly, an optimal healer has the global view of the entire network, whereas a profile healer has only the local view (neighbors at its vicinity). Secondly, an optimal healer can explicitly identify the state (e.g., susceptible, infected, recovered) of every mobile node, but a profile healer is not capable of identifying the state of a mobile node in its proximity. Finally, while the former knows the future (i.e., time instance at which each healer is going to observe the maximum number

Algorithm 6: Profile healers with backoff and dynamic threshold scheme (D-PHB)

Input Epoch time τ, observation epochs Γ such that $\Gamma > 1$, initial threshold α, and maximum backoff η such that $\eta > 1$

Initially:

$t \leftarrow 0$, $\Delta \leftarrow 1$, $next\_epoch\_time \leftarrow \tau$, $state \leftarrow$ `LEAR_EXEC`

$threshold \leftarrow \alpha, epoch\_count \leftarrow 0, time\_to\_switch\_state \leftarrow 0$

Start $sensing\_timer(\Delta)$ ▷ Start timer with duration Δ

Upon the expiration of $sensing\_timer$:

$t \leftarrow t + \Delta$

Record $num\_of\_neighbors$ in proximity into Σ

if $state =$ `LEAR_EXEC` **then**

if current $num\_of\_neighbors > threshold$ **then**

Broadcast a patch

Randomly select κ between $(0, \eta)$

$time\_to\_switch\_state \leftarrow next\_epoch\_time - t + \kappa \times \tau$

$state \leftarrow$ `ONLY_LEARN`

end if

end if

if $t = next\_epoch\_time$ **then**

$epoch\_count \leftarrow epoch\_count + 1$

if $epoch\_count = \Gamma$ **then**

Estimate $threshold$ from the recorded $num\_of\_neighbors$ at each Δ

Clear records from Σ

$epoch\_count \leftarrow 0$

end if

$next\_epoch\_time \leftarrow next\_epoch\_time + \tau$

end if

if $t = time\_to\_switch\_state$ **then**

$state \leftarrow$ `LEAR_EXEC`

end if

Start $sensing\_timer(\Delta)$

of infected nodes), the latter has no such knowledge. All these advantages make the optimal healers ideal, but impractical. Having the first two capabilities of an optimal healer would make any healer impractical for real world. However, in case of the third capability, we can equip a healer with the ability to predict the event of observing relatively higher number of mobile nodes[4] in the near future with the goal of hitting a middle ground between the optimal healers and the profile healers. Therefore, we propose a new family of healers called *prediction healers* (PDH).

Similar to a profile healer, each prediction healer can exist in one of the three states as shown in Fig. 4.9, except it does not estimate a threshold. Instead, each healer now computes a *stationary transition probability matrix* [5]. Let X_t^h be the state, i.e., the total number of mobile nodes observed by the hth healer at time instance t. Further, assume that the stationary transition probability matrix for the healer h is $\mathcal{P}^h = [p_{ij}^h]_{n\times n}$ where $p_{ij}^h = \mathsf{Pr}[X_{t+1}^h = j | X_t^h = i]$, i.e., the probability of observing j nodes in the next time instance given that the healer has seen i nodes at the current time instance and n be the total number of mobile nodes in the system[6]. Each healer must deploy a patch during every epoch, but it is free to choose the deployment time instance within an epoch. This deployment time instance is chosen based on whether it is worth deploying the patch now or to hold off for a better future state that may be observed within this epoch. If the healer reaches the deadline of the current epoch and has not deployed the patch yet, it must deploy the patch right away. Each prediction healer uses a prediction function $\mathcal{F}$ which is based on this intuition. We define the function $\mathcal{F}$, formally, as follows:

$$\mathcal{F}(\lambda, x|\mathcal{P}) = \begin{cases} 1 & \text{if } \mathcal{G}(\lambda, x|\mathcal{P}) > \sum_y p_{xy}\mathcal{G}(\lambda-1, y|\mathcal{P}) \\ 0 & \text{otherwise} \end{cases}$$

[4]Mobile nodes in general, not only infected ones
[5]Similar to the transition probability matrix of a Markov Model
[6]Superscript means the identity of the healer, not the h-step transition probabilities of Markov chain.

where, λ is the remaining time to the deadline of the current epoch, x is the current state of the healer, y is any possible next state and $y \in [0, n]$, $\mathcal{P}$ is the transition probability matrix of the healer, and

$$\mathcal{G}(\lambda, x|\mathcal{P}) = \begin{cases} 0 & \text{if } \lambda < 0 \\ x & \text{if } \lambda = 0 \\ \max\{x, \sum_y p_{xy}\mathcal{G}(\lambda - 1, y|\mathcal{P})\} & \text{otherwise} \end{cases}$$

Note that $\mathcal{G}(\lambda, x|\mathcal{P}) \geq \mathcal{G}(\lambda-1, x|\mathcal{P})$, for all λ, x. That is, the worth of the patch either stays the same or diminishes with the decrease in λ (equivalently, with the increase in time). In other words, the more a healer waits to deploy a patch, the more the patch loses its worth. Note that it captures the time constraint of the T-Cover problem. Fig. 4.11 shows the internals of the prediction function where x is the current state of the healer. In the next time instance, the healer can move to any state $j \in [0, n]$ with probability p_{xj}. The healer predicts the future and decides on whether or not to deploy a patch using $\mathcal{F}(\lambda, x|\mathcal{P})$ at the current state x.

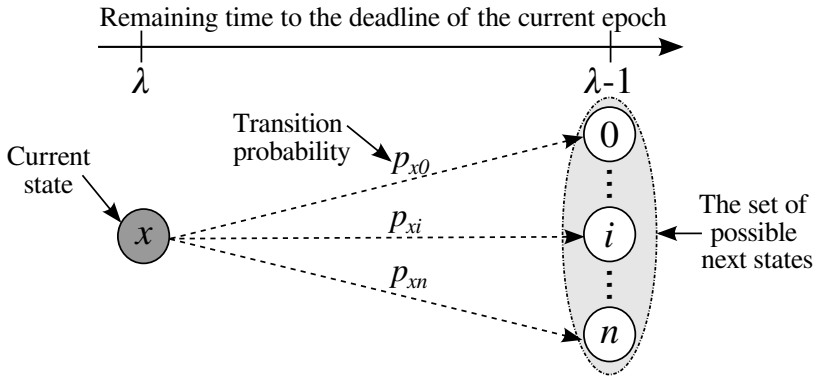


Figure 4.11.: The internals of the prediction function used by a prediction healer

Algorithm 7 describes the healer in detail. During the `LEARNING` state, each healer records the number of neighbors observed at each time instance. After the observation period, each healer computes the transition probability matrix $\mathcal{P}$ and stores it for

Algorithm 7: Prediction Healers (PDH)

Input Epoch length τ, observation time T such that $T > 1$

Initially:
  $t \leftarrow 0$, $\Delta \leftarrow 1$, *state* $\leftarrow$ `LEARNING`, *deploy_status* $\leftarrow$**false**
  $\lambda \leftarrow 0$ ▷ λ is the time to be elapsed until the deadline of the current epoch
  Start *sensing_timer*(Δ) ▷ Start timer with duration Δ

Upon the expiration of *sensing_timer*:
  $t \leftarrow t + \Delta$
  if *state* = `LEARNING` **then**
    if $t < T$ **then**
      Record *num_of_neighbors* in proximity and store in $\mathcal{S}$
    else
      Compute $\mathcal{P}$ from the recorded $\mathcal{S}$ ▷ $\mathcal{P}$ is the transition probability matrix
      state $\leftarrow$ `EXECUTION`
      $\lambda \leftarrow \tau$
    end if
  else if *state* = `EXECUTION` **then**
    $\lambda \leftarrow \lambda - 1$ ▷ Elapsed one second
    $x \leftarrow$ current *num_of_neighbors* in promixity
    if *deploy_status* = **false then**
      if $\lambda = 0$ **or** $\mathcal{F}(\lambda, x|\mathcal{P}) = 1$ **then**
        Broadcast a patch
        deploy_status $\leftarrow$ **true**
      end if
    end if
    if $\lambda = 0$ **then**
      $\lambda \leftarrow \tau$
      deploy_status $\leftarrow$ **false**
    end if
  end if
  Start *sensing_timer*(Δ)

future reference. In the `EXECUTION` state, each healer decides to deploy the patch if (1) the healer has reached the deadline of the current epoch or (2) it is worth deploying at the current time instance based on $\mathcal{F}(\lambda, x|\mathcal{P})$. Whenever λ becomes zero, it performs some reinitialization to prepare itself for the next epoch.

Computing $\mathcal{F}(\lambda, x|\mathcal{P})$ requires a healer to compute the $\mathcal{G}(.)$ recursively from λ to 0. Recursive implementations of $\mathcal{F}(.)$ and $\mathcal{G}(.)$ are highly expensive when the system contains hundreds of nodes and the epoch length is in the order of minutes. In addition, recursive implementation wastes computations by solving the same subproblem multiple times. To overcome these challenges, we leverage *dynamic programming*, to efficiently implement $\mathcal{G}$ and $\mathcal{F}$. The intuition behind dynamic programming is to first solve the smaller subproblems and then utilize the answers to solve the overall problem. The pseudocode for $\mathcal{G}(.)$ is shown in Algorithm 8. A healer computes $\mathcal{F}(.)$ in a similar way.

4.4 Healer-Based Protocols Evaluation

In this section, we describe our evaluation methodology and present the performance of the various healer-based defense mechanisms outlined in §4.3. Table 4.1 summarizes the notations and the parameters of the healer algorithms that we evaluate in the section.

