UNIVERSITY
OF OSLO

Master's thesis

Runtime Verification with Linux eBPF

Nemanja Lakicevic

Programming and System Architecture
60 ECTS study points

Department of Informatics
Faculty of Mathematics and Natural Sciences

Spring 2023

Nemanja Lakicevic

Runtime Verification with Linux eBPF

Supervisor:
Volker Stolz

Abstract

As software systems become increasingly complex, there is a growing need for better tools and concepts to analyse and understand them. This requires practical instrumentation tools that can provide safe and efficient insights into running systems. Tools that enable better ways of reasoning about systems are required, especially as their state and behaviour change over time. Runtime verification is a promising field of research, which focuses on rigorously specifying properties of software systems and creating corresponding automata-based monitors. It positions itself as a lightweight formal method, by verifying whether the system under scrutiny satisfies specification under runtime. New observability and security enforcement tools are emerging in parallel. A novel tracing framework within the Linux ecosystem, eBPF, has quickly garnered attention for its to its versatility and effective tracing. This thesis explores the use of Linux eBPF for conducting runtime verification, and presents `dottobpf`, a tool which generates eBPF programs from three-valued LTL specification. The effectiveness of the tool, as well as suitability of eBPF for Runtime Verification is evaluated through the analysis of both single- and multi-process systems.

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface. 19, 21, 24, 48, 57, 59, 61, 69

BA Büchi Automata. xi, 8, 13

BCC Berkley Packet Filtering (BPF) Compiler Collection. 19, 21, 34, 66

BPF Berkley Packet Filtering. xi, 1, 15–19, 69

BPF CO-RE BPF Compile Once — Run Everywhere. 19, 21–24, 34, 53, 67–69, 83, 85, 87

BTF BPF Type Format. 18, 19, 21, 27–29, 67, 68, 71, 77

CNCF Cloud Native Computing Foundation. 19

CNI Container Networking Interface. 19

DBA Deterministic Büchi Automata (BA). 8

DFA Deterministic Finite Automata. 7, 8, 33, 34, 67

eBPF extended BPF. 1, 2, 15–19, 21–25, 27–29, 33–36, 39, 40, 43–45, 47, 52, 54, 58, 59, 61, 63, 66–71, 73

ELF Executable and Linkable Format. 18, 21, 35, 52, 53

FA Finite Automata. 7, 8, 10, 13

FSM Finite State Machine. 7, 13

JIT Just-in-Time. 17

JSON JavaScript Object Notation. 29

LT Linear Time. 12, 35

LTL Linear Temporal Logic. xi, 1, 2, 10–12, 34, 35, 49, 60, 66

LTL$_3$ Three-valued Linear Temporal Logic (LTL). 1, 2, 12, 13, 33, 34, 43, 67, 73

LTTng Linux Trace Toolkit: next generation. 16

NBA Nondeterministic BA. 8, 13

NFA Nondeterministic Finite Automata. 7, 8, 10

PLTL Past-time LTL. 34

SUS System Under Scrutiny. 5, 6, 34, 35, 49, 52, 66, 67, 73

USDT User-level Statically Defined Tracepoint. 16, 68, 81

XDP eXpress Data Path. 19, 71

Preface

First and foremost, I would like to express gratitude my supervisor, Volker Stolz, for introducing me to this awesome technology, as well as his invaluable guidance and support throughout the entire research process. His insightful feedback, constructive criticism, and encouragement have been instrumental in shaping this thesis and enhancing the quality of my work.

I would also like to thank my parents, and my sister for their constant optimism and reassurance. Their unwavering belief in me and my abilities has been a constant source of motivation. I am grateful for their support during this journey, and I hope that this thesis reflects the efforts of all those who have helped me along the way.

Lastly, I want to thank you for taking the time to read my thesis and I hope that you will find the same pleasure in reading it as I did in creating it.

Chapter 1

Introduction

Designing and developing software systems that are safe – free from vulnerabilities and crashes, and correct – behave as intended, is of paramount importance, especially considering the permeance and ubiquity of ICT today. To accomplish this, engineers rely on a variety of techniques, such as testing, debugging, code-reviews, systematic fuzzing and formal methods. Formal verification methods apply mathematical models and proof systems to ensure correctness of a software system. Nonetheless, rigorous verification techniques can be computationally expensive for complex systems, and require significant expertise in formal methods.

Runtime verification is lightweight verification technique that monitors the execution of a system to ensure correctness with respect to some specification. Thus, it provides real-time feedback on the behaviour of a system and can detect errors that were not detected during testing. Its utility attracts interest from a multitude of domains, with some tools having even been integrated into the Linux kernel [48][19].

extended BPF (eBPF) is is a highly versatile and programmable framework that provides a secure and efficient way to analyse and filter network packets. It has quickly gained popularity for its ability to perform real-time tracing and system enforcement with very low overhead. eBPF achieves this by allowing users to safely inject custom code into the kernel without compromising system stability or security. This enables developers to implement a wide range of monitoring, tracing, and security features in a performant and flexible manner.

This thesis leverages the potential of eBPF for runtime verification, and makes following contributions:

1. A tool called `dottobpf` is designed and implemented for generating eBPF-based monitors for properties specified in Three-valued LTL (LTL_3). When used in conjunction with *LamaConv*, an automata conversion library, `dottobpf` provides a platform for runtime verification.

2. The generated monitors are applied to two software systems: a simple stack implementation and a distributed authentication system. These case studies aim to demonstrate the capabilities of `dottobpf` generated monitors, as well as their limitations.

3. With insights gained from these case studies, the design and limitations of `dottobpf` are evaluated, and directions for future research on utilising eBPF for runtime verification are proposed.

Thesis Structure

This thesis is structured into two distinct parts. Part I lays the foundation by introducing the theoretical aspects of runtime verification and technical aspects of eBPF. Part II presents `dottobpf` and its capabilities through case studies. Part III provide installation and reproduction guides for discussed case studies.

- Chapter 2 introduces formal methods, and introduces runtime verification in light of model checking. Additionally, it provides as brief overview of automata theory and its formalism. Chapters are orderes as follows:
- Chapter 3 delves into LTL and LTL_3 syntax and semantics and presents a procedure for deriving LTL_3-based automata.
- Chapter 4 introduces eBPF and its architecture, along with various tools that facilitate its development.
- Chapter 5 covers the technical details of developing with eBPF seeking to equip the reader with the necessary knowledge to understand and evaluate eBPF programs generated by `dottobpf`.
- Chapter 6 marks the start of part II and introduces `dottobpf` and discusses its design principles and implementation, highlighting the connection between logical and practical concepts by associating LTL properties with eBPF tracing facility.
- Chapter 7 presents a case study of a simple stack, serving as an introductory example to the runtime verification workflow with `dottobpf`.
- Chapter 8 proceeds with the runtime verification of a more complex case study, which spans multiple systems and showcases the applicability and limitations of eBPF for runtime verification. These limitations are further discussed in section 10.4.
- Chapter 9 provides an alternative method to perform runtime verification to that presented in Chapter 8 and specifically highlights the capabilities of `dottobpf` to parameterise automata.
- Chapter 10 discusses the main challenges encountered during this thesis, including the design choices, implementation of `dottobpf`, and the applicability of generated monitors.
- Chapter 11 summarises the findings and results and provides concluding remarks for this thesis.

Part I

Foundations

Chapter 2

Formal Methods

Formal methods refer to a set of system design techniques that employ unambiguous mathematical models for the development of software and hardware systems, and provide mathematical proof to ensure correctness with respect to specified behaviour. This becomes especially significant when systems grow in complexity and safety becomes a critical concern, making the formality an attractive option for added insurance. While traditional validation techniques, are en important part of system design cycle, they are *finite*, and can only demonstrate that the system works for *tested cases*. Being unable to demonstrate that the system should work as intended outside of tested cases, formal methods provide proof by considering the system as *infinite*. Formal methods are not a substitute for testing and validation, but rather complementary. However, for all their benefits and amazing progress in recent years, traditional formal methods have viewed with suspicion. The main challenges that impede their wide adoption are related to cost of implementation, design complexity and applicability. In order to provide complete coverage, formal verification tools are generally solving problems that are NP-complete[1] [50, ch. 10]. Moreover, formal verification is performed on the model of the system, rather than the system itself.

These challenges motivated research on lightweight verification techniques, that approach application of formal methods selectively, by leveraging strengths of different methods for different subsystems. *Runtime verification* is one such technique that aims to monitor *live* execution of a system and verify whether its *trace* satisfies specified properties. It is related to field of *Model checking*, which it adopts many concepts from. A short introduction of Model Checking is therefore worthwhile.

2.1 Model checking

Model checking involves constructing mathematical models of the System Under Scrutiny (SUS), and exploring all possible states in a brute-force manner. Analogous to a computer chess program checking all possible moves, a model checker (the software tool performing model checking), examines all possible system scenarios systematically. Thus, it provides a more comprehensive validation than testing, and being automatic, more practical than theorem proving. Being a static analysis technique, it seeks to uncover errors during the process of system design and development. Application of model checking to a design can be divided into distinct phases [4, p. 11]:

- *Modelling phase* – Deriving a formal model of SUS and formally specifying some desired property φ that the model should verify.
- *Execution phase* – Run an exhaustive search for a computation[2] that falsifies φ.
- *Analysis phase* – Assert whether property φ is satisfied or violated. If violated or out-of-memory result, consider the counterexample generated by the simulation, refine the model and repeat the procedure.

[1]The time to solve the problem is exponential in relation to the size of its input set.
[2]Computation here refers to executing the model checker.

Model checking has proved quite successful in verifying finite-state systems like hardware controllers and communication protocols. Regardless, inherent limitations motivate exploration of complementary techniques. Firstly, a verification is rendered on the model of the system rather than the system itself. Secondly, it suffers from decidability issues for infinite-state systems. Consequently, its use is more appropriate for control-intensive replication rather than data-intensive, as data typically ranges over infinite domains. Finally, the main obstacle model checking faces is the so-called *state explosion problem* by solving NP-complete problems. Despite the development of effective methods to combat this problem, models of realistic systems may still be too large to fit in memory [7].

2.2 Runtime verification

Runtime verification is a formal method concerned with monitoring and analysis of runtime behaviour. Formal definition, given by Leucker [2012] in [37] is as follows:

Definition 2.2.1 (Runtime verification)
Runtime verification is the discipline of computer science that deals with the study, development, and application of those verification techniques that allow checking whether a run of a SUS satisfies or violates a given correctness property. Its distinguishing research effort lies in synthesising monitors from high level specifications.