4.4.1 Evaluation Methodology

To evaluate the performance of the families of healers we proposed, we simulate the various healer-based protocols we described (see Table 4.1) using the NS-3 [104] network simulator on a network containing 300 nodes. We perform two different sets of experiments, one with nodes having RWP and the other with TLW as their mo-

Algorithm 8: Compute $\mathcal{G}(.)$

Input Total number of nodes n, epoch size τ, the transition probability matrix PT
Output A matrix GT of size $\tau \times n$

Declare a matrix GT of size $\tau \times n$
for $j \leftarrow 0$ **to** n
    $GT(0, j) \leftarrow j$
end for
for $i \leftarrow 1$ **to** τ
    for $j \leftarrow 0$ **to** n
        $sum \leftarrow 0$
        for $k \leftarrow 0$ **to** n
            $sum \leftarrow sum + PT(j, k) * GT(i - 1, k)$
            if $j > sum$ **then**
                $GT(i, j) \leftarrow j$
            else
                $GT(i, j) \leftarrow sum$
            end if
        end for
    end for
end for
return GT

bility model. We assume that the range of each healer is 20 meters and the *epoch length*, τ, is 30 seconds (so that each node stays within the range of a healer for one epoch length on an average before leaving the coverage area of the healer). In addition, 10% of the population is assumed to be initially infected to enable bootstrapping the system. We can technically start with one infected node (which was our initial attempt), but we observe that this only delays infection spread and increases the chance that infection will disappear. Healers are placed in the system using the strategy outlined in §4.3.3 and are activated (i.e., started) when the fraction of infected nodes exceeds 70% of the total population to give the system sufficient time to warm-up. We note that 70% is one possible worst case scenario and projects the capabilities of the healer. In real-world scenarios, this value depends on how fast one can setup healers during an epidemic outbreak. However, to analyze the sensitivity of these two parameters on the performance we conducted two-way analysis of variance (ANOVA)[7] tests with significance level of 0.05. We varied the number of initially infected nodes as 10%, 15%, and 20% and the time to activate healers when the 50%, 60%, and 70% of the population become infected. Both the parameters, either with or without the interaction between them, did not make any statistically significant impact on the performance. We also conducted pairwise comparison tests amongst the different categorical values of a parameter to see what significant differences are present amongst the values of the parameter. For both the parameters, the results indicate that there are no statistically significant pairwise differences between the categorical values. Therefore, we choose to proceed with 10% of the nodes as initially infected and to activate healers when 70% of the nodes becomes infected considering less aggressive nature in the both the cases.

[7]In statistics, a two-way ANOVA test is used to examine the influence of different categorical independent variables on one dependent variable [165].

Table 4.1.: List of protocols proposed and evaluated

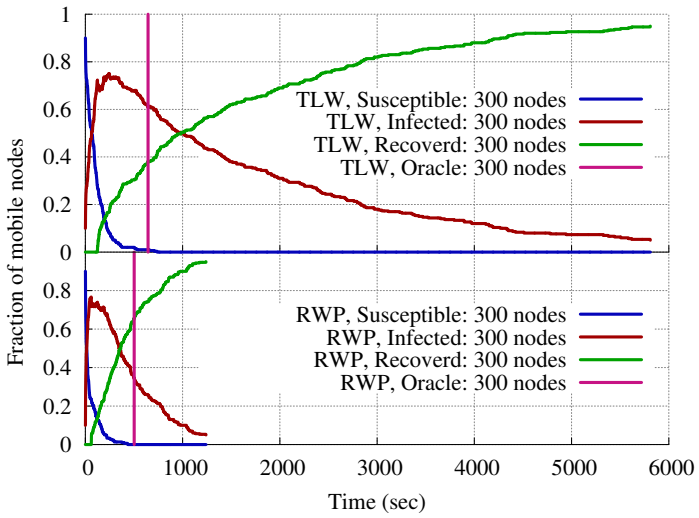
Protocol	Description	Parameters (default values)
RH_p	Randomized healers	Patch deployment probability p $(= 1)$
PH_{MSD}	Profile healers	Decision threshold $MSD = Mean + 1.5 \times stddev$
PH_M	Profile healers	Decision threshold $M = Median$
PHB_M	Profile healers with backoff	Maximum no. of epochs to backoff η $(= 2)$ and decision threshold M
D-PHB_M	Profile healers with backoff and dynamic threshold scheme	Observation epochs Γ $(= 10)$, maximum no. of epochs to backoff η $(= 2)$ and decision threshold M
PDH	Prediction healers	

Once the healers are activated, they follow the protocols outlined in §4.3.4, §4.3.5, and §4.3.6. All results are averaged across 10 runs of each experiment, to obtain statistically significant results, by varying the *seed* of a pseudo-random number generator. To measure the performance of each protocol, we define:

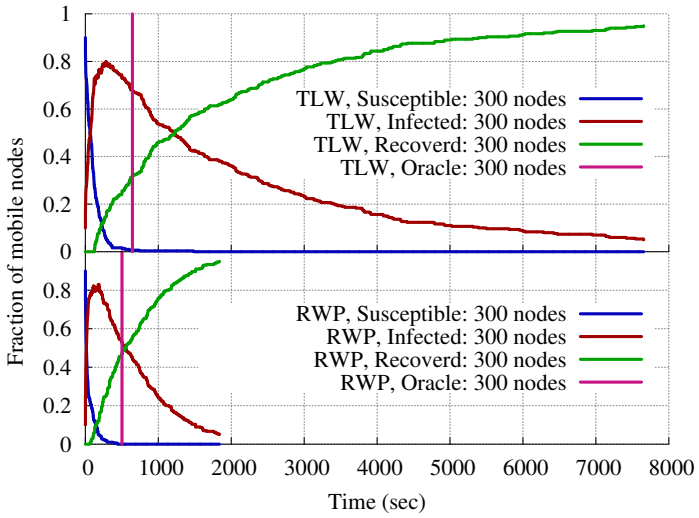
- *Total recovery time*: It represents the amount of simulation time required by the set of healers to recover at least 95% of the nodes in the system.

- *Total number of patches*: It represents the count of patches deployed by the set of healers to heal the system such that at least 95% of the total number of nodes are recovered.

Note that we choose 95% to account for scenarios similar to the *rare block problem* [166] in P2P networks - we observe the presence of some infected nodes that take exceedingly long time to enter the range of healers because they are wandering along the edge of the field and hence prolong our simulation.



(a) RH(p=1)



(b) RH(p=0.5)

Figure 4.12.: Evaluation of Randomized Healer family

4.4.2 Results for Family of Randomized Healers

Fig. 4.12(a) - 4.12(b) show the temporal view of infection propagation and the recovery of the system for RH family using RWP and TLW. The graphs show that regardless of the protocol, the required recovery time is always smaller in case of RWP than TLW which is due to RWP's higher contact rate.

Fig. 4.12(a) shows the required recovery time for randomized healers with $p = 1$, *i.e.*, $RH_{(p=1)}$. The upper graph is for TLW and the lower one is for RWP. Additionally, we also point out the recovery time required by ORACLE using a vertical line. In case of RWP, ORACLE requires 502 seconds to heal the system whereas $RH_{(p=1)}$ requires almost double this time, *i.e.*, 1,241 seconds. In case of TLW, ORACLE needs 645 seconds to heal the system whereas $RH_{(p=1)}$ requires about nine times the optimal recovery time. We also note that the recovery time required by $RH_{(p=1)}$ is the minimum time that we can achieve using healers that do not depend on system feedback (*e.g.*, estimating the arrival distribution of nodes).

Fig. 4.12(b) shows the results for $RH_{(p=0.5)}$, i.e., each healer deploys a patch per epoch with a probability $p = 0.5$. It is expected that $RH_{(p=0.5)}$ requires more time than $RH_{(p=1)}$ to heal the system since now the healers skip some epochs. In comparsion with the recovery time required by $RH_{(p=1)}$, $RH_{(p=0.5)}$ shows 48% increase in case of RWP and 31% increase in case of TLW.

4.4.3 Results for Family of Profile Healers

To measure the impact of different maximum backoff values on the PHB_{MSD} and the PHB_{M}, we varied the maximum backoff from 2 epochs to 16 epochs. Fig. 4.13 shows the results of this experiment. We also include the results of $RH_{(p=1)}$ as a baseline of the performance. We use two Y-Axes for this graph: the left one for the total number of patches and the right one for the total recovery time. Each point is the average of 10 different runs of the simulation and is plotted along its 95% confidence intervals. With the increase in maximum backoff values, the total recovery time is increasing rapidly in case of PHB_{MSD} in comparison with PHB_{M}. On the other hand, the total of number of patches is decreasing rapidly for PHB_{MSD} in comparison with PHB_{M}. We conjecture that if the recovery time is to be optimized, then PHB_{M}

is a better solution; but if the energy of the healers is to be optimized, then the PHB_{MSD} is a better choice. However, the downside of PHB is its large observation time. D-PHB is a solution to this downside of PHB. We also include the performance of D-PHB_M in Fig. 4.13. The results demonstrate that D-PHB_M performs as good as PHB_M in terms of both the metrics. So if the large observation time is unacceptable, D-PHB_M heals the system as fast as PHB_M and does not require any observation time.