As follows from the definition, runtime verification is dynamic in nature and applied to the execution of a program, continually verifying whether the system satisfies its specified properties. While ensuring correctness of runtime behaviour is the quintessential application, Bartocci et al. [2013] cite publications for other types of analysis e.g., falsification analysis and runtime enforcement [5, p. 1].

In comparison to model checking, runtime verification procedure can be divided into three analogous phases, where the first phase, namely *modelling*, is quite similar. The *execution phase* in model checking is performed by model checking software, while in runtime verification, the execution phase is performed by a synthesised monitor which consumes target systems execution trace. Thus, the synthesis of monitors with respect to specified property φ is one of the central tenets in runtime verification. Moreover, the verification verdict obtained by model checking software, is referring to the model of the system, as applying such techniques directly to the real implementation is intractable. An implementation, however, might behave differently from the predicted by the model, and the behaviour of the application may depend heavily on the environment of the target system. Consequently, the *analysis phase* in runtime verification is fundamentally different to that of model checking — where model checking is dependent on the quality of the model, runtime verification can give precise information obtained in the real world, albeit at the price of limited coverage. Additionally, runtime verification can be applied both during system design for testing, verification or debugging purposes, and after deployment for ensuring reliability, safety, and security. Thus, the analysis and can performed simultaneously with the execution phase. Analysis of an executing system trace is referred to as *online analysis*, as opposed to *offline analysis* where a recorded trace is analysed in a postmortem fashion. Online analysis makes acting on events that violate the specified properties possible, and so create 'steering systems' that automatically alter a running system in response [38, p. 294-295].

Finally, considering usability, runtime verification can be applied to black box systems with proper instrumentation, while model checking requires a concrete model of the program in question.

2.3 Automata Theory

Discovering the fundamental capabilities and limitations of computers has motivated mathematicians since the 1930s. These efforts have flourished into the field now known as theory of computing. Its central area are automata, computability, and complexity theory. Automata theory is concerned with the definitions and properties of mathematical models of computation, i.e., logic of computation. It is closely associated to theory of formal languages, from which context automata act as finite representations of possibly infinite formal languages.

Automatons are abstract computational models that facilitate application of mathematical formalism. Invariably, automata perform computations on some input, thereby traversing a set of states. Automata are derived with respect to the formal language best suited for the problem at han. One model, which will be discussed below, the finite automaton, is applied in text processing, compilers and hardware design. Another model, namely *pushdown* automata, also known as *context-free grammar*, proliferates in programming languages and artificial intelligence. The most general and powerful automata is the Turing machine [51].

2.3.1 Finite Automata

Finite Automata (FA), or Finite State Machine (FSM), are characterized by the typical directed graph, or state diagram. Their memory is limited to the finite number of possible states, and can accordingly only handle situations in which the information to be stored is strictly bounded. They are well suited for applications with relatively small amount of memory. FA consists of five elements: *state*, *alphabet*, *transition function*, *starting state* and *final states* [51, p. 35].

Definition 2.3.1 (Finite Automaton)
A finite automaton is a 5-tuple $(Q, \Sigma, \delta, q, F)$, where

Q	is a finite set of *states*
Σ	is a finite set of input symbols, called the *alphabet*
$\delta : Q \times \Sigma \to Q$	is a *transition function*
$q \in Q$	is a *starting state*
$F \subseteq Q$	is the set of *final states*

Finite automata generate *regular languages*, or in other words, a language is called a *regular language* if some FA recognizes it.

2.3.2 Determinism and Equivalence

FA can be either deterministic or nondeterministic, respectively Deterministic Finite Automata (DFA) and Nondeterministic Finite Automata (NFA). The formal definition 2.3.1 is of a DFA. A deterministic computation implies that the transition function yields a single resulting state. A nondeterministic computation, on the other hand, may yield multiple resulting states at any point in time, i.e., its transition function yields a power set of Q, denoted $P(Q)$. Moreover, NFA transition function accepts an empty string, ε. The formal definition of NFA transition function is thus

$$\delta : Q \times \Sigma_\varepsilon \to P(Q)$$

When presented with multiple possible next states, an NFA creates copies of itself and follows *all* possibilities in parallel. Such a computation can be visualized as a tree of possibilities. The root of the tree corresponds to the automaton's initial state, and each branching point represents a point in the computation where the machine has multiple options to consider [51, p .48]

A natural question that arises is whether problems that can be solved by one model but not the other, exist. Considering that both DFA and NFA recognize regular languages, given by regular expressions, the answer is no. Two machines are said to be *equivalent* if they recognize the same language. Consequently, for every NFA an equivalent DFA can be derived[3]. The equivalence is surprising as it is useful; using NFA as a design tool, as they easier to reason about, even if DFA is ultimately required [51, p .54].

2.3.3 Büchi Automata

As the name implies, FA are finite, that is they recognize regular languages that by definition are finite. BA however, formulated by Büchi [1960], is defined to accept *ω-regular languages*, that generalize the definition of regular languages to infinite words. Thus, BA are conceptually similar to FA, but differ in their acceptance criteria, which in BA are extended to the infinite domain. A finite automaton accepts a string from some alphabet, if and only if it starts in one of the states from the set of starting states, and finishes in a state from a set of final states. In contrast, BA are defined for infinite strings and accept a string only if the automaton visits some final state infinitely often. Similarly to FA, BA can deterministic or nondeterministic, respectively Deterministic BA (DBA) and Nondeterministic BA (NBA) [4, p. 170].

[3]For proof see, [51, p. 55].

Chapter 3

Temporal Logic

As the complexity of computational systems increases, it becomes crucial to provide unambiguous descriptions of their behaviour. Application of formal languages, and particularly formal logic, are increasingly popular methods for precise representations. Formal logic offers a precise language for specifying computational properties and a well established logical framework for manipulating and analysing these descriptions. Such specifications can be used to establish the correctness of a given system in all possible situations. In order to appropriately describe complex dynamic properties, temporal logic is a popular choice in computer science. Originally used to represent tense in natural language, temporal logic has become an important tool for formal specification and verification of concurrent and distributed systems [24, p. 1-7].

Temporal logic is a modal logic with respect to time. A *modal* is an expression that qualifies the truth of an assertion. By extending classical logic with modalities, modal logic addresses the idea that a truth evaluation is relative to *possible worlds*. That is, an assertion at some world may depend on assertions at other accessible worlds. Then, applying modalities to the domain of time permits referral to the infinite behaviour of a reactive system. Temporal logics have thus successfully tackled the analysis of reactive systems. Properties, such as safety, liveness, and fairness can be formally and concisely expressed. Consequently, logical analysis can be conducted on specified properties and ultimately deriving a logical proof about system behaviour [24, p. 1-7][34, p. 3].

It is worth noting that adjective *temporal* here does not refer to the real-time behaviour of a system, but rather allows for specification of the *relative order* of events. The complex nature and properties of time are reflected in the rich variety of temporal models. The most common structures are *linear* — having at most one accessible world at any time, and *branching* — multiple accessible worlds at any time. Another important distinction is between discrete, dense, or continuous models. Which temporal logic suits a given context is dependent on notion of time that best captures the phenomena one wishes to study [27]. Application of temporal logic to automata theory was pioneered by Pnueli [45].

3.1 Transition Systems

In computer science, temporal logics are used to model transition systems thereby enabling reasoning about system behaviour over time. Transition systems are formal models of computation and are characterised by the typical directed graph or state diagram. They consist of states, transitions, and a set of rules that determine how the transitions are executed. A state indicates the current values of system variables, transitions specify how the system evolves from one state to another, and a labelled function which triggers a transition. Atomic propositions intuitively express simple known facts about the states of the system under consideration, and are used to formalise temporal characteristics [4, p. 19].

Definition 3.1.1 (Labelled Transition Systems)
A labelled transition system TS is a tuple $(S, \rightarrow, L)$ where

- S denotes a set of states
- $\rightarrow \subseteq S \times S$ denotes a transition relation
- $L : S \rightarrow 2^{AP}$ denotes a labelling function

where AP is a set of atomic propositions, with assumption that $AP \subseteq S$ with labelling function $L(s) = \{s\} \cap AP$.

Traversal of states in a transition systems produces a *path*, either finite or infinite.

Definition 3.1.2 (Path)
Given a path π,

$\mid \pi \mid$	denotes the length of π, for an infinite path $\mid \pi \mid = \infty$
π_i	denotes the i-th position in π for all $0 \leq i \leq \mid \pi \mid$
π^i	denotes the i-th suffix of π, i.e. $\pi^i = \pi_i \, \pi_{i+1} \, \pi_{i+2} \ldots \pi_{\mid\pi\mid-1}$

However, states themselves are not ‘observable’, but rather their atomic propositions. Thus, sequences of states are considered as interpretation of atomic propositions, evaluated by a labelling function. Such sequences are called traces [4, p. 97].

Definition 3.1.3 (Trace)
Let $TS = (S, \rightarrow, L)$ be a transition system, and π a path. The trace of π

$$trace(\pi) = L(\pi_0)\, L(\pi_1)\, L(\pi_2) \, \ldots L(\pi_{\mid\pi\mid-1})$$

The keen reader has likely noted the similarity between transition systems and FAs. However, transition systems differ from FAs on following points:

- A set of states need not be finite
- A set of transitions need not be finite
- No initial and finite states must be given

Nonetheless, they are still conceptually related. Labelled transition systems may be obtained from FAs, simply by ignoring the set of initial and final states. Conversely, transition systems may be finite, in which case initial and final states are included. A labelled transition system having a finite set of states and a finite alphabet, as well as initial and finial states, corresponding NFA can be derived.

3.2 Linear Temporal Logic

LTL considers time as linear sequence of discrete events. A discrete event, here, corresponds to a world, or state in LTL terminology, and the next corresponds to its immediate successor. *Linear* implies that the sequence of states forms a straight line, ergo, system behaviour is assumed to be observable and the time points 0, 1, 2 .... Commonly, sequences of states are considered infinite and referred to as a path or a trace. LTL-formulae specify properties that are evaluated a long a trace, which either satisfies or violates a formula.

3.2.1 Syntax

LTL formulae consist of atomic propositions, boolean connectors and, temporal modalities. In LTL, temporal modalities are expressed through following operators:

- G | $\Box$ — Always
- F | $\Diamond$ — Eventually
- X | $\bigcirc$ — Next
- U — Until
- R — Release
- W — Weak Until

Definition 3.2.1 (LTL — Syntax)
Let *AP* be a set of atomic propositions. LTL formulae over *AP* are given by following syntax

$$\varphi \ ::= \ \text{true} \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi \ U \ \varphi$$

where $p \in AP$.