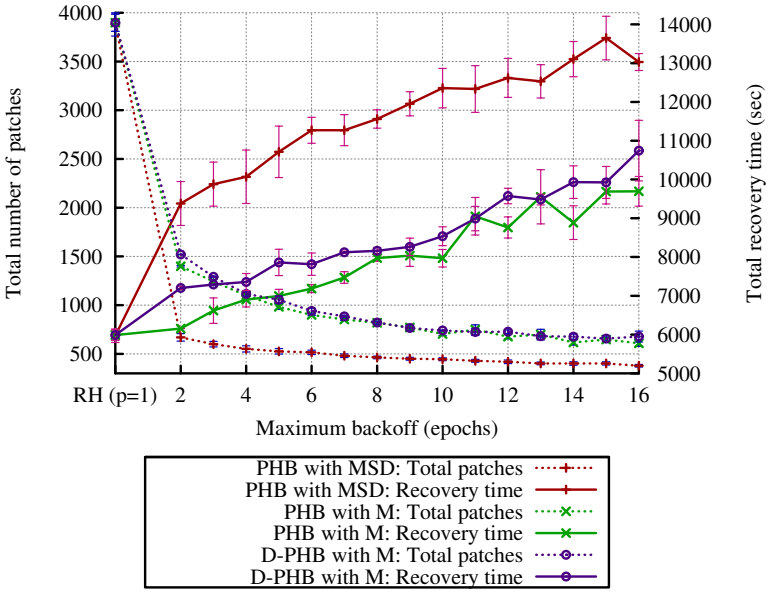


Figure 4.13.: Effect of varying maximum backoff (D-PHB_M performs as good as PHB_M in terms of both the metrics with the added advantage that it does not have an observation time)

Let X_{MSD} and X_M represent a profile-based healer X that utilizes MSD($= Mean + 1.5 \times Standard\ Deviation$) and M($= Median$) as its threshold, respectively. Fig. 4.14 shows the performance of PH for the RWP and the TLW mobility models. PH_{MSD} requires more time to heal the system in comparison with the other two RH healers. Since we are more interested in the human-mimicking mobility model, we evaluate PHB and D-PHB for only TLW in Fig. 4.15(a) and Fig. 4.15(b), respectively. Due

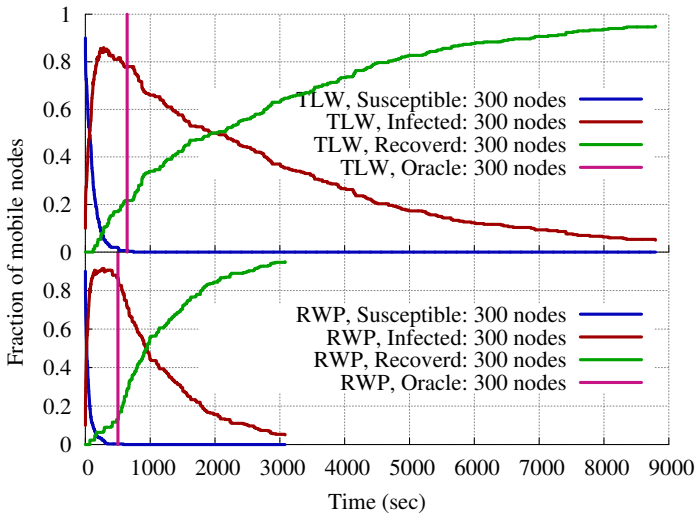
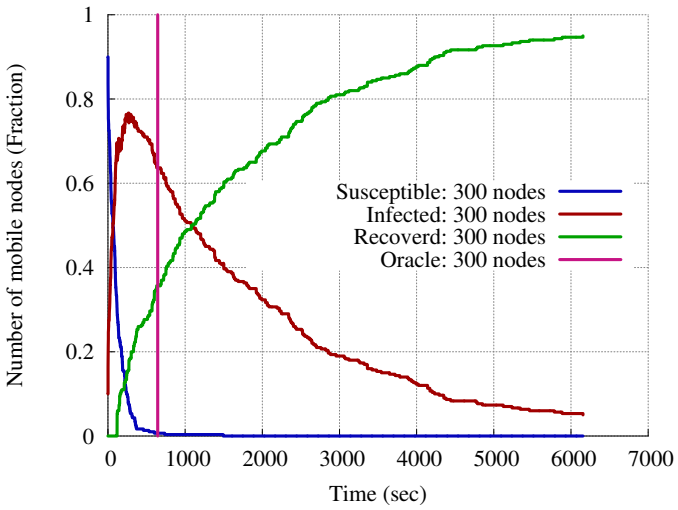


Figure 4.14.: Evaluation of PH_{MSD}

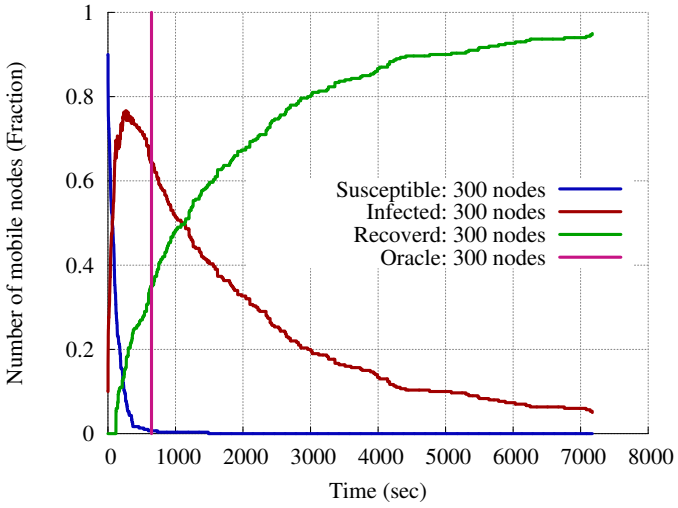
to space limitation, we present the performance of PHB and D-PHB with maximum backoff $\eta = 2$ and M as the threshold value in Fig. 4.15(a)- 4.15(b). When we compare PH_{MSD}, PHB_M, and D-PHB_M using the TLW mobility model, PHB_M outperforms the other two in terms of total recovery time.

4.4.4 Results for Family of Prediction Healers

Fig. 4.16 shows the temporal view of the infection propagation and recovery of the system using prediction healers in case of the TLW mobility model. The prediction healers require the least amount of time to recover the system when compared to the RH and PH families. This is due to the prediction capability of the healers that allow them to deploy patches efficiently. The recovery time required by the prediction healers is 18% less and 22.5% less than the best random healers $RH_{(p=1)}$ and the best profile healers PHB_M, respectively.



(a) PHB_M with $\eta = 2$



(b) D-PHB_M with $\eta = 2$

Figure 4.15.: Evaluation of various PH_M

Summary: Fig. 4.17 summarizes the results consisting of both the metrics obtained by each of the healers for the TLW mobility model. In terms of the number of patches, PH_{MSD} requires the least number of patches but at the cost of a larger recovery time. The prediction healers PDH outperforms the others in terms of the total recovery time. However, it requires 89% more patches than PH_{MSD}. Next comes the $RH_{(p=1)}$ that requires 21% more recovery time than PDH. However, in order to achieve this

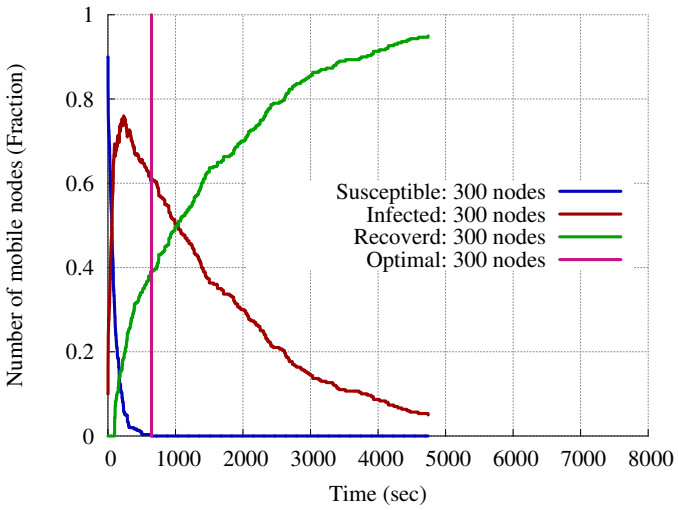


Figure 4.16.: Evaluation of PDH

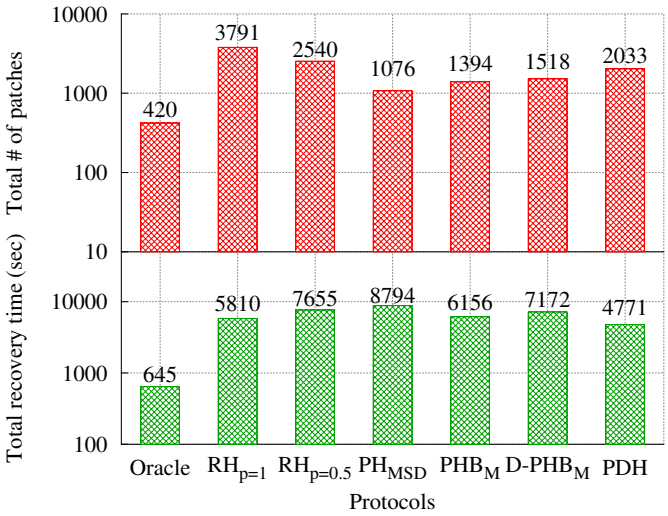


Figure 4.17.: Summary of the performances of healer families for TLW

recovery, $RH_{(p=1)}$ has to deploy the maximum amount of patches. In fact, PHB$_M$ performs the best since it requires only 29% more recovery time in comparison to PDH and only 30% more patches than that of PH$_{MSD}$.

Our results show that each of the schemes has advantages and disadvantages. First, randomized healers offer the immediate advantage that they do not rely on system feedback nor do they have to learn the system before starting to recover the

system. Second, prediction healers would be beneficial in a time-constrained system as these healers are fastest at recovering an infected system by utilizing prediction capability. However, they result in using 1.8x patches than the profile healers (with MSD threshold). Finally, profile healers with backoff offer intelligent decision making thereby saving energy in the form of utilizing less number of patches and would benefit the most in an energy-constrained environment. However, they result in taking 1.8x time to recover in comparison to PDH healers. On the other hand, when compared with the ORACLE, we observe that PDH, RH, and PHB_M healers take 7.4x, 9x, and 9.5x recovery time, respectively. Furthermore, to recover the system PH_{MSD} healers require 2.5x patches than the ORACLE.

4.5 Summary

Mobile malware have become an emerging problem that threatens smartphones which are growing significantly in recent days. In this work, we considered realistic mobility patterns to model proximity dependent malware and compared them against de facto models like random waypoint mobility model. We presented several defense mechanisms that allow tuning of parameters to control the optimization along two dimensions: time to recovery and energy utilized. The extensive evaluation of all our defense mechanisms shows that prediction healers would be more effective in a time constrained environment whereas profile healers would benefit the most in an energy constrained environment.

5 RELATED WORK

The related work for Chapters 2, 3, and 4 is discussed here. For clarity, we present the related work on compliance checking, adversarial testing, and infection mitigation in Section 5.1, 5.2, and 5.3, respectively.

5.1 Compliance Checking

We now outline the previous work that are closely related to our compliance checking work by grouping them into different categories.

Software model checking Formally proving that a program satisfies some properties has been considered an important problem [167, 168]. Software model checking can be roughly divided into the following categories: execution-based [13, 169, 170] and abstraction-based techniques [14, 171]. Counterexample-guided abstraction refinement (CEGAR) [55] exploits the best of the above two approaches by automatically generating the abstractions of the program under verification and refining when a spurious counterexample is encountered. CEGAR ensures that the abstract state space of the program is small enough to be searched efficiently. Many automatic tools that use some form of CEGAR, *e.g.*, SLAM [16], BLAST [17], CPACHECKER [15] have been proposed. In spirit, our compliance checking technique uses the CEGAR approach. While, the above tools are used for verifying general program, CHIRON is targeted towards checking compliance of event-driven network protocols and hence we can use protocol specific optimizations for compliance checking. Jaffar *et al.* [172] exploits symbolic execution and interpolants to learn infeasible paths and hence avoids

exploring an exponential number of paths. The above analysis techniques can improve CHIRON's precision and scalability while extracting the **E-FSM**. Gulavani *et al.* [173] proposes a new property checking algorithm that combines the ideas of counterexample-guided model checking, directed testing, and partition refinement.

Verifying network protocols and event-driven programs Holzmann *et al.* [19] extract the abstract event-driven program model from a source code using a purely syntactic approach, which requires the end user to annotate the source code heavily. The extracted model is later model checked against the given properties after harnessing the model further by utilizing the user provided map of source statements relevant to the verification to be performed and the user provided test driver to simulate the necessary behaviors to interact with the application. Our approach uses symbolic execution and requires significantly less developer inputs to automatically extract the **E-FSM** of the protocol.

The work by Musuvathi *et al.* [18,20] is the closest to our approach. They develop an explicit state model checker for C/C++ source CMC that verifies user-provided invariants. CHIRON differs from CMC in the following four ways: (1) CHIRON uses pLTL to express properties whereas CMC uses boolean formulas, (2) CHIRON explicitly extracts the **E-FSM** whereas CMC generates parts of the **E-FSM** as necessary for verification, (3) CHIRON uses a symbolic model checker whereas CMC is an explicit-state model checker, (4) CMC focuses on low-level programming errors whereas we are focused on detecting logical programming errors while implementing the **S-FSM**. Clarke *et al.* [174] propose a specialized model checker for security protocols. However, the protocol is required to be expressed in their high level language. CHIRON works on the C source of the protocol. Chaki and Datta [175] combine software model checking with standard protocol security model to verify authentication and secrecy properties of protocol. They require the source to be ported to their

language ASPIRE. CHIRON's FSM extractor directly works on the source and does not emphasize on authentication or secrecy. Bhargavan *et al.* [99] uses SPIN [13] and HOL theorem prover [176] to prove key properties of distance vector routing protocols standards. However, their approach cannot handle protocol implementations. Bhargavan *et al.* [177] advocate the automatic extraction and verification of symbolic cryptographic models from executable code; however, they require the implementation to be in specific language (*i.e.*, F#).

Counterexample guided testing Counterexamples have been used as test inputs to test non-trivial properties [66,67]. The basic idea is generating a model and testing it against some well-formed property. If the property is violated, the CEX is used to generate tests for the program. CHIRON's automatic extraction of the **E-FSM** of the protocol can help generate more targeted and non-trivial test cases from the CEX.

Inferring protocol specification Several other works have looked at inferring protocol specification— based on network traces [57–59,178,179], using program analysis [60, 61, 180, 181], or through model checking [14, 62]. Comparetti *et al.* [57] infer protocol state machines from observed network traces by clustering messages based on the similarity of message contents and their reaction to the execution. While Caballero *et al.* [58] extracts the protocol message format from a trace of protocol messages, Cho *et al.* [59] extracts the protocol state machines from network traces with the help of a set of user-provided abstraction functions to generate an abstract alphabet out of trace messages. However, the protocol state machines extracted following these techniques merely capture the sequences of messages that represent valid sessions of the protocol, which often remains incomplete as it depends on the captured network traces and does not possibly represent the actual protocol FSM.

On the other hand, MACE [61] infers the abstract model given a seed of protocol messages by constructing an abstract grammar and performs concolic execution on the protocol binary to discover vulnerabilities, while iteratively refining the model as it encounters new input/output messages. Unlike MACE, our work does not rely on any seed of messages and extract the actual FSM implemented by the protocol rather than a state machine depicting the sequence of messages of a valid session. Kothari *et al.* [60] employs symbolic execution to derive the FSM from TinyOS programs. However, the FSM is built from the program states representing a very low-level program state machine rather than the protocol FSM. Whereas our work focuses on extracting the high-level protocol FSM from the C-based implementation of the protocol. Similarly, Lie *et al.* [62] uses a source-to-source transformation procedure that translates the protocol implementation (C code) to the modeling language (*metal*) for the model checker (*Murphi*). In addition, it requires the end users to annotate the protocol code in the metal language, which will be used to slice the selected protocol code from the large code base. The users also need to provide a list of translation patterns to be used to translate the sliced- program AST into model checking language. The extracted model combined with user provided hardware model are used to model check against the specified correctness properties. Whereas our technique is much richer in a sense that we extract a high-level protocol FSM, not the program FSM, by using utilizing program analysis rather than source-to-source transformation. Additionally, our technique is independent of the modeling language and the model checker one uses for verification.

Vulnerability discovery in network protocols A variety of works focus on finding vulnerability in network protocol implementations, many using fuzzing. While random fuzz testing [182] is often effective in finding interesting corner case errors, the probability of "hitting the jackpot" is substantially low because it typically mu-

tates the well-formed inputs and tests the program on the resulting inputs. To overcome this inherent problem of fuzzing, a set of works like SNOOZE [63], KiF [64], SNAKE [65] leverage protocol state machine to cover deeper and more relevant portions of the search space. All of them require the end users to provide the protocol specification (*e.g.*, message format, state machine) and various fault injection scenarios or evaluation metric to discover vulnerabilities in stateful protocols such as SIP and transport protocols. On the contrary, several works like MACE [61], Prospex [57] infer protocol models (a sequence of messages valid in a session) to be used for fuzzing to discover vulnerabilities. Our work orthogonally focuses on checking compliance of high-level properties in protocol implementations, which may lead to discovery of state machine bugs.

Several other works [31,183–185] leverage program analysis, for example, symbolic execution, to find vulnerabilities in protocol implementations. MAX [31] focuses on two-party protocols to find performance attacks mounted by a compromised participant that can manipulate the victim's execution control flow. However, MAX relies on the user specified information about a known vulnerability of the implementation to limit the search space during symbolic execution. Similarly, SymbexNet [183] tests two-party protocols by executing one party symbolically to operate on symbolically marked input packets. Thus it can generate high-coverage test input packets for the implementation, which are verified against the packet rules manually derived from the specification. Unlike them, KleeNet [184] and SDE [186] apply symbolic execution to network protocols and distributed systems, respectively. The key idea is to spawn a symbolic execution instance for each participant and allow communication between the symbolic states of different participants as if they were exchange messages through network packets. Besides low-level programming errors, it allows the end users to check simple high-level properties by providing global assertions.

General bug finding tools There is a rich literature (*e.g.*, DART [187], CUTE [188], KLEE [77], EXE [189], BitBlaze [190], S2E [191]) on general bug finding tools that employ symbolic execution. Some (*e.g.*, DART, CUTE, SAGE) are built on the concept of concolic execution – which concretely runs a single execution path and collects and solves symbolic constraints to drive the concrete execution to the next path, whereas others (*e.g.*, KLEE, S2E, EXE) try to execute all possible paths in a single run of the system. These techniques have been effectively used to find bugs or to generate exploits in sequential programs such as Unix utilities. Such bugs often manifest due to low-level programming errors (*e.g.*, segmentation faults, various memory errors due to read/write overflows). While such tools allow the end users to specify invariants in the code using special code construct, such code constructs are not always sufficient to check complex and intricate invariants (*e.g.*, state transition due to a specific network event) as they require additional code to extract the protocol state and perform invariants checking, which further increases the burden on the end users (or the verifier). However, without negating the necessity of those techniques, we intend to find if the implementation under test violates any given high-level protocol properties thereby complementing the general bug finding tools that focus on low-level errors.

5.2 Adversarial Testing

Model checking techniques [13, 192, 193] have been used to verify the correctness of protocol models. Once the model is specified in a high-level modeling language, its correctness is verified mathematically. Many works extended such methods to consider the wireless environment [98, 99, 194, 195]. While model checking techniques have been helpful to show the correctness of the model of a protocol, the high-level descriptions abstract away many details of the actual implementation resulting in

missing vulnerabilities in the abstract model, which may manifest in the actual implementation. Exploration based model checking techniques [18, 20, 170, 196] apply model checking directly on implementations. Specifically, CMC [18] has been applied on different implementations of the AODV protocol, but requires the implementations to be ported to its specialized runtime environment.

Without denying the benefit of model checking, our work is orthogonally different since we focus on bugs/attacks that impair the performance of the protocol in actual executions of the implementation. In addition, one can argue to establish ground truth using model checking or using formally verified reference implementation like [197]. However, note that being able to model check liveness and performance properties is a challenging problem, and to the best of our knowledge, the existing model checking techniques cannot be used to check performance properties. Also, to the best of our knowledge, there are no verified reference implementation for the protocols we tested.

Systematic fault injection is another popular method to improve software robustness [198, 199]. Unlike model checking or symbolic execution, fault injection focuses on exceptional behavior of software by injecting faults. However, such works do not consider adversarial environments as ours where we inject malicious faults that are tailored to imitate attackers.

Several network emulation tools have been developed, for example, NIST Net [200], DummyNet [201], catering wired networks and Emulab [108], Orbit [110], MobiNet [109] catering wireless networks. Some of them even support emulation of network faults while testing various network protocols. Conceptually, these tools, at least the ones designed for wireless networks, could replace the NS-3 network emulator and the virtualization-based nodes. However, such tools would require the user to provide a separate (and malicious) implementation of the routing protocol under test and that

is for each adversary in the network. Whereas, we do not require any such malicious version of the protocol under test.