Note that definition above only covers the basic temporal operators, $\bigcirc$ *(Next)* and *U (Until)*. This is due to the fact that these operators are complete insofar that they can express remaining operators, as will be shown below. This implies that the syntax of LTL is not *minimal*, i.e., syntactical sugar is provided by deriving remaining operators by using already defined operators. These can be defined as follows.

$$\begin{aligned}
\Diamond\varphi &\overset{\text{def}}{=} \top\, U\, \varphi \\
\Box\varphi &\overset{\text{def}}{=} \neg\Diamond\neg\varphi \\
\varphi_1\, R\, \varphi_2 &\overset{\text{def}}{=} \neg(\neg\varphi_1\, U\, \neg\varphi_2) \\
\varphi_1\, W\, \varphi_2 &\overset{\text{def}}{=} \varphi_1\, U\, \varphi_2 \vee \neg(\top\, U\, \neg\varphi_1)
\end{aligned}$$

3.2.2 Semantics

The semantics of LTL are defined as languages over infinite or non-empty finite words. An *alphabet* is set of letters, which form a finite or infinite *word*. A word is defined analogously to a path in a transition system, see definition 3.1.2. Let φ be an LTL formula and π a word. The satisfaction relation $\pi \models \varphi$ is defined inductively over the structure of the formula.

Definition 3.2.2 (LTL — Satisfaction)
A word π satisfies an LTL formula φ, written $\pi \models \varphi$, under following conditions

$$\begin{array}{lll}
\pi \models \top & & \\
\pi \models p & \text{iff} & p \in \pi_0, \text{ (i.e., } \pi_0 \models p) \\
\pi \models \varphi_1 \wedge \varphi_2 & \text{iff} & \pi \models \varphi_1 \text{ and } \pi \models \varphi_2 \\
\pi \models \neg\varphi & \text{iff} & \pi \not\models \varphi \\
\pi \models \bigcirc\varphi & \text{iff} & \pi_1 \models \varphi \\
\pi \models \varphi_1\, U\, \varphi_2 & \text{iff} & \exists k \geq 0.\, \pi^k \models \varphi_2 \text{ and} \\
 & & \pi^i \models \varphi_1, \text{ for all } 0 \leq i < k
\end{array}$$

On this basis, it is said that a word π *satisfies*, or *modelse*, φ. For a given alphabet an LTL formula defines the languages over words as stated below.

Definition 3.2.3 (LTL — Language)
Let φ be an LTL formula over alphabet $\Sigma = 2^{AP}$. An LTL language induced by φ is

$$\begin{aligned} L_F^{\Sigma}(\varphi) &= \{\, u \in \Sigma^+ \mid u \models \varphi\} \\ L^{\Sigma}(\varphi) &= \{\, \alpha \in \Sigma^{\omega} \mid \alpha \models \varphi\} \end{aligned}$$

where Σ^+ denotes finite alphabet, and Σ^{ω} an infinite alphabet. Superscript Σ if $\Sigma = 2^{AP(\varphi)}$ is omitted.

For an interpretation of an LTL formula over paths and states of a transitions system, the semantics are extended as follows.

Definition 3.2.4 (Semantics — Interpretation over Paths and States)
Let $TS = (S, \rightarrow, L)$ be a transition system, φ be an LTL formula over AP, π be some path of TS, state $s \in S$. Then, drawing on definition 3.2.3 of LTL language, the satisfaction relation follows

$$\begin{array}{llll} \pi & \models \varphi & \text{iff} & trace(\pi) \models \varphi \\ s & \models \varphi & \text{iff} & s \in \pi.\, \pi \models \varphi \\ TS & \models \varphi & \text{iff} & trace(TS) \subseteq L^{\Sigma}(\varphi) \end{array}$$

3.3 Three-valued LTL

The standard LTL semantics provide means to determine whether some *infinite* trace σ satisfies formula φ. In real-world scenarios, only finite traces can be produced and must be interpreted as such. For runtime verification, this implies interpretation of Linear Time (LT) properties on a *finite prefix* of an *infinite* trace. Consequently, and regardless of the fact that LTL semantics for finite traces are formulated, a finite prefix may be insufficient to draw a verdict whether some LT property satisfies an LTL formula. Bauer et al. [2011] propose an extension to standard truth values $\{\top, \bot\}$, with *inconclusive* option, denoted ?, presenting the following argument.

Consider the formula $\neg p\ U\ init$, stating that p should not occur, until *init* function is called. Should p become true before *init* occurs, the formula is violated (for any continuation of current execution). Conversely, should *init* occur and p has not been observed previously, the formula is satisfied, regardless of what happens in the future. In addition to detecting failures, determining whether a property is true or if the current observation is inconclusive is equally critical for testing and verification purposes. An inconclusive observation implies that a violation of the property being checked is still possible [6, p. 5].

LTL$_3$, semantics are thus extended with notion of prefixes Drawing on the original formulation by Kupferman and Vardi, Bauer et al. provide definitions below [35][6, p. 19].

Definition 3.3.1 (LTL$_3$ — Good and bad prefixes)
Let $\Sigma = 2^{AP}$ be an alphabet, Given a language L over Σ^{ω} a finite word $u \in \Sigma^*$ is called a

- *good prefix* for L if for all $w \in \Sigma^{\omega}$, $uw \in L$
- *bad prefix* for L if for all $w \in \Sigma^{\omega}$, $uw \notin L$

Consequently, the satisfaction relation is expanded to include prefixes and the *inconclusive* truth value.

Definition 3.3.2 (LTL$_3$ — Satisfaction)
Given a LTL formula φ and a finite word $u \in \Sigma^*$, then

$$[\ u \models \varphi\ ] = \begin{cases} \top & \text{if } u \text{ is a } \textit{good} \text{ prefix for } L^{\Sigma}(\varphi) \\ \bot & \text{if } u \text{ is a } \textit{bad} \text{ prefix for } L^{\Sigma}(\varphi) \\ ? & \text{otherwise} \end{cases}$$

3.4 Automata-based LTL monitors

Automated procedures for generating monitors for logical formulae can be derived by leveraging a formal connection between logic and automata theory. This process is conceptually similar to the derivation of FA from regular expressions. Bauer et al. [2011] provide a procedure for monitor generation from LTL_3 formulae. For a given LTL_3 formula φ, the resulting FSM is a monitor M^{φ}, able to read finite words $u \in \Sigma^*$ and output a verdict, $[u \models \varphi]$, which is a value in $\mathbb{B}_3$. The procedure is presented and commented on below.

Let $\varphi \in LTL_3$, and NBA_q denote a NBA that coincides with NBA, except for the set of initial states Q_0, which is redefined in NBA_q as $Q_0 = \{q\}$.

1. Define a LTL_3 formula φ and its dual expression $\neg\varphi$
2. Derive $NBA_{\varphi} = (\Sigma, Q^{\varphi}, Q_0^{\varphi}, \delta^{\varphi}, F^{\varphi})$ and $NBA_{\neg\varphi} = (\Sigma, Q^{\neg\varphi}, Q_0^{\neg\varphi}, \delta^{\neg\varphi}, F^{\neg\varphi})$, which accept all words satisfying, and, respectively violating φ. Note that in order to obtain the $\neg NBA$, merely negation of φ is required, rather than the NBA itself.
3. Using *strongly connected components*[1] to define a labelling function $F^{\varphi} : Q^{\varphi} \to \mathbb{B}$ (with $\mathbb{B} = \{\top, \bot\}$), where $F^{\varphi}(q) = \top$ iff $L^{NBA_q^{\varphi}} \neq \varnothing$
4. Using the labelling function from previous step, F^{φ}, derive equivalent $NFA^{\varphi} = (\Sigma, Q^{\varphi}, q^{\varphi}, \sigma^{\varphi}, F^{\varphi})$, where $F^{\varphi} = \{q \in Q^{\varphi} \mid F^{\varphi}(q) = \top\}$, and analogously $NFA^{\neg\varphi}$, where, $F^{\neg\varphi} = \{q \in Q^{\neg\varphi} \mid F^{\neg\varphi}(q) = \top\}$
5. From NFA^{φ} and $NFA^{\neg\varphi}$, derive equivalent DFA^{φ} and $DFA^{\neg\varphi}$
6. Create a product automaton $FSM^{\varphi} = DFA^{\varphi} \times DFA^{\neg\varphi}$ where labelling function is defined by

$$\lambda((q^{\varphi}, q^{\neg\varphi})) = \begin{cases} \top & \text{if } q^{\varphi} \notin F^{\varphi} \\ \bot & \text{if } q^{\neg\varphi} \notin F^{\neg\varphi} \\ ? & \text{if } q^{\neg\varphi} \in F^{\neg\varphi} \text{ and } q \in F^{\varphi} \end{cases}$$

The resulting monitor M^{φ} is a Moore machine, obtained by minimising the FSM^{φ}.

Deriving a BA, which gives a verdict according to an LTL_3 formula, as in step 2, requires dual definition of BA in order to solve the *non-emptiness* problem. Typically, the method developed by Vardi[1996] is applied.[2]. The purpose of functions F^{φ} and $F^{\neg\varphi}$ is to verify that the languages generated by NBA^{φ} and $NBA^{\neg\varphi}$, respectively, in starting state q, are nonempty. If, so it is possible to feed the sub-automaton (NBA_q) a trace which could get accepted. The objective of constructing a product automaton is to simulate the behaviour of feeding an input trace to both DFA_{φ} and $DFA^{\neg\varphi}$ concurrently. Hence, the states in the product automaton correspond to feasible combinations of states from DFA_{φ} and $DFA^{\neg\varphi}$, respectively. The verdict provided by M^{φ} when processing a trace is determined by its labelling function. If the monitor is in a state labelled $\top$ when the trace is being processed, the verdict is true. If the label is $\bot$, the verdict is false. For a label of ?, the verdict is inconclusive.

[1]In graph theory, a strongly connected component of a graph G refers to a subgraph of G in which there exists a path from every pair of nodes n and m. For more details see [51, p. 12, p. 332]

[2]A detailed explanation can be found in [36].

Chapter 4

eBPF

BPF is a technology first developed in 1992 for packet capturing tools. It is a small bytecode language that allows to write filters, for instance which packets should be displayed by `tcpdump`, or received on a raw socket. Its potential was rediscovered in 2013, and a major rewrite was introduced in Linux kernel in 2016, under the name of eBPF, transforming BPF into a general-purpose execution engine. Before presenting eBPF, tracing in Linux — the foundation on which tracing in eBPF is builds upon — deserves attention.