There have been some recent effort on finding attacks automatically in implementations [30–32, 102]. Kothari et al. [31] automatically find attacks that manipulate control flow by modifying messages using static analysis by relying on a priori knowledge about vulnerability. Stanojevic et al. [32] automatically search for gullibility in two-party protocols by leveraging a variety of techniques: packet-dropping and packet header modifications. Lee et al. [30] automatically discover performance attacks caused by insiders in distributed systems without requiring instrumented implementation. All these works except [102] require the implementation to be written in a specific language.

5.3 Infection Mitigation

We divide the previous works related to our infection mitigation work into two groups. In the first group, we discuss related work in the area of mathematical modeling and analysis of worms and viral epidemics. We then move on to discussing the existing works on controlling the worm propagation.

Epidemic models. Wired networks have been the focus of most literature on worm propagation. A comprehensive overview of major malware outbreaks in networks with a discussion of their trends is given in [202]. There are two popular models that are generally used to describe worm propagation: deterministic [49–51, 203–207] and stochastic [208, 209] epidemiological models. Staniford *et al.* [205] use the SIR epidemiological model to capture the effects of human countermeasures and the congestion due to the worm spread. Shen *et al.* [210] provide a discrete-time worm model that considers patching, cleaning and certain local scanning techniques. All

these approaches abstract network topology and change in the size of vulnerable population as the worm spreads. Theodorakopoulos *et al.* [211] take deterministic modeling one step further and combine it with a game theoretic process that involves learning. A probabilistic queueing framework has been proposed in [212] to model the spread of mobile viruses using short range wireless interfaces (e.g., Bluetooth) of mobile devices. While similar in spirit, our work focuses on modeling infection dynamics in MANETs as a function of the mobility models.

Peng et al. [213] propose a two-dimensional cellular automata to characterize the propagation dynamics of worms in smartphones. Their scheme integrates an infection factor evaluate the spread degree of infected nodes, and a resistance factor to evaluate the degree that susceptible nodes resist. Wang et al. [214] deploy agents in the form of hidden contacts on the device to capture messages sent from malicious applications. The authors combine these captured messages in conjunction with a latent space model to estimate the current dynamics of the system and use this to predict the future state of malware propagation within the mobility network. Our work is complementary to these efforts in that our decentralized algorithms can utilize their models during the learning phase. Szongott et al. [215] present a prototype of a replicating mobile malware that spreads from device to device in downtown Chicago. Using simulations, they show that smartphones create a viable substrate for epidemic mobile malware. Our work differs from them in two key aspects. First, unlike them, we use a more realistic truncated levy walk mobility model. Second, they only study infection propagation whereas we propose several algorithms for recovery.

Worm containment. There have been some works in controlling the spread of worms inside a wireless network [52, 144, 145, 216–218]. Williamson *et al.* [216] present a technique to limit the rate of connections to "new" machines. This is effec-

tive at both slowing and halting virus propagation without affecting normal traffic. Their work is based on heuristics and simulations which consider a static choice of reduced communication rate. Wong *et al.* [217] present a technique that relies on limiting the contact rate of worm traffic. Specifically, they investigate rate control at individual end hosts and at the edge and backbone routers, for both random propagation and local-preferential worms. They show that both host and edge-router based rate control result in a slowdown that is linear to the number of hosts implementing the rate limiting filter.

More recently, Barbera et al. [219] consider the problem of computing an efficient patching strategy to stop worm spreading between smartphones. They consider cases where the worm spreads between the devices and where the worm attacks the cloud before moving to the device. Tang et al. [220] propose distributing patches to certain key nodes so they can opportunistically disseminate them to the rest of the network. In their work, they present a predictive mobile malware containment system where devices collect co-location data in a decentralized manner and report to a central server which processes and targets delivery of hot fixes to a small subset of k devices at runtime. In contrast, our work does not assume a central server and all our algorithms are fully decentralized.

Cole *et al.* [218] present both analytic and simulation analysis of worm propagation focusing specifically on the features of a tactical, battlefield MANETs which are unique to a defense environment. Their goal was to develop an accurate set of performance requirements on potential mitigation techniques of worm propagation for such MANETs. Zou *et al.* [221] compare email worm propagation on three topologies: power law, small world, and random graph topologies; and then study how the topology affects immunization defense on email worms. Their email worm model includes the effect of human interactions. Yang *et al.* [222] utilize a software

diversity approach to deal with the spread of worm in wireless sensor networks. Zhu *et al.* [223] take into account the social relationship of mobile users to contain MMS worms within a limited range in cellular networks. Unlike them, we introduce a suite of defense protocols used by a set of static healers to thwart the epidemic spread inside MANETs.

6 CONCLUSION

Given the importance of the emerging wireless networks, it is essential to ensure their secure and reliable operations, which calls for both pre- and post-deployment measures. Pre-deployment measures are great for proactively identifying and fixing errors in the protocol implementations, thereby gaining confidence in the implementations. In contrast, post-deployment measures are crucial for addressing the aftermath of a security attack through rapid containment of the infection while reducing the recovery time and costs. This dissertation provides novel techniques to fortifying these wireless networks through specification compliance checking and adversarial testing of protocol implementations before deployment, and through infection mitigation at the event of attack after deployment.

Checking implementations for specification compliance using the existing tools is a painstakingly challenging task as they often require extensive manual efforts to specify the model that is checked against the desired properties and suffer from imprecisions due to the underlying syntactic approaches. In this regard, our work on compliance checking, CHIRON, has filled a vital gap by extracting the FSM of the protocol automatically from the source code of the given implementation by symbolically executing the code while requiring a little input from the developer. CHIRON then follows a model-checking approach utilizing a symbolic model checker to check if each of the desired properties is valid against the extracted FSM. In case of any violations, the model checker generated counterexamples undergo CHIRON's two-step validation process to rule out the false positives.

We provide a concrete implementation of CHIRON on top of KLEE. To demonstrate the effectiveness of CHIRON, we applied it to 5 protocol implementations and uncovered 10 instance of non-compliances having security, interoperability, and performance implications. Using traditional testing approaches (*e.g.*, manual testing, random fuzz testing, unit testing), some of these non-compliances can easily remain undetected as they would require the developer to imagine those subtle and intricate sequences of actions to drive the implementations follow some relatively rare execution paths. On the contrary, we discovered these non-compliances using CHIRON with a very little effort and time. Therefore, we conclude that CHIRON can expedite the process of developing specification compliant implementations of network protocols.

Having a specification compliant implementation does not necessarily provide robustness to attacks mounted by compromised nodes in an adversarial environment. Therefore, we proposed Turret-W, an adversarial testing platform to automatically test wireless routing protocol implementations. Turret-W uses a network emulator and virtualization to test unmodified protocol implementations beyond their basic functionalities. Besides general attacks against routing, Turret-W can test various wireless specific attacks.

By using Turret-W on 5 routing protocol implementations, we discovered 37 attacks capable of either impeding the availability by crashing the benign nodes or reducing their performance by disrupting the routing service, and 3 implementation bugs that impair the protocol performance even in a benign environment. To the best of our knowledge, all these bugs and 5 of the total attacks were not previously reported. Given the significance of routing as a fundamental component of wireless networks, we concentrated on the routing protocol implementations in our work of Turret-W. However, it can easily be extended for the protocols operated on the other

layers of the network stack. Therefore, we consider that Turret-W can be applied to greatly improve the robustness of various network protocol implementations.

With the evidence of malware capable of propagating through proximity-based communication (*e.g.*, Bluetooth, NFC), the aftermath of such attacks demands for reactive measures—specific to these networks—to contain the infection and reduce the impacts. In our infection mitigation work, we model the propagation of such malware amongst humans carrying smartphones using epidemiology theory and study the problem as a function of the underlying mobility models. We improve the state-of-the-art by taking into account realistic mobility patterns to model proximity dependent malware as opposed to using the *de facto* mobility models like random waypoint mobility model. Since the optimal approach to heal an infected network using a set of static healers is an NP-Complete problem, we provide three families of healer protocols that allow tuning of parameters to control the optimization along two dimensions: time to recovery and energy utilized. We were very thorough in evaluating the defense mechanisms, *i.e.*, the healer protocols. We observe that the profile healers would benefit the most in an energy constrained environment while the prediction healers would be more effective in a time constrained environment.

Future work. There are several compelling directions to pursue for future work. First, CHIRON requires the properties to be derived manually from the specifications and written in pLTL format. Automating this process is challenging as it involves semantic parsing of the protocol specifications written in natural languages (*e.g.*, English). To tackle this problem, one can leverage well-studied natural language processing techniques. Second, the adversarial testing with Turret-W can be improved by guiding the search of the attack-space using high-coverage test input packets. To tackle this problem, one can leverage some white-box testing techniques to eliminate redundant test input packets and to guide the search toward previously unexplored

execution paths. Finally, in our infection mitigation work, we only provide empirically evaluation of the proposed defense mechanisms through simulation. One can perform complexity analysis to achieve theoretical bounds on those defense mechanisms. This problem is challenging as it requires to take into account some extraneous factors inherent to these mobile networks such as the mobility aspects and the transmission range of the wireless nodes.

REFERENCES

REFERENCES

[1] M. Gast. *802.11 Wireless Networks: The Definitive Guide*. O'Reilly Media, 2005.

[2] N. Kushalnagar, G. Montenegro, D. Culler, and J. Hui. Transmission of IPv6 packets over IEEE 802.15.4 networks. RFC 4944, 2007. `http://www.rfc-editor.org/rfc/rfc4944.txt`.

[3] B. Miller and C. Bisdikian. *Bluetooth revealed*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[4] K. Finkenzeller. *RFID handbook: Fundamentals and applications in contactless smart cards and identification*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[5] R. Want. Near field communication. *IEEE Pervasive Computing*, (3), 2011.