4.1 Tracing in Linux

Tracing refers to an event-based recording technique, that collects and publishes information about program's flow of execution, and analysis of its runtime behaviour. Tracers are special purpose tools, which differ widely in use-cases and applicability. Linux `strace`, for instance, records system call events, while `top` measures events using fixed statistical counters. Tracers generally consist of three components:

• data sources	—	where data comes from
• data collection	—	mechanisms for collection from sources
• data presentation	—	tracing frontends, the tool to interact with to collect or analyse data

Tracers are characterised by their capability to capture raw events and associated metadata. This data can be quite extensive, and often requires further processing to generate useful summaries. Tracing can be either *statically* or *dynamically* instrumented, which refers to the time the tracers are applied to the software. Static tracing implies that the instrumentation points are inserted into the program before its execution, i.e., they are compiled into the programs binary. Dynamic tracing, permits inserting instrumentation points into live software. Both variants come with their own advantages and disadvantages. Static instrumentation ensures *interface stability*, that is, event data is available even if event or binary text symbol changes names or location. Furthermore, it allows instrumentation of functions inlined[1] by the compiler, which may be unavailable otherwise. Dynamic tracing, on the other hand, allows hooking to the target software on-the-fly and provide a larger surface on which tracing can be performed. Comparable to the techniques used by debuggers, the tracing software records information and once finished, relieves control to the target software to continue execution. Dynamic tracing, however, interferes with target system execution and its applicability has been limited due to

[1]Inlining is a manual or compiler optimisation that replaces a function call site with the body of the called function. See, for instance GCC documentation at [31] for more details.

security and safety concerns; modifying instructions live in an address space can be risky as errors can lead immediate corruption and process or kernel crashes.

The Linux kernel ships with an advanced set of tracing facilities, mainly applied in runtime analysis of kernel latencies and performance issues. The most notable is perhaps `ftrace` for kernel function tracing, `perf` which can attach to a multitude of kernel event sources. Tracepoints were the first to be included in the kernel, as of Linux 2.6.32. They serve as a means of kernel static instrumentation, by tracing of calls that have been incorporated into the kernel code at logical locations, which are then compiled into the kernel binary. `kprobes` (kernel-level probes) were introduced in Linux 2.6.9, and `uprobes` (user-level probes) in Linux 3.5, providing dynamic instrumentation capabilities [29, ch. 2.6-2.10] . Introduction of these facilities for data collection paved way for numerous tracing frameworks. fig. 4.1 provides an overview of interaction of these frameworks and facilities.

LTTng is a Linux tracing framework, that has been available since 2006. It shares the same underlying kernel infrastructure with eBPF, relying on krpobes, uprobes, tracepoints and User-level Statically Defined Tracepoint (USDT) for instrumentation. LTTng tracing requires loading of kernel modules.

DTrace is a tracing framework, originally developed by Sun Microsystems (now Oracle) for Solaris, and has since been ported to macOs and FreeBSD. DTrace was one of the first tools to provide scripting capabilities in the kernel. Bpftrace is heavily inspired by DTrace.

SystemTap is the Red Hat's response to DTrace and LTTng, first released in 2009. Similarly to other tracing frameworks, it never made to mainlain kernel [22].

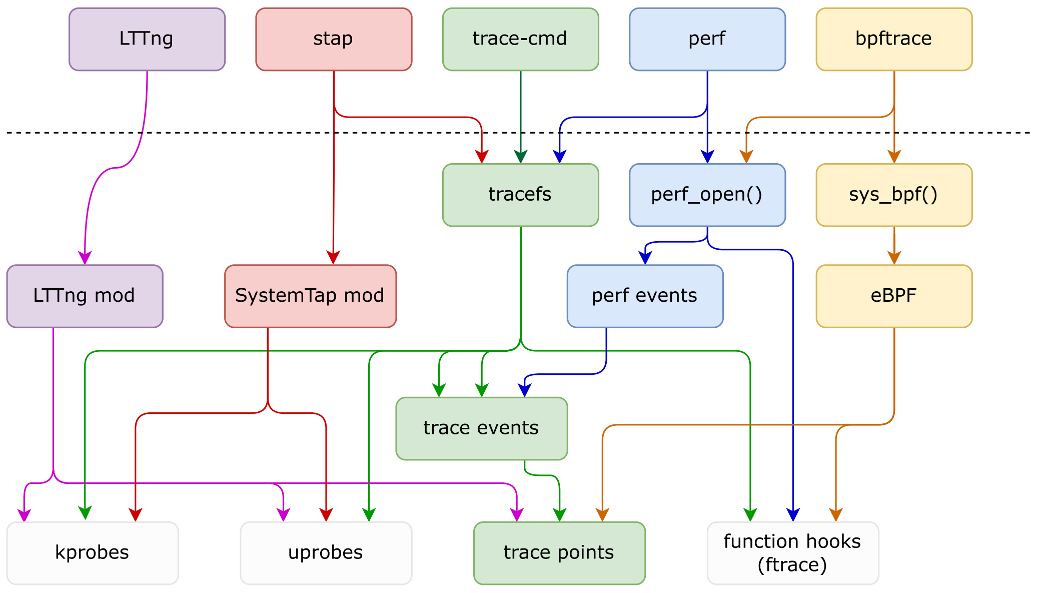


Figure 4.1: Tracing technologies in Linux. Adapted from [22].

4.2 eBPF Origins

eBPF is an emergent technology, already notorious for its versatility. To put it succinctly, it provides a way to mini-programs on kernel and application events. Abbreviations BPF and eBPF are often used interchangeably in contemporary literature. The kernel contains only one instruction set, which runs both classic and extended BPF programs. Before diving into internals, a brief reflection on its motivation eBPF reveals much about its potential.

Consider the nature of Linux Kernel development — an open-source project, consisting of thousands of engineers, each having different interests. Any source code modification will eventually run on millions of devices, from high-end servers to embedded systems. That leads to an interesting scenario, where engineers may have different goals, yet they all must agree that a particular change makes sense for all of the; what may be sensible form some environments, e.g., high-scale server environments, but may be obtrusive in others, e.g., embedded systems. It became exceedingly harder to form a consensus. As a consequence, the discussion evolved to the possibilities of making the kernel programmable and using BPF to do so.

The following sections introduce various aspects of eBPF. The exposition is adapted from [29, ch. 1-2], unless cited otherwise.

4.3 Architecture

Classical BPF was designed to accept filter expressions and pass these to the kernel for execution by an interpreter, as illustrated in fig. 4.2. Execution of user-level filters in kernel makes the need for costly copies to user-level process obsolete, and significantly improves the packet-filtering performance. In pursuit of making eBPF general purpose virtual machine, it has since been extended with more registers, 64-bit words instead of 32, JIT compilation to native instructions, flexible storage, support for probes and tracepoints to name some. The interactions between the many components of eBPF are illustrated in fig. 4.3 and architecture in fig. 4.4.

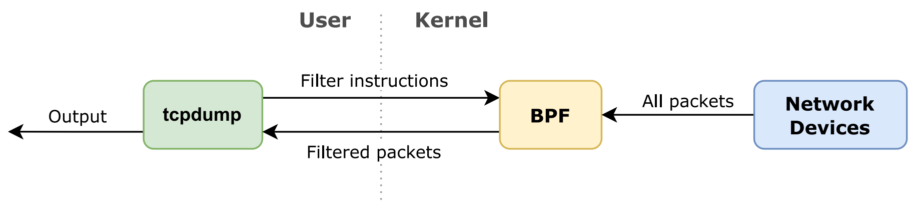


Figure 4.2: Packet filtering in BPF

Written eBPF code is compiled to instructions, often called BPF bytecode, and then passed on to the verifier, which arguably plays the most important role in the eBPF ecosystem. The verifier provides static coverage for given code and ensures that the eBPF code cannot crash nor stall the kernel. Worth noting is that it does not safeguard against malicious or buggy code. Nonetheless, the verification provides safety assurance for the kernel, giving eBPF a significant advantage over Linux kernel modules.

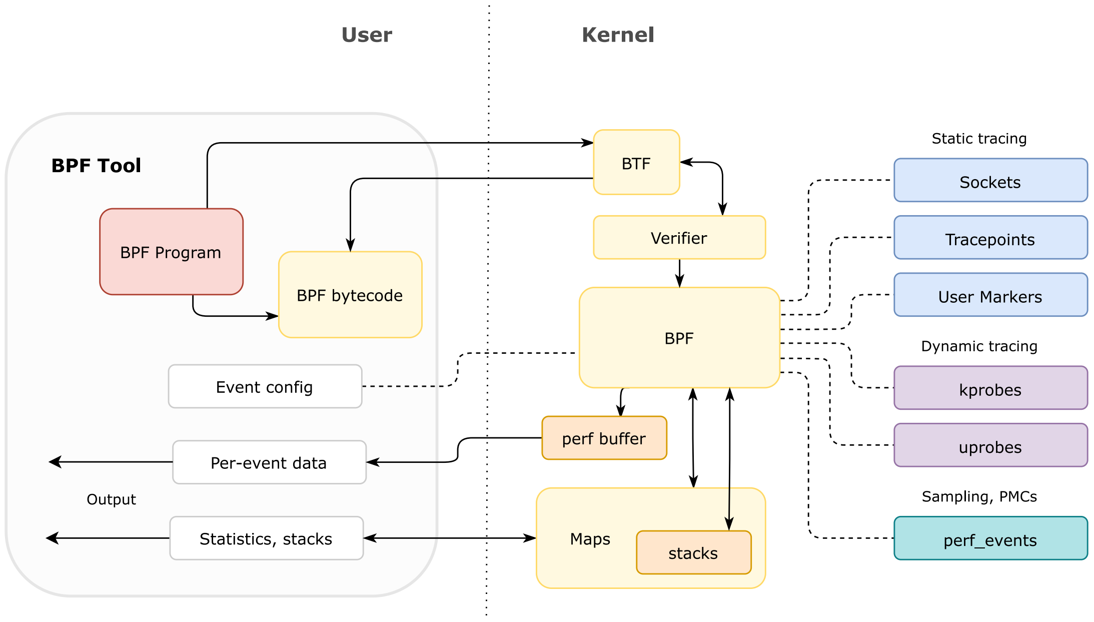


Figure 4.3: eBPF Technologies

Once the code is verified, it is loaded into the kernel for execution. If JIT is enabled, the bytecode is compiled into CPU's native instruction set. From an efficiency perspective, the execution speed is as if the kernel is recompiled with the user space code. Alternatively, the kernel provides an interpreter for the eBPF code. Once loaded, the eBPF program is executed on events — `kprobes`, `uprobes`, and `tracepoints` provided by the kernel interfaces. Probes attach to either the address of a function, thus having access to the function arguments, or the return address of the function called *retprobes*, where it has access to the return value of the function. Probes and their attach addresses are made available by text segment symbol resolution from a binary's Executable and Linkable Format (ELF). These can be listed with tools such as `readelf`, `objdump` or `nm`.

eBPF maps are data structures that reside in kernel space, and are used to keep some form of state among multiple eBPF program invocations. In fact, maps are not *owned* by any eBPF program, but by the kernel itself. Thus, maps can be shared by multiple eBPF programs and provide means for communication and synchronisation. Maps usually contain some sort of application configuration, permitting separation of execution logic and configuration data; map data can be altered without the necessity of recompiling the eBPF program. Consider an eBPF firewall; rules stored in a map can easily be updated, without having to restart the firewall program.

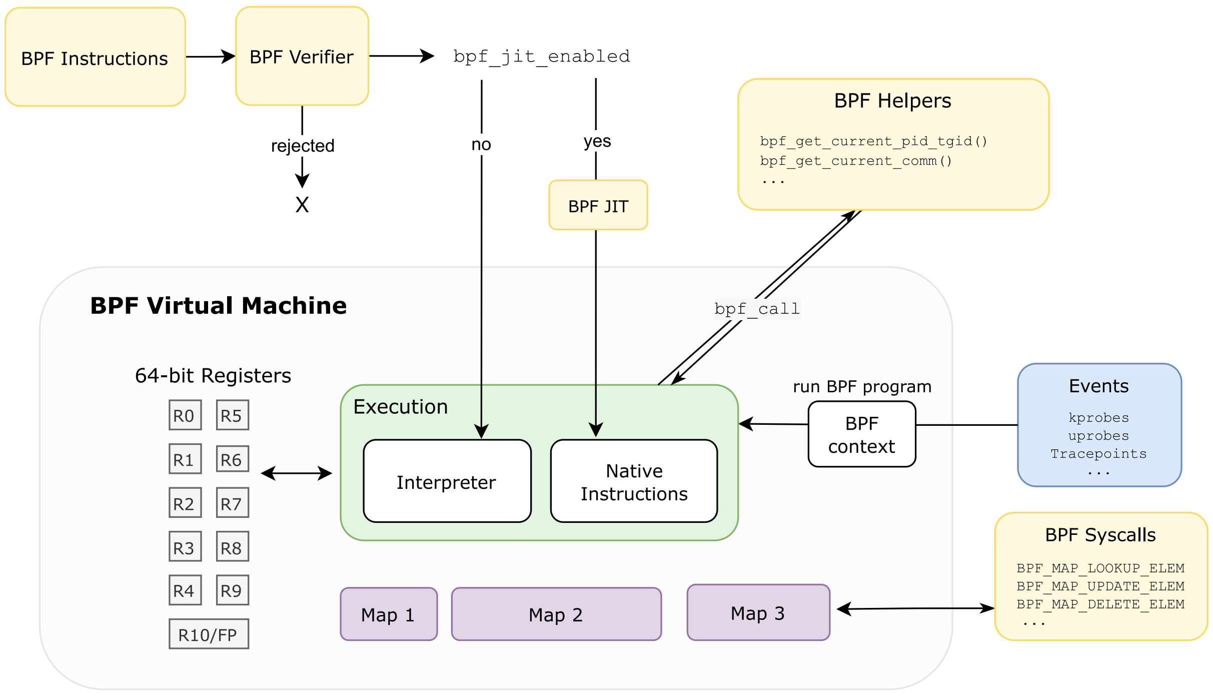


Figure 4.4: eBPF architecture.

BTF

The BPF Type Format (BTF) contains metadata structures that enhance debug information for eBPF programs, maps, and functions. It includes source information that can be used by tools like `bpftool`, introduced in section 5.4, to interpret eBPF data. The eBPF verifier also uses this information to ensure that the structure types defined by the program are correct. BTF metadata is stored in the binary program under a designated '.BFT' metadata section, which significantly increase the size of binary files as it stores type information for all declarations in the program [17, ch. 2].

4.3.1 eBPF vs Kernel Modules

In comparison to Kernel Modules, Greggs lists several advantages of eBPF [29, ch. 2.3.2]. One already mentioned is the verification of programs, to eliminate kernel panics and introduction of possible bugs. Moreover, eBPF provides rich data structures via maps, eBPF programs do not require kernel build artefacts to be compiled, programs can be compiled once and run on different kernel versions, atomic reloading and, finally it's ease of use make it more accessible. In terms of networking, eXpress Data Path (XDP) offers immense performance benefits. XDP is an eBPF based framework, enabling faster response to network operations by executing code and filtering packets directly on the network drivers, rather than copying packets to the kernel to do filtering.

Nonetheless, eBPF has some limitations that should be taken into account. While kernel modules can employ other kernel functions and facilities, eBPF cannot call arbitrary kernel functions and is restricted to the helper functions API. In pursuit of safety, eBPF will only accept loops whose bounds are known at compile time. eBPF programs have a maximum stack limit, at the time of writing set to 512 bytes, and the complexity limit is 1 million instructions. eBPF instruction set is therefore not Turing-complete, as completeness is traded for kernel safety.

4.4 eBPF Frontends and tools

Coding eBPF instructions directly is a tedious endeavour, and various frontends providing higher level abstractions have been developed. The very first of such frameworks, BPF Compiler Collection (BCC), provides a C programming environment for writing kernel eBPF code and other languages for the user-level interface: Python, Lua, C++. It is also the origin of the *libbcc* and *libbpf* libraries. Libbcc provides support for attaching eBPF programs, and libbpf provides facilities for loading eBPF code into the kernel.

bpftrace is a newer frontend, that provides a special purpose language for developing eBPF tools. It offers, concise scripting language excellent for observability and profiling of workloads. Another frontend called *ply* is designed to be lightweight, requiring minimal dependencies, thus suitable for embedded environments. Its syntax closely resembles that of *bpftrace*. Another influential project is BPF Compile Once — Run Everywhere (BPF CO-RE) aims to allow eBPF programs to be compiled to eBPF bytecode once, saved, and then distributed and executed on other systems. The major advantages are avoiding eBPF compilation toolchains installed on target machines, and portability of eBPF code across different kernel versions, of which Nakryiko provides an excellent introduction in [40]. BPF CO-RE leverages recent developments of *libbpf* and BTF, and positions itself as a powerful competitor to BCC by improving resource utilisation, dependency resolution, compatibility, and ease of use.

Many solutions taking advantage of eBPF capabilities are already developed and available for third-party use. Perhaps the most notable is *Cilium*, a Kubernetes CNI solution developed by Isovalent and incubated at CNCF, leveraging the power XDP and eBPF to improve cluster networking security. Another tool is *Tetragon*, also developed by Isovalent, with focus on observability and runtime enforcement. Other notable tools are *Katran*, *Hubble*, for distributed networking solutions, *Falco*, *Pixie*, *Sysmon for Linux*, for cluster observability and anomaly detection, *KubeArmor* for cluster security, to name a few[2].

The opportunities eBPF provides is quite evident, simply judging by the number of novel tools in the sphere of networking, observability and security. Indeed, even Microsoft recognise this potential and is currently working on eBPF for Windows, in order to allow existing eBPF toolchains and APIs in the Linux ecosystem to be run on top of Windows [21].

[2]An exhaustive list is provided by the project *eBPF-Guide*, see [47].

Chapter 5

eBPF Primer

The following section presents a tiny primer for writing eBPF code in C with BPF CO-RE. Both user-level and kernel-level programs are written in C language, and familiarity with C is assumed. Clang 10 or newer is required for compilation. The user-space program is responsible for the lifecycle of the kernel space program. This implies invoking the verifier, loading the program into kernel space, attaching to the desired event, and finally dismounting the program. Once the eBPF program is attached to desired event, it executes upon specified event occurrence.

The following presentation uses probes to instrument `tcp_v4_connect` function, that is an IP4 connect system call. A `kprobe` is attached to function entry, in order to store the address of the socket pointer, whose values are populated during function execution. Similarly a `kretprobe` is attached to the return value of the function in order to inspect the populated socket. Programs shown below are based on BCC inspection tool `tcpconnect` adapted to BPF CO-RE [32]. The presentation is an adaption from Nakryiko's excellent introduction to BPF CO-RE in [41].

5.1 eBPF Side

Listings 1 and 2 show an example of the eBPF code. It is relevant to note that the program is operating in kernel context, which implies that standard C libraries, and other system-wide types, tools, and libraries are unavailable. The first included header `vmlinux.h` is the heart of BPF CO-RE, as it provides all the Linux kernel type definitions, and deals with compatibility issues across kernel versions. `vmlinux.h` is generated by the `bpftool`, see section 5.4. Header `bpf/bpf_helpers.h` provides basic eBPF related types, constants and helper functions necessary for interaction with kernel-side eBPF APIs. Headers `bpf/bpf_tracing.h` and `bpf/bpf_core_read.h` provide some extra macros and functions for writing BPF CO-RE-based programs. The following `LICENCE` variable defines the licence of the eBPF program and is mandatory. The `SEC` helper assigns variables and functions into specific ELF sections. These conventions are mandated by libbpf.