[6] L. Atzori, A. Iera, and G. Morabito. The Internet-of-Things: A survey. *Computer Networks*, 54(15), 2010.

[7] Apple's SSL/TLS bug. `https://www.imperialviolet.org/2014/02/22/applebug.html`. Accessed: 2015.

[8] The Heartbleed bug. `http://heartbleed.com/`. Accessed: 2015.

[9] C. Miller. Exploring the NFC attack surface. In *Proceedings of Blackhat*, 2012.

[10] Anatomy of an attack – The Internet-of-Things (IoT). `https://trapx.com/anatomy-of-an-attack-2/`. Accessed: 2015.

[11] Contiki: The open source OS for the Internet of things. `http://www.contiki-os.org/`.

[12] Contiki bug report. `http://github.com/contiki-os/contiki/commit/d862e`. Accessed: 2015.

[13] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering (TSE)*, 23(5), 1997.

[14] 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.

[15] S. Löwe. CPAchecker with explicit-value analysis based on CEGAR and interpolation. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer-Verlag, 2013.

[16] T. Ball and S. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2002.

[17] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast: Applications to software engineering. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5), 2007.

[18] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: Pragmatic approach to model checking real code. *ACM SIGOPS Operating Systems Review*, 36(SI), 2002.

[19] G. Holzmann and M. Smith. A practical method for verifying event-driven software. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, 1999.

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

[21] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[22] C. E. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. *ACM SIGCOMM Computer Communication Review*, 24(4), 1994.

[23] 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.

[24] 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.

[25] D.B. Johnson and D.A. Maltz. Dynamic source routing in ad hoc wireless networks. *Mobile computing*, 353, 1996.

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

[27] F. de Renesse and A. Aghvami. Formal verification of ad-hoc routing protocols using SPIN model checker. In *Proceedings of the IEEE Mediterranean Electrotechnical Conference (MELECON)*, 2004.

[28] Network simulator 2. `http://www.isi.edu/nsnam/ns/`. Accessed: 2015.

[29] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. GloMoSim: A library for parallel simulation of large wireless networks. *ACM SIGSIM Simulation Digest*, 28(1), 1998.

[30] 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.

[31] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi. Finding protocol manipulation attacks. *ACM SIGCOMM Computer Communication Review*, 41(4), 2011.

[32] 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.

[33] A. Gupta, I. Wormsbecker, and C. Wilhainson. Experimental evaluation of TCP performance in multi-hop wireless ad hoc networks. In *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2004.

[34] G. Anastasi, E. Ancillotti, M. Conti, and A. Passarella. Experimental analysis of a transport protocol for ad hoc networks (TPA). In *Proceedings of the ACM International Workshop on Performance Evaluation of Wireless Ad hoc, Sensor and Ubiquitous Networks (PE-WASUN)*, 2006.

[35] 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.

[36] S. ElRakabawy and C. Lindemann. A practical adaptive pacing scheme for TCP in multihop wireless networks. *IEEE/ACM Transactions on Networking (ToN)*, 19(4), 2011.

[37] Android operating system. `https://www.android.com/`. Accessed: 2015.

[38] FreeRTOS. `http://www.freertos.org/`. Accessed: 2015.

[39] McAfee Threats Report: 3rd Quarter 2011. `http://goo.gl/jIQPJ`. Accessed: 2015.

[40] Juniper Mobile Threats Report 2010-11. `http://goo.gl/v3yFg`. Accessed: 2015.

[41] Android is target for 98 percent of all mobile malware. `http://goo.gl/bpnF2i`. Accessed: 2015.

[42] The Internet of things has arrived – and so have massive security issues. `http://goo.gl/Ft4Js`. Accessed: 2015.

[43] The Internet of things is wildly insecure – and often unpatchable. `http://goo.gl/a597n9`. Accessed: 2015.

[44] Secure all the (Internet of) things. `http://searchsecurity.techtarget.com/feature/Secure-all-the-things`. Accessed: 2015.

[45] Single NFC bonk subjugated Samsung Galaxy SIII and slurped it out. `http://www.theregister.co.uk/2012/09/21/android_nfc/`. Accessed: 2015.

[46] McAfee warns of NFC malware risk. http://www.itpro.co.uk/malware/19275/mcafee-warns-nfc-malware-risk. Accessed: 2015.

[47] Wall Of Sheep hacker group exposes NFC's risks At DefCon 2013. http://goo.gl/j59SQ7. Accessed: 2015.

[48] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer worm. *IEEE Security & Privacy*, 1(4), 2003.

[49] C. Zou, W. Gong, and D. Towsley. Code Red worm propagation modeling and analysis. In *Proceedings of the ACM conference on Computer and communications security (CCS)*. ACM, 2002.

[50] A. Wagner, T. Dübendorfer, B. Plattner, and R. Hiestand. Experiences with worm propagation simulations. In *Proceedings of the ACM workshop on Rapid malcode (WORM)*. ACM, 2003.

[51] J. Kephart and S. White. Directed-graph epidemiological models of computer viruses. In *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE CompSoc, 1991.

[52] S. Sellke, N. Shroff, and S. Bagchi. Modeling and automated containment of worms. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 5(2), 2008.

[53] 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.

[54] FNET embedded TCP/IP stack. http://fnet.sourceforge.net/. Accessed: 2015.

[55] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2000.

[56] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

[57] 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.

[58] 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.

[59] C. Cho, D. Babi ć, 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.

[60] 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.

[61] C. Cho, D. Babic, P. Poosankam, K. Chen, E. Wu, and D. Song. MACE: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proceedings of the USENIX Conference on Security (SEC)*, 2011.

[62] D. Lie, A. Chou, D. Engler, and D. Dill. A simple method for extracting models from protocol code. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*. IEEE, 2001.

[63] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. SNOOZE: Toward a stateful network protocol fuzzer. In *Proceedings of the 9th international conference on Information Security (ISC)*. Springer-Verlag, 2006.

[64] H. Abdelnur, R. State, and O. Festor. KiF: A stateful SIP fuzzer. In *Proceedings of the International Conference on Principles, Systems and Applications of IP Telecommunications*. ACM, 2007.

[65] 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.

[66] G. Fraser, F. Wotawa, and P. Ammann. Testing with model checkers: A survey. *Software Testing, Verification & Reliability*, 19(3), 2009.

[67] D. Beyer, A. Chlipala, T. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2004.

[68] Out in the open: The little-known open source os that rules the internet of things. `http://www.wired.com/2014/06/contiki/`. Accessed: 2015.

[69] Using the fast ethernet controller on the Qorivva MPC564xB/C. `http://cache.freescale.com/files/32bit/doc/app_note/AN4577.pdf`. Accessed 2015.

[70] N. Lynch and M. Tuttle. An introduction to input-output automata. Technical Report MIT-LCS-TM-373, Massachusetts Institute of Technology (Cambridge, MA US), 1988.

[71] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2), 2013.

[72] C. Cadar, P. Godefroid, S. Khurshid, C. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2011.

[73] C. Păsăreanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer*, 11(4), 2009.

[74] P. Godefroid, P. de Halleux, A. Nori, S. Rajamani, W. Schulte, N. Tillmann, and M. Levin. Automating software testing using program analysis. *IEEE Software*, 25(5), 2008.

[75] B. Alpern and F. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3), 1987.

[76] Frama-C. `http://frama-c.com`. Accessed: 2015.

[77] 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.

[78] 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.

[79] KQuery language. `http://klee.github.io/docs/kquery/`. Accessed: 2015.

[80] V. Ganesh and D. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV)*. Springer-Verlag, 2007.

[81] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang. Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. *IEEE Transactions on Software Engineering (TSE)*, 41(7), 2015.

[82] R. Droms. Dynamic host configuration protocol. RFC 2131, 1997. `http://www.rfc-editor.org/rfc/rfc2131.txt`.

[83] J. Postel and J. Reynolds. Telnet protocol specification. RFC 854, 1983. `http://www.rfc-editor.org/rfc/rfc854.txt`.

[84] D.J. Bernstein. The Q method of implementing telnet option negotiation. RFC 1143, 1990. `http://www.rfc-editor.org/rfc/rfc1143.txt`.

[85] TinyOS. `http://www.tinyos.net/`.

[86] T. Bergan, D. Grossman, and L. Ceze. Symbolic execution of multithreaded programs from arbitrary program contexts. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. ACM, 2014.

[87] M. Hoque, H. Lee, R. Potharaju, C. Killian, and C. Nita-Rotaru. Adversarial testing of wireless routing implementations. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*. ACM, 2013.

[88] S. Radhakrishnan, G. Racherla, C.N. Sekharan, N. Rao, and S. Batsell. DST – A routing protocol for ad hoc networks using distributed spanning trees. In *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC)*, 1999.

[89] 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: 2015.

[90] M. Zapata and N. Asokan. Securing ad hoc routing protocols. In *Proceedings of the ACM Workshop on Wireless Security (WiSE)*. ACM, 2002.

[91] B. Awerbuch, R. Curtmola, D. Holmer, C. Nita-Rotaru, and H. Rubens. ODSBR: An on-demand secure byzantine resilient routing protocol for wireless ad hoc networks. *ACM Transactions on Information and System Security (TISSEC)*, 10(4), 2008.

[92] Y. Hu, A. Perrig, and D. Johnson. Ariadne: A secure on-demand routing protocol for ad hoc networks. *Wireless Networks*, 11(1-2), 2005.

[93] AODV-UU. `http://sourceforge.net/projects/aodvuu`. Accessed: 2015.

[94] OLSRD. `http://www.olsr.org`. Accessed: 2015.

[95] ARAN. `http://prisms.cs.umass.edu/arand`. Accessed: 2012.

[96] Click modular router. `http://www.read.cs.ucla.edu/click`. Accessed: 2015.

[97] B.A.T.M.A.N. Advanced. `http://www.open-mesh.org/projects/batman-adv/wiki`. Accessed: 2015.

[98] S. Chiyangwa and M. Kwiatkowska. A timing analysis of AODV. *Proceedings of the 7th IFIP WG 6.1 International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS)*, 2005.