A standard C `struct` is defined on line 16, and in turn passed as a data type argument to eBPF map defined on line 26. Using macros and functions from included headers, a hash map, with maximum size of 64 entries, key of type `u32`, value of type `struct sock *`, and with the name `currsock`, is created. The annotation `SEC(".maps")` is required to create a BTF-defined map in the kernel and wire everything appropriately in the eBPF code. A local function for byte swapping[1] is defined on line 33. All local helper functions must be defined as `static inline`, otherwise the compiler cannot know how to handle functions that are not attached to any events.

eBPF programs that will be loaded into the kernel are defined on lines 37 and 47, represented as more or less regular C functions in specially named sections. In reality, eBPF programs accept a single pointer to `struct pt_regs* context` argument. `context` assumes different values depending on the

[1]Internet protocols mandate a *network byte order* to facilitate communication among machines with varying byte order conventions when transmitting data over the network. Amd64 uses little endian order.

```c
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>
#include <bpf/bpf_core_read.h>

char LICENSE[] SEC("license") = "Dual BSD/GPL";

#ifndef TASK_COMM_LEN
#define TASK_COMM_LEN    16
#endif

#ifndef AF_INET
#define AF_INET            2
#endif

struct ipv4_data_t {
    u64 ts_us;
    u32 pid;
    u32 saddr;
    u32 daddr;
    u16 lport;
    u16 dport;
    char task[TASK_COMM_LEN];
};

struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 64);
    __type(key, u32);
    __type(value, struct sock *);
} currsock SEC(".maps");

static inline int _bswap16(u16 val) {
    return ((val & 0x0ff) << 8) | ((val & 0xff00) >> 8);
}

```

Listing 1: `tcpconnect.bpf.c` — kernel-space code with BPF CO-RE (1)

specific event type to which the eBPF program is attached, and is directly handled by eBPF instruction set. However, the macro `BPF_KRPOBE` permits users to pretend like traced function's arguments are available directly. The first argument is the name of the eBPF program, followed by arguments to the function that is instrumented. The `trace_tcp_v4_connect_entry` function firsts emits a formatted string, i.e., a debug message, using the helper `bpf_printk`, to a special file, which can be viewed with the `sudo cat /sys/kernel/debug/tracing/trace_pipe` command. The the process id is retrieved using the helper `bpf_get_current_pid_tgid`, and finally the map `currsock` is updated with the helper `bpf_map_update_elem`, using the address of `tid` variable as entry key, and the address of the socket pointer `sk` as entry value. The `trace_tcp_v4_connect_ret` retrieves the address of socket pointer `sk`, first by retrieving process id for the key, and then executing a map lookup using the function `bpf_map_lookup_elem`. Lines 69 to 72 copy data from values external to eBPF program and

```c
SEC("kprobe/tcp_v4_connect")
int BPF_KPROBE(trace_tcp_v4_connect_entry, struct sock *sk)
{
    bpf_printk("tcp_v4_connect probe");
    u32 tid = bpf_get_current_pid_tgid();
    bpf_map_update_elem(&currsock, &tid, &sk, BPF_ANY /* flags */);

    return 0;
}

SEC("kretprobe/tcp_v4_connect")
int BPF_KPROBE(trace_tcp_v4_connect_ret)
{
    bpf_printk("tcp_v4_connect retprobe");

    char task[TASK_COMM_LEN];
    bpf_get_current_comm(&task, sizeof(task));

    u64 pid_tgid = bpf_get_current_pid_tgid();
    u32 pid = pid_tgid >> 32;
    u32 tid = pid_tgid;

    struct ipv4_data_t data4 = {
        .pid = pid,
        .ts_us = bpf_ktime_get_ns() / 1000,
    };

    struct sock **skpp = bpf_map_lookup_elem(&currsock, &tid);
    if (!skpp)
        return -1;

    struct sock *skp = *skpp;
    data4.lport = BPF_CORE_READ(skp, __sk_common.skc_num);
    data4.saddr = BPF_CORE_READ(skp, __sk_common.skc_rcv_saddr);
    data4.daddr = BPF_CORE_READ(skp, __sk_common.skc_daddr);
    u16 dport = BPF_CORE_READ(skp, __sk_common.skc_dport);
    data4.dport = _bswap16(dport);

    bpf_probe_read_str(&data4.task, sizeof(data4.task), &task);

    /* Do something with acquired data */

    return 0;
}
```

Listing 2: `tcpconnect.bpf.c` — kernel-space code with BPF CO-RE (2)

helper functions, to it's local variables. The eBPF verifier rejects direct copies[2] due to the risk of reading arbitrary kernel memory. The macro `BPF_CORE_READ` is supplied by `bpf_core_read.h`, takes care of this in a succinct manner — using regular helper `bpf_probe_read` would require a function call for each pointer in a nested structure, while pointer structures can be easily 'disassembled' in the same

[2]Such as `data4.lport = skp->__sk_common.skc_num;`.

BPF_CORE_READ function call. Furthermore, it records necessary BPF CO-RE re-locations along the way, allowing libbpf[3] to adjust all the field offsets to the specific memory layout of the host kernel.

5.2 User-space Side

Listings 3 and 5 illustrate how eBPF is handled on the user-space side. The eBPF kernel-space code is provided to the user-space as a eBPF skeleton through the tcpconnect.skel.h header, which is generated by bpftool in a Makefile step.

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
#include <sys/resource.h>
#include <bpf/libbpf.h>
#include "tcpconnect.skel.h"

static int libbpf_print_fn(enum libbpf_print_level level, const char *format, va_list args)
{
    return vfprintf(stderr, format, args);
}

static volatile sig_atomic_t stop;

static void sig_int(int signo)
{
    stop = 1;
}

```

Listing 3: tcpconnect.c — user-space loading with libbpf (1)

The skeleton facilitates structuring and deployment by embedding contents of the compiled eBPF object code in a header file. It is a purely libbpf construct, i.e. the kernel is entirely unaware of the skeleton, providing a quality of life improvement for the development process. Consider the high-level overview provided in listing 4. A struct bpf_object *obj is defined, followed by structs for maps, programs, and links sections, which provide access to eBPF objects defined in the kernel-space code. These references can be passed to libbpf APIs to execute some desired operations. bss, data, and rodata structs allow for direct access (no extra system calls required) to eBPF global variables from user-space. As the only global variables used in this example is the map currsock, only references to it are provided. Finally, functions for interaction with libbpf API for opening, loading, attaching and destroying the eBPF object code are declared, and subsequently defined.

Now onto the main function, listing 5. After declaring the skeleton struct, and a variable for error handling, the function libbpf_set_print function provides a custom callback for all libbpf logs, particularly useful during development. By default, it will only log error level messages. Then, using the provided functions from the skeleton header, the ebpf program is opened and loaded into the kernel and passed on to the verifier. Upon successful verification, the program, by now readily waiting in the kernel, is attached to the specified instrumentation point on line 39. This ‘activates’ the eBPF program and the kernel will start executing specified programs for each tcp_v4_connect function call and it’s return

[3]Loading library, see section 5.2.

```
/* SPDX-License-Identifier: (LGPL-2.1 OR BSD-2-Clause) */

/* THIS FILE IS AUTOGENERATED BY BPFTOOL! */
#ifndef __TCPCONNECT_BPF_SKEL_H__
#define __TCPCONNECT_BPF_SKEL_H__

#include <errno.h>
#include <stdlib.h>
#include <bpf/libbpf.h>

struct tcpconnect_bpf {
    struct bpf_object_skeleton *skeleton;
    struct bpf_object *obj;
    struct {
        struct bpf_map *currsock;
        struct bpf_map *rodata;
    } maps;
    struct {
        struct bpf_program *trace_tcp_v4_connect_entry;
        struct bpf_program *trace_tcp_v4_connect_ret;
    } progs;
    struct {
        struct bpf_link *trace_tcp_v4_connect_entry;
        struct bpf_link *trace_tcp_v4_connect_ret;
    } links;

#ifdef __cplusplus
    static inline struct tcpconnect_bpf *open(const struct bpf_object_open_opts *opts = nullptr);
    static inline struct tcpconnect_bpf *open_and_load();
    static inline int load(struct tcpconnect_bpf *skel);
    static inline int attach(struct tcpconnect_bpf *skel);
    static inline void detach(struct tcpconnect_bpf *skel);
    static inline void destroy(struct tcpconnect_bpf *skel);
    static inline const void *elf_bytes(size_t *sz);
#endif /* __cplusplus */
};
...
```

Listing 4: `tcpconnect.skel.h` generated by `bpftool`

instruction. libbpf is able to automatically determine where to attach the program by the provided `SEC` section. Signal handling for program interruption by `<Ctrl-C>` is provided on line 45, followed by an endless loop which ensures that the eBPF program stays attached, until killed by the user, that is.

```c
int main(int argc, char **argv)
{
    struct tcpconnect_bpf *skel;
    int err;

    libbpf_set_strict_mode(LIBBPF_STRICT_ALL);
    /* Set up libbpf errors and debug info callback */
    libbpf_set_print(libbpf_print_fn);

    /* Open load and verify BPF application */
    skel = tcpconnect_bpf__open_and_load();
    if (!skel) {
        fprintf(stderr, "Failed to open BPF skeleton\n");
        return 1;
    }

    /* Attach tracepoint handler */
    err = tcpconnect_bpf__attach(skel);
    if (err) {
        fprintf(stderr, "Failed to attach BPF skeleton\n");
        goto cleanup;
    }

    if (signal(SIGINT, sig_int) == SIG_ERR) {
        fprintf(stderr, "can't set signal handler: %s\n", strerror(errno));
        goto cleanup;
    }

    printf("Successfully started! Please run `sudo cat /sys/kernel/debug/tracing/trace_pipe` "
           "to see output of the BPF programs.\n");

    while (!stop) {
        fprintf(stderr, ".");
        sleep(1);
    }

cleanup:
    tcpconnect_bpf__destroy(skel);
    return -err;
}
```

Listing 5: `tcpconnect.c` — user-space loading with libbpf (2)

5.3 Makefile

With enough context of the example code, a short peek into how the Makefile compiles everything into a final executable is due. Familiarity with Makefile syntax and process is assumed, and only relevant parts are shown. The build process happens in a few steps. Firstly, if libbpf and bpftool are added as project dependencies, building them initially is required. These steps can be omitted if system-wide libbpf and bpftool used for compilation, in which case the path to libbpf static library must be provided for subsequent compilation of eBPF code.

```
INCLUDES := -I$(OUTPUT) -I $(LIBBPF_SRC)/include
LIBBPF_SRC := /some/path/to/libbpf/src

# Build libbpf
$(LIBBPF_OBJ): $(wildcard $(LIBBPF_SRC)/*.[ch] $(LIBBPF_SRC)/Makefile) | $(OUTPUT)/libbpf
	$(call msg,LIB,$@)
	$(Q)$(MAKE) -C $(LIBBPF_SRC) BUILD_STATIC_ONLY=1                    \
	            OBJDIR=$(dir $@)/libbpf DESTDIR=$(dir $@)               \
	            INCLUDEDIR= LIBDIR= UAPIDIR=                            \
	            install

# Build bpftool
$(BPFTOOL): | $(BPFTOOL_OUTPUT)
	$(call msg,BPFTOOL,$@)
	$(Q)$(MAKE) ARCH= CROSS_COMPILE= OUTPUT=$(BPFTOOL_OUTPUT)/           \
	    -C $(BPFTOOL_SRC) bootstrap
```

Listing 6: Makefile: Building libbpf and bpftool

Recall that the header `vmlinux.h` was included to kernel-space code to provide necessary BTF. The following steps locates system-wide BTF definitions, and generate a C header from this information with `bpftool`.