[99] K. Bhargavan, D. Obradovic, and C.A. Gunter. Formal verification of standards for distance vector routing protocols. *Journal of the ACM (JACM)*, 49(4), 2002.

[100] S. Woo and S. Singh. Scalable routing protocol for ad hoc networks. *Wireless Networks*, 7(5), 2001.

[101] C. Killian, J. Anderson, R. Braud, R. Jhala, and A. Vahdat. Mace: Language support for building distributed systems. *ACM SIGPLAN Notices*, 42(6), 2007.

[102] 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 the International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2014.

[103] I. Habib. Virtualization with kvm. *Linux Journal*, 2008.

[104] Network simulator 3. `http://www.nsnam.org`. Accessed: 2015.

[105] G. Finn. Routing and addressing problems in large metropolitan-scale internetworks. Technical report, ISI/RR-87-180, Information Sciences Institute, 1987.

[106] B. Karp and H. Kung. GPSR: Greedy perimeter stateless routing for wireless networks. In *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*. ACM, 2000.

[107] M. Bahr. Update on the hybrid wireless mesh protocol of IEEE 802.11s. In *Proceedings of the IEEE International Conference on Mobile Adhoc and Sensor Systems (MASS)*. IEEE, 2007.

[108] Emulab – network emulation testbed. `http://www.emulab.net/`. Accessed: 2015.

[109] P. Mahadevan, A. Rodriguez, D. Becker, and A. Vahdat. Mobinet: A scalable emulation infrastructure for ad hoc and wireless networks. *ACM SIGMOBILE Mobile Computing and Communications Review*, 10(2), 2006.

[110] ORBIT. `http://www.orbit-lab.org`. Accessed: 2015.

[111] B. Chambers. *The grid roofnet: A rooftop ad hoc wireless network*. PhD thesis, Massachusetts Institute of Technology, 2002.

[112] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3), 2000.

[113] Grid project. `http://pdos.csail.mit.edu/grid/`. Accessed: 2015.

[114] P. Kyasanur and N. Vaidya. Routing and link-layer protocols for multi-channel multi-interface ad hoc wireless networks. *ACM SIGMOBILE Mobile Computing and Communications Review*, 10(1), 2006.

[115] Y. Peng, Y. Yu, L. Guo, D. Jiang, and Q. Gai. An efficient joint channel assignment and QoS routing protocol for IEEE 802.11 multi-radio multi-channel wireless mesh networks. *Journal of Network and Computer Applications*, 36(2), 2013.

[116] Iperf. `http://sourceforge.net/projects/iperf`. Accessed: 2015.

[117] S. Paris, C. Nita-Rotaru, F. Martignon, and A. Capone. Cross-layer metrics for reliable routing in wireless mesh networks. *IEEE/ACM Transactions on Networking (TON)*, 21(3), 2013.

[118] A. Alvarez, R. Orea, S. Cabrero, X. Pañeda, R. García, and D. Melendi. Limitations of network emulation with single-machine and distributed NS-3. In *Proceedings of the International ICST Conference on Simulation Tools and Techniques (SIMUTools)*. ICST, 2010.

[119] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc on-demand distance vector (AODV) routing. RFC 3561, 2003. `http://www.rfc-editor.org/rfc/rfc3561.txt`.

[120] Netfilter. `http://www.netfilter.org/`. Accessed: 2015.

[121] N. Horman. Understanding and programming with Netlink sockets. `http://www.smacked.org/docs/netlink.pdf`, 2004. Accessed: 2015.

[122] K. Sanzgiri, D. LaFlamme, B. Dahill, B.N. Levine, C. Shields, and E.M. Belding-Royer. Authenticated routing for ad hoc networks. *IEEE Journal On Selected Areas in Communications*, 23, 2005.

[123] ASL. `http://sourceforge.net/projects/aslib`. Accessed: 2015.

[124] OpenSSL toolkit. `http://www.openssl.org/`. Accessed: 2015.

[125] Q. Li, M. Zhao, J. Walker, Y. Hu, A. Perrig, and W. Trappe. SEAR: A secure efficient ad hoc on demand routing protocol for wireless networks. *Security and Communication Networks*, 2(4), 2009.

[126] T. Clausen and P. Jacquet. Optimized link state routing protocol (OLSR). RFC 3626, 2003. `http://www.rfc-editor.org/rfc/rfc3626.txt`.

[127] OLSRD source package in Debian. `https://launchpad.net/debian/+source/olsrd/0.6.3-4`.

[128] C. Adjih, T. Clausen, P. Jacquet, A. Laouiti, P. Muhlethaler, and D. Raffo. Securing the OLSR protocol. In *Proceedings of the IFIP Annual Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net)*, 2003.

[129] C. Adjih, D. Raffo, and P. Mühlethaler. Attacks against OLSR: Distributed key management for security. In *Proceedings of the 2nd OLSR Interop/Workshop*, 2005.

[130] T. Clausen and E. Baccelli. Securing OLSR problem statement. `https://tools.ietf.org/html/draft-clausen-manet-solsr-ps-00`, 2005. Accessed: 2015.

[131] Y. Hu, D. Johnson, and A. Perrig. SEAD: Secure efficient distance vector routing for mobile wireless ad hoc networks. *Ad Hoc Networks*, 1(1), 2003.

[132] T. Wan, E. Kranakis, and P. Van Oorschot. Securing the destination-sequenced distance vector routing protocol (S-DSDV). In Javier Lopez, Sihan Qing, and Eiji Okamoto, editors, *Information and Communications Security*, volume 3269 of *Lecture Notes in Computer Science*. Springer, 2004.

[133] E. Graarud. Implementing a secure ad hoc network. Master's thesis, Norwegian University of Science and Technology, 2011.

[134] E. Hoque, R. Potharaju, C. Nita-Rotaru, S. Sarkar, and S.S. Venkatesh. Taming epidemic outbreaks in mobile adhoc networks. *Ad Hoc Networks*, 24, Part A, 2015.

[135] R. Potharaju, E. Hoque, C. Nita-Rotaru, S. Sarkar, and S.S. Venkatesh. Closing the pandora's box: Defenses for thwarting epidemic outbreaks in mobile adhoc networks. In *Proceedings of the IEEE International Conference on Mobile Adhoc and Sensor Systems (MASS)*. IEEE, 2012.

[136] Raspbian. `https://www.raspbian.org/`. Accessed: 2015.

[137] A. Khelil, C. Becker, J. Tian, and K. Rothermel. An epidemic model for information diffusion in MANETs. In *Proceedings of the ACM International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM)*. ACM, 2002.

[138] A. Bose, X. Hu, K. Shin, and T. Park. Behavioral detection of malware on mobile handsets. In *Proceedings of the ACM International Conference on Mobile Systems, Applications and Services (MobiSys)*. ACM, 2008.

[139] C. Fleizach, M. Liljenstam, P. Johansson, G. Voelker, and A. Mehes. Can you infect me now?: Malware propagation in mobile phone networks. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode (WORM)*. ACM, 2007.

[140] A. Bose and K. Shin. On mobile viruses exploiting messaging and bluetooth services. In *Proceedings of Securecomm and Workshops*. IEEE, 2006.

[141] G. Zyba, G. Voelker, M. Liljenstam, A. Méhes, and P. Johansson. Defending mobile phones from proximity malware. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 2009.

[142] R. Potharaju and C. Nita-Rotaru. Pandora: A platform for worm simulations in mobile ad-hoc networks. *ACM SIGMOBILE Mobile Computing and Communications Review*, 14(4), 2011.

[143] I. Rhee, M. Shin, S. Hong, K. Lee, S. Kim, and S. Chong. On the levy-walk nature of human mobility. *IEEE/ACM Transactions on Networking (TON)*, 19(3), 2011.

[144] M. Khouzani, E. Altman, and S. Sarkar. Optimal quarantining of wireless malware through reception gain control. *IEEE Transactions on Automatic Control*, 57(1), 2012.

[145] M. Khouzani, S. Sarkar, and E. Altman. Optimal dissemination of security patches in mobile wireless networks. *IEEE Transactions on Information Theory*, 58(7), 2012.

[146] J. Broch, D. Maltz, D. Johnson, Y. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*. ACM, 1998.

[147] C. Chiang and M. Gerla. On-demand multicast in mobile wireless networks. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*. IEEE, 1998.

[148] J. Garcia-Luna-Aceves and M. Spohn. Source-tree routing in wireless networks. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*. IEEE, 1999.

[149] T. Camp, J. Boleng, and V. Davies. A survey of mobility models for ad hoc network research. *Wireless Communications and Mobile Computing*, 2(5), 2002.

[150] M. Gonzalez, C. Hidalgo, and A. Barabasi. Understanding individual human mobility patterns. *Nature*, 453(7196), 2008.

[151] J. Boleng. Normalizing mobility characteristics and enabling adaptive protocols for ad hoc networks. In *Proceedings of the Local and Metropolitan Area Networks Workshop (LANMAN)*, 2001.

[152] K. Lee, S. Hong, S. Kim, I. Rhee, and S. Chong. Slaw: A new mobility model for human walks. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 2009.

[153] C. Boldrini and A. Passarella. HCMM: Modeling spatial and temporal properties of human mobility driven by users' social relationships. *Computer Communications*, 33(9), 2010.

[154] S. Isaacman, R. Becker, R. Cáceres, M. Martonosi, J. Rowland, A. Varshavsky, and W. Willinger. Human mobility modeling at metropolitan scales. In *Proceedings of the ACM International conference on Mobile systems, applications and services (MobiSys)*. ACM, 2012.

[155] V. Capasso and G. Serio. A generalization of the Kermack-McKendrick deterministic epidemic model. *Mathematical Biosciences*, 42(1), 1978.

[156] C. Huang, C. Sun, and H. Lin. Influence of local information on social simulations in small-world network models. *Journal of Artificial Societies and Social Simulation*, 8(4), 2005.

[157] R. Potharaju, C. Nita-Rotaru, S. Sarkar, and S. Venkatesh. Infection quarantining for wireless networks using power control. In *Proceedings of the Annual Information Security Symposium (SREIS)*. CERIAS - Purdue University, 2010.