```
# Try to detect best kernel BTF source
VMLINUX_BTF_PATHS := /sys/kernel/btf/vmlinux /boot/vmlinux-$(KERNEL_REL)
VMLINUX_BTF_PATH := $(or $(VMLINUX_BTF),$(firstword                        \
                                          $(wildcard $(VMLINUX_BTF_PATHS))))


$(OUTPUT)/vmlinux.h: $(VMLINUX_BTF_PATH) | $(OUTPUT)
	$(Q)$(call msg,GEN,$@)
	$(Q)if [ ! -e "$(VMLINUX_BTF_PATH)" ] ; then \
		echo "Couldn't find kernel BTF; set VMLINUX_BTF to"              \
			"specify its location." >&2;                                 \
		exit 1;\
	fi
	$(Q)$(BPFTOOL) btf dump file $(VMLINUX) format c > $@

```

Listing 7: Makefile: Find or generate `vmlinux.h`

Compilation of eBPF code is executed with `clang`. `-g` is necessary for Clang to emit BTF information, and `-O2` is necessary for eBPF compilation. Macros for `bpf_tracing.h` header, handling low-level `struct pt_regs` required for probes, is defined by `-D__TARGET_ARCH_$(ARCH)`. Finally, `DWARF`[4] symbols are stripped from the generated object file, as it is not used in this context.

```
# Build BPF code
$(OUTPUT)/%.bpf.o: %.bpf.c $(LIBBPF_OBJ) $(wildcard %.h) | $(OUTPUT)
	$(call msg,BPF,$@)
	$(Q)$(CLANG) -g -O2 -target bpf -D__TARGET_ARCH_$(ARCH)                \
	    $(INCLUDES) $(CLANG_BPF_SYS_INCLUDES) -c $(filter %.c,$^) -o $@
	$(Q)$(LLVM_STRIP) -g $@ # strip useless DWARF info
```

Listing 8: Makefile: Building eBPF kernel code

Subsequently, bpftool is used to generate the skeleton header file, using the `bpftool gen skeleton` command. Along with it, whenever the eBPF skeleton is updated, user-space application parts are rebuilt as well, as they embed eBPF skeleton during compilation. Once user-space object files are built, they are used together with the `libbpf.a` static library, the final binary is generated. `-lelf` and `-lz` are libbpf dependencies and must be provided explicitly. Invoking the compiled binary is as simple as `sudo ./tcpconnect`.

```
# Generate BPF skeletons
$(OUTPUT)/%.skel.h: $(OUTPUT)/%.bpf.o | $(OUTPUT) $(BPFTOOL)
	$(call msg,GEN-SKEL,$@)
	$(Q)$(BPFTOOL) gen skeleton $< > $@

# Build application binary
$(APPS): %: $(OUTPUT)/%.o $(LIBBPF_OBJ) | $(OUTPUT) $(BIN_OUTPUT)
	$(call msg,BINARY,$@)
	$(Q)$(CC) $(CFLAGS) $^ $(ALL_LDFLAGS) -lelf -lz -o $(BIN_OUTPUT)/$@
```

Listing 9: Makefile: Generate eBPF skeleton and build application binary

[4] DWARF is a debugging information file format used by many compilers and debuggers to support source level debugging.

5.4 Bpftool

`bpftool` is a command-line tool facilitating interaction with the eBPF subsystem and, essentially, acts as a Swiss Army knife for managing eBPF objects by allowing inspection and simple modification [9]. It provides commands to load and unload eBPF programs, query maps, attach programs to kernel subsystems, generate eBPF bytecode from high-level languages and debug eBPF programs. Note that `bpftool` does not implement low-level eBPF handling itself, but acts as an interface to eBPF virtual machine by relying on libbpf. Some examples are presented below.

- `bpftool prog` – Subcommand for loading and unloading eBPF programs. Option `-d` unloads a program, and option `-l` lists all loaded programs [14].
- `bpftool prog attach` – Subcommand for attaching eBPF programs to kernel subsystems. Option `-p` specifies eBPF program to attach, and with option `-t` may the type of kernel subsystem be specified [14].
- `bpftool prog dump` – Subcommand for dumping contents of eBPF program's bytecode for debugging purposes. Option `-j` outputs the bytecode in JSON format, and with option `-f` may the output file be specified [14].
- `bpftool map` – Subcommand for managing eBPF maps. Option `-D` deletes a map, and the option `-l` lists all maps [13].
- `bpftool-gen` – Subcommand for generating eBPF programs, i.e. eBPF bytecode, from high-level languages such as C, Python or Lua. Option `-p` is used to specify the language of the input file and the `-o` to specify the output file [12].
- `bpftool-btf` – Subcommand for manipulating BTF data, for instance generating BTF data from C header files, merging BTF from multiple files, and dump contents for debugging purposes [11].

Part II

Thesis

Chapter 6

Dottobpf

Thus far, pieces of the runtime verification puzzle have been presented — specification and assertion of system properties in LTL_3, automata-based monitor derivation from respective formulae, and the eBPF tracing facility in Linux. The remaining piece, and the objective of this thesis, is generation of eBPF based monitors from LTL derived automata. This section introduces `dottobpf`, a command-line tool solving this specific task. Performing runtime verification with `dottobpf` assumes the following workflow:

1. Specification of system properties in LTL_3.
2. Translation of LTL_3 formula into equivalent automaton (using a third-party tool).
3. Writing a configuration file with respect to the automaton.
4. Generate eBPF monitors with `dottobpf` from provided specification.
5. Compile generated sources and execute the compiled program.

6.1 Related Work

`graphviz2dtrace` is a tool developed by Rosenberg, which generates monitors conforming LTL_3 semantics, for the DTrace tracing framework. Work done by Rosenberg in his thesis served as a great inspiration for `dottobpf` [46].

`dot2c` is a tool developed by Bristot, generates a representation of DFA models in C from the automata provided as a DOT graph. In combination with the tool `dot2k` by same author, skeletons for a kernel monitor are created, relying on loading the monitor as kernel modules [44].

`dot2bpf`, with an uncannily similar name as the product of this thesis, is a tool leveraging `dot2c` for automata generation in C in order to generate skeletons for eBPF monitors [43]. The tool does, effectively, solve the same problem as this thesis. Differences in approach and functionality are discussed in section 10.3.

6.2 Technology evaluation

In order to provide a solid foundation for presentation of `dottobpf` design and implementation, it is worth taking a moment to consider necessary technologies for the intended workflow, as well as the rationale behind these choices.

6.2.1 Logic Translators

Given the compelling argument for LTL_3 presented in section 3.3, a functional requirement for `dottobpf` to generate monitors that are fully capable of supporting LTL_3, was established. The requirement is accordingly extended to the logic translation tool. Two options were evaluated: *LamaConv* and *LTLf2DFA*. LamaConv —*Logics and Automata Converter Library*, written in Java, is capable of translating temporal logic expressions into equivalent automata and generate respective monitors [49]. *LamaCong* supports numerous logics, among these LTL and LTL_3, thereby satisfying the necessary requirement.

LTLf2DFA, a Python library, translates LTL and Past-time LTL (PLTL) formulae into equivalent DFA using MONA [25][33]. Written in python, it offers the potential for seamless integration with `dottobpf`, relieving the user of reliance on multiple tools.

As the reader may recall from section 3.4, automata LTL monitoring relies on synthesis of Moore machines. Achieving this with LTLf2DFA would require implementing extensions for this task, ultimately out of scope of this thesis. Failing to meet the necessary requirements, leaving LamaConv as the only valid alternative.

6.2.2 eBPF frontend

As introduced previously, various frontends for interaction with eBPF are readily available. Relying one of the more prominent projects, namely bcc, bpftrace or bpf core, was initially decided, simply due to their greater reach within the community thus facilitating research and debugging throughout the thesis. Among these, BCC was quickly discarded due to development of performance being considered deprecated [28]. Bpftrace and BPF CO-RE rely on different philosophies; while bpftrace provides a concise and quick, yet powerful tracing facility, BPF CO-RE maintains portability as a driving objective, enabling distribution of eBPF programs as tiny binaries compatible across different kernels. Both technologies have been closely examined throughout the thesis, yet ultimately, the choice for generated code fell upon BPF CO-RE. Partly, due to BPF CO-RE perceived extensibility and, partly due to being a new technology of personal interest. Moreover, bpftrace syntax closely resembles that of DTrace, and simple edits to `graphviz2dtrace` would make bpftrace script generation possible. The desire to make a novel contribution led to a preference for BPF CO-RE.

6.2.3 Implementation language

Regarding the programming language for the implementation of `dottobpf`, usability, from the perspective of end user, was driving the evaluation. In pursuit of seamless integration with LamaConv, reducing the number of dependencies for the assumed workflow, an implementation in Java was entertained. Given the lack of documentation on interfacing LamaConv (other than that from the command line), this prospect was abandoned in favour of Python. The decision prompted by its accessibility in Linux environments and simplicity of use.

6.3 Design principles

In broad strokes, programs generated by `dottobpf` should perform following tasks:

1. Gather data about SUS
2. Feed the obtained trace to the monitor
3. Yield a verdict
4. Inform user

In achieving these, deliberation about `dottobpf` design and requirements is proper. Firtly, given that eBPF provides the lens through which the SUS is observed, the first crucial question arises as to how to establish a meaningful connection between observations made and the LTL formula. More precisely, what should atomic propositions represent? *dottobpf* addresses this by:

Associating atomic properties with elements observable by eBPF event sources.

A system's trace is obtained by use of eBPFs numerous event sources. For instance, in the case of `[u|k]probes`, an event source is a traceable function, that is, whose function symbol is represented in SUS' ELF. The occurrence of the event, the observable element by eBPF, may be associated to an atomic proposition, i.e. proposition is true when the event triggers. Furthermore, consider the fact that probes allow for inspection of target function's arguments, or static tracepoints allow inspection of some published data. Then, being observable elements provided by eBPF event sources, the possibility to predicate the association of atomic propositions based on some value arises, i.e. proposition is *true* if `data == x` *when* the specific event fires. In effect, users are able to analyse the SUS by specifying predicated events of interest, and reasoning about these using temporal logic. The consequence of this approach, however, is the limitation of properties users can reason about, imposed by the technical restrictions of eBPF.

Secondly, eBPF programs are characterised by their ephemeral nature - the program is executed once an event triggers and stack wiped, upon relinquishing control to SUS, and thus unable to store any data persistent across invocations themselves. The need for a globally persistent structure to store state values becomes apparent, and raises questions about how the automata should be represented. In order to limit the scope of both the tool and thesis, the following limit is placed:

An invocation of `dottobpf` *can only generate a monitor for a single model.*

Returning to persistent data, eBPF provides two alternatives; global static variables or eBPF maps. Static variables are naturally more efficient than maps, firstly, by initialisation, and secondly, by fewer number of operations required to read and write values. This approach, however, introduces hard constraints. Firstly, it requires all eBPF programs to be written in the same translation unit, i.e. the same file, in order for all of them to access the statically defined variable – thereby limiting the possibility of decentralised architecture. Secondly, in the case of parameterising automata, reliance on static variables for state storage inevitably leads to scalability issues. Entertain the thought of having to store, say 1000 state variables, each representing a running automaton, and the issue becomes apparent. eBPF maps are much more flexible in this regard. Having settled on maps for storage of state values, should the entire automata definition be stored in maps as well? Consider the remaining elements of automata; sets of states and propositions, the initial state, the transition function, and possibly rejecting and accepting states. None of these are written to after initialisation, and can thereby be regarded as 'read only' properties. Scalability of these elements, need not be addressed due to the fact the monitor generated by `dottobpf` can only contain a single model - any parameterised automata reference the same model. As an example, consider the case of verifying the an LT property across multiple instances of automata. While the trace of each instance certainly may differ, all of them should satisfy the same model. The state is only a product of their trace at certain point in time, while remaining model elements are constant. Therefore maintaining a state value parameterised (by unique object identifier), while sharing constant data, provides better utilisation of system resources and reduces the overall memory footprint of the application. Consequently, leading to following decisions:

Separate state value data from the automaton structure.

A final consideration must be made on how to inform the user of the current state and verdict on specified properties of SUS. Available alternatives are: printing to system tracepipe, publishing data on tracepoints, or communicating data to user-space by use of ring buffers. Printing information to the system tracepipe utilises `bpf_printk` function call and and is considered more as a debug utility rather than an operational one. Publishing data on tracepoints is certainly a viable solution, and one employed by the `dot2k` and `dot2bpf`. The approach, however, requires of users to retrieve published data with facilities such as `ftrace` or `perf`. By employing ring buffer, on the other hand, eBPF kernel programs are able to publish data to the buffer, while the user-space programs are able to poll the data and handle it as desired. Oriented towards usability, the design decision was to

Communicate data to user-space with ring buffers.

6.4 Implementation

`dottobpf` essentially functions as a source-to-source compiler, invoked with the representation of the model the in DOT language, along with a configuration file necessary information for eBPF events. The DOT format is open and widely used to represent graphs and automata, and for example, is used in tools acknowledged in section 6.1. The rationale for separation of model representation and related eBPF configuration, is due to the inconvenience of encoding associations in the DOT file itself. LamaConv will, in fact, throw syntax errors on strings containing special characters. Bearing in mind that eBPF programs are written as C functions, the configuration provides better expressiveness to specify necessary function arguments and allowing for specification of predicates. In that manner, `dottobpf` attempts to limit the need for any manual adjustments required by the user. However, such a need may still arise as will be discussed in chapter 8.

Upon invocation of `dottobpf`, the file representing the automaton is converted into a combination of C structs and an eBPF map, in line with the predetermined design decision. This conversion process relies on the automata representation generated by LamaConv, and specifically addresses certain artefacts induced by the tool. One such artefact is a "hidden node" with a complementary ‘START’ edge, visually indicating the initial state of the automaton. Once the starting node is located and the initial state of the automaton is established, these artefacts are removed. Another LamaConv artefact requiring processing are edges marked with a ‘?’, here referred to as *wildcard edges*, and indicate all possible events. Typically, these wildcard edges lead to a sink state —a state with no outgoing edges — and allow for the identification of possible rejecting or accepting states. However, in case of a wildcard edge leading to another node, `dottobpf` raises a value error as it is currently unable to handle such events. This processing should be rightfully attributed to [46].

6.4.1 Automaton

All necessary code regarding the model are generated in the `automaton.bpf.h`. The states and propositions of the automaton, as well as the verdict on the provided trace, are defined using `enum type state`, `event` and `verdict`, respectively. These provide useful identifiers for clarity and ease of use. The `struct automaton` contains remaining automaton properties - states and events are omitted due to already being encoded as enums. The decision to define the state, proposition, and verdict enums separately from the automaton, is influenced by the combination of the design decision to use a ring buffer for communication with user-space and the motivation for greater usability. While the DOT file uses regular strings to represent states and events, enums in C are essentially `int` types, and presenting state and event information as integers to users requires manual lookup of values to make sense. Therefore, in order to enhance usability, enum type definitions are extracted to a separate header, which is included in both user-space and kernel code. The kernel code references enum identifiers for event handling and transmits these values to user-space. The user-space, in turn, references a lookup table to obtain the names of states and events as represented in the DOT file prior to presenting the information to user.

With `dottobpf`, two options for constructing automata transition functions are available: a static nested array in C or an eBPF hashmap. The latter option offers a unique advantage, by allowing dynamic