[158] N. Aschenbruck, R. Ernst, E. Gerhards-Padilla, and M. Schwamborn. BonnMotion: A mobility scenario generation and analysis tool. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools)*. ICST, 2010.

[159] C. Bettstetter and C. Wagner. The spatial node distribution of the random waypoint mobility model. In *Proceedings of the German Workshop on Mobile Ad Hoc Networks (WMAN)*, 2002.

[160] G. Resta and P. Santi. An analysis of the node spatial distribution of the random waypoint mobility model for ad hoc networks. In *Proceedings of the ACM International Workshop on Principles of Mobile Computing (POMC)*. ACM, 2002.

[161] E. Hyytiä and J. Virtamo. Random waypoint mobility model in cellular networks. *Wireless Networks*, 13(2), 2007.

[162] J. Levine, J. Grizzard, and H. Owen. Detecting and categorizing kernel-level rootkits to aid future detection. *IEEE Security & Privacy*, 4(1), 2006.

[163] V. Vazirani. *Approximation Algorithms*. Springer-Verlag New York, Inc., 2001.

[164] R. Bridson. Fast Poisson disk sampling in arbitrary dimensions. In *ACM SIGGRAPH*. ACM, 2007.

[165] Two-way analysis of variance. `http://en.wikipedia.org/wiki/Two-way_analysis_of_variance`. Accessed: 2015.

[166] C. Gkantsidis and P. Rodriguez. Network coding for large scale content distribution. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 2005.

[167] R. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Proceedings of a Symposium on Applied Mathematics*, volume 19. American Mathematical Society, 1967.

[168] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 1969.

[169] T. Andrews, S. Qadeer, S. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*. Springer, 2004.

[170] 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.

[171] 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.

[172] J. Jaffar, V. Murali, J. Navas, and A. Santosa. TRACER: A symbolic execution tool for verification. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*. Springer, 2012.

[173] B. Gulavani, T. Henzinger, Y. Kannan, A. Nori, and S. Rajamani. Synergy: A new algorithm for property checking. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2006.

[174] E. Clarke, S. Jha, and W. Marrero. Verifying security protocols with Brutus. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4), 2000.

[175] S. Chaki and A. Datta. ASPIER: An automated framework for verifying security protocol implementations. In *Proceedings of the 22nd IEEE Computer Security Symposium (CSF)*, 2009.

[176] M. Gordon and T. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, 1993.

[177] K. Bhargavan, C. Fournet, A. Gordon, and S. Tse. Verified interoperable implementations of security protocols. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(1), 2008.

[178] 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.

[179] W. Cui, J. Kannan, and H. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *Proceedings of 16th USENIX Security Symposium*, 2007.

[180] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proceedings of Network & Distributed System Security Symposium (NDSS)*, volume 8, 2008.

[181] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM, 2007.

[182] B. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12), 1990.

[183] J. Song, C. Cadar, and P. Pietzuch. SymbexNet: Testing network protocol implementations with symbolic execution and rule-based specifications. *IEEE Transactions on Software Engineering*, 40(7), 2014.

[184] R. Sasnauskas, O. Landsiedel, M. Alizai, C. Weise, S. Kowalewski, and K. Wehrle. KleeNet: Discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proceedings of the IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. ACM, 2010.

[185] R. Banabic, G. Candea, and R. Guerraoui. Finding trojan message vulnerabilities in distributed systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[186] R. Sasnauskas, O. Dustmann, B. Kaminski, K. Wehrle, C. Weise, and S. Kowalewski. Scalable symbolic execution of distributed systems. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2011.

[187] 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.

[188] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2005.

[189] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2), 2008.

[190] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS)*. Springer-Verlag, 2008.

[191] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[192] K. McMillan. *Symbolic model checking: An approach to the state explosion problem*. PhD thesis, Pittsburgh, PA, USA, 1992.

[193] D.L. Dill, A.J. Drexler, A.J. Hu, and C.H. Yang. Protocol verification as a hardware design aid. In *Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors (ICCD)*. IEEE, 1992.

[194] S. Nanz and C. Hankin. A framework for security analysis of mobile wireless networks. *Theoretical Computer Science*, 367(1), 2006.

[195] I. Zakiuddin, M. Goldsmith, P. Whittaker, and P. Gardiner. A methodology for model-checking ad-hoc networks. In Thomas Ball and Sriram K. Rajamani, editors, *Model Checking Software*, volume 2648 of *Lecture Notes in Computer Science*. Springer, 2003.

[196] W. Visser, K. Havelund, G. Brat, S.J. Park, and F. Lerda. Model checking programs. *Automated Software Engineering (ASE)*, 10(2), 2003.

[197] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and J. Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2015.

[198] P.D. Marinescu and G. Candea. Efficient testing of recovery code using fault injection. *ACM Transactions on Computer Systems (TOCS)*, 29(4), 2011.

[199] H. Gunawi, T. Do, P. Joshi, P. Alvaro, J. Hellerstein, A. Arpaci-Dusseau, R. Arpaci-Dusseau, K. Sen, and D. Borthakur. Fate and Destini: A framework for cloud recovery testing. In *Proceedings of the Conference on Symposium on Networked Systems Design and Implementation (NSDI)*. Usenix, 2011.

[200] M. Carson and D. Santay. NIST Net: A Linux-based network emulation tool. *ACM SIGCOMM Computer Communication Review*, 33(3), 2003.

[201] L. Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review*, 27(1), 1997.

[202] D.M. Kienzle and M.C. Elder. Recent worms: A survey and trends. In *Proceedings of the ACM Workshop on Rapid Malcode (WORM)*. ACM, 2003.

[203] J.O. Kephart, S.R. White, and D.M. Chess. Computers and epidemiology. *IEEE Spectrum*, 30(5), 1993.

[204] J.O. Kephart and S.R. White. Measuring and modeling computer virus prevalence. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 1993.

[205] S. Staniford, V. Paxson, and N. Weaver. How to own the Internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*. Usenix, 2002.

[206] G. Serazzi and S. Zanero. Computer virus propagation models. In M. Calzarossa and E. Gelenbe, editors, *Performance Tools and Applications to Networked Systems*, volume 2965 of *Lecture Notes in Computer Science*. Springer, 2004.

[207] G. Kesidis, I. Hamadeh, and S. Jiwasurat. Coupled Kermack-McKendrick models for randomly scanning and bandwidth-saturating Internet worms. In *Proceedings of the 3rd International Conference on Quality of Service in Multi-service IP Networks (QoS-IP)*. Springer-Verlag, 2005.

[208] R.M. Anderson and R.M. May. *Infectious Diseases of Humans: Dynamics and Control*. Oxford University Press, 1992.

[209] H. Andersson and T. Britton. *Stochastic Epidemic Models and Their Statistical Analysis*. Springer Verlag, 2000.

[210] Z. Chen, L. Gao, and K. Kwiat. Modeling the spread of active worms. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 2003.

[211] G. Theodorakopoulos, J.S. Baras, and J.Y. Le Boudec. Dynamic network security deployment under partial information. In *Proceedings of the 46th Annual Allerton Conference on Communication, Control, and Computing*. IEEE, 2008.

[212] J. Mickens and B. Noble. Modeling epidemic spreading in mobile environments. In *Proceedings of the ACM Workshop on Wireless Security (WiSe)*. ACM, 2005.

[213] S. Peng, G. Wang, and S. Yu. Modeling the dynamics of worm propagation using two-dimensional cellular automata in smartphones. *Journal of Computer and System Sciences*, 79(5):586 – 595, 2013.

[214] W. Wang, I. Murynets, J. Bickford, C. Wart, and G. Xu. What you see predicts what you getlightweight agent-based malware detection. *Security and Communication Networks*, 6(1), 2013.

[215] C. Szongott, B. Henne, and M. Smith. Evaluating the threat of epidemic mobile malware. In *Proceedings of the IEEE 8th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*. IEEE, 2012.

[216] M. Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. In *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2002.

[217] C. Wong, C. Wang, D. Song, S. Bielski, and G. Ganger. Dynamic quarantine of internet worms. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2004.

[218] R. Cole. Initial studies on worm propagation in MANETs for future army combat systems. In *Proceedings of the 24th Army Science Conference*, 2004.

[219] M. Barbera, S. Kosta, J. Stefa, P. Hui, and A. Mei. CloudShield: Efficient anti-malware smartphone patching with a P2P network on the cloud. In *Proceedings of the IEEE 12th International Conference on Peer-to-Peer Computing (P2P)*. IEEE, 2012.

[220] J. Tang, H. Kim, C. Mascolo, and M. Musolesi. STOP: Socio-temporal opportunistic patching of short range mobile malware. In *Proceedings of the IEEE International Symposium on World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. IEEE, 2012.

[221] C. Zou, D. Towsley, and W. Gong. Email worm modeling and defense. In *Proceedings of the 13th International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2004.

[222] Y. Yang, S. Zhu, and G. Cao. Improving sensor network immunity under worm attacks: A software diversity approach. In *Proceedings of the ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*. ACM, 2008.

[223] Z. Zhu, G. Cao, S. Zhu, S. Ranjan, and A. Nucci. A social network based patching scheme for worm containment in cellular networks. In M. Thai and P. Pardalos, editors, *Handbook of Optimization in Complex Networks*, volume 58 of *Springer Optimization and Its Applications*. Springer New York, 2012.

VITA

VITA

Md. Endadul Hoque received his Bachelor of Science in Computer Science and Engineering from Bangladesh University of Engineering and Technology (BUET), Bangladesh in 2008 and Master of Science in computer science from Marquette University, Wisconsin in 2010. He received his PhD in computer science from Purdue University in 2015. During his time at Purdue, he was a member of the Dependable and Secure Distributed Systems Lab and was affiliated with the Center for Education and Research in Information Assurance and Security (CERIAS). His research focused on network security—compliance checking of protocol implementations, vulnerability discovery using adversarial testing, and countermeasures to mitigate malware infection in the network. He also conducted research on finding performance attacks in reliable distributed system implementations.