```c
#ifndef __COMMON_H
#define __COMMON_H

#ifndef TASK_COMM_LEN
#define TASK_COMM_LEN 16
#endif /* TASK_COMM_LEN */

typedef enum e_state { q2 = 0, q0, q1, state_max } state;

typedef enum e_proposition { empty = 0, push, pop, prop_max } proposition;

typedef enum e_verdict { inconclusive = 0, reject, accept, erdict_max } verdict;

struct event {
    int pid;
    char comm[TASK_COMM_LEN];
    proposition prop;
    state state;
    verdict verdict;
    int id;
};
#endif /* __COMMON_H */
```

Listing 10: Generated `common.h` for case study presented in chapter 7.

manipulation of transition functions during runtime, without requiring recompilation. This approach aims to provide a degree of operational flexibility while performing runtime verification. It should be noted, however, that this flexibility is limited to the transition function only, as other automaton data is stored as static values. Recognising that this level of flexibility may not be necessary in all use cases, and striving to improve performance, an alternative option utilising a static array is also provided. Both alternatives index the transition function in constant time $O(1)$. Listings 10 and 11 show how the generated code might look. The hash map utilised for the storage of state values, aptly named `state_map`, specifies its key type as an integer in the provided listing 3. Hashmaps allow for convenient parameterisation of automata, by allowing users to specify key types and assign values. This is further discussed in chapter 9.

```c
#ifndef __AUTOMATON_H
#define __AUTOMATON_H

#include "common.h"

typedef struct automaton_s {
    state rejecting_states[1];
    state initial_state;
    state tf[state_max][prop_max];
} automaton_t;

static automaton_t aut = {
    .rejecting_states = { q0 },
    .initial_state = q2,
    .tf = {
        { q2, q1, q2 },
        { q2, q1, q0 },
        { q0, q0, q0 }
    }
};

struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 64);
    __type(key, int);
    __type(value, state);
} state_map SEC(".maps");

struct {
    __uint(type, BPF_MAP_TYPE_RINGBUF);
    __uint(max_entries, 256 * 1024);
} rb SEC(".maps");

static inline verdict get_verdict(state state) { ... }

static inline void handle_verdict(verdict vd, int* entry_id) { ... }

static inline void submit_rb_event(state* state, proposition* prop,
                                   verdict* verdict, int* entry_id) { ... }

static inline state update_state(int* entry_id, proposition* prop) { ... }

#endif /* __AUTOMATON_H */
```

Listing 11: Generated `automaton.bpf.h` for case study in chapter 7. The example uses C static array for the transition function.

Should eBPF maps be utilised for the transition functions `dottobpf` produces an `aut_tfkey_s struct` which includes both state and event values. The hash of each `aut_tfkey` instance serves as a key, allowing for the retrieval of the resulting state for any given combination. The map itself is initialised only when the first event is triggered, should it not have been initialised already.

```c
typedef struct aut_tfkey_s {
    state state;
    proposition prop;
} aut_tfkey;

static automaton_t aut = {
    .rejecting_states = { q0 },
    .initial_state = q2,
    .tf_inited = 0
};
...
static inline void init_aut_transition_function(u8* tf_inited)
{
    if (*tf_inited != 0) return;
    aut_tfkey tfkey;
    state res;

    tfkey.state = q0;
    tfkey.prop = empty;
    res = q0;
    bpf_map_update_elem(&tf, &tfkey, &res, BPF_ANY);

    tfkey.prop = push;
    res = q0;
    bpf_map_update_elem(&tf, &tfkey, &res, BPF_ANY);
    ...
    *tf_inited = 1;
};
```

Listing 12: Generated `automaton.bpf.h` using hashmap for storing the transition function.

6.4.2 eBPF Programs

Once the automaton is processed and constructed, the eBPF programs are created by primarily utilising the information specified in the configuration file. The eBPF programs are generated in separate source file(s). Functionality to interface the automaton is provided by included automaton header. These eBPF programs attach to user-provided event sources, assert propositions possibly, based on predicates, and subsequently update the state, obtain its verdict, and submit event data to the ring buffer (`rb` in the listing 11).

In order to parameterise the automata, users can specify which key should be used for retrieval of the correct element from the `state_map`. Should users omit to do so, a standard key `int entry_id = 0` is provided. Each generated perform simple stats as shown in listing 13. If eBPF programs are required to perform additional logic, this must be manually added by the user.

```
#include "vmlinux.h"
#include <bpf/bpf_core_read.h>
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>

#include "common.h"
#include "automaton.bpf.h"

char LICENSE[] SEC("license") = "Dual BSD/GPL";

SEC("uretprobe//usr/local/bin/stack:empty")
int BPF_KRETPROBE(handle_empty, int retval)
{
    if (!(retval == 1)) return 0;
    proposition prop = empty;

    /* WARNING - the solution below provides only one value as a key
    which implies it can only keep state of a single object */
    int entry_id = 0;
    state new_state = update_state(&entry_id, &prop);
    verdict vd = get_verdict(new_state);
    handle_verdict(vd, &entry_id);
    submit_rb_event(&new_state, &prop, &vd, &entry_id);

    return 0;
}
```

Listing 13: Generated `handle_empty` eBPF program for case study in chapter 7.

6.4.3 User-space side

The generated user-space code is quite similar to that described in section 5.2. Listing 14 shows only parts missing from listing 5. The most interesting part is perhaps, initialisation of the ring buffer polling, as seen on line 30. `rb` definition from `maps` section from the generated skeleton is passed to the helper function `bpf_map__fd`, the sets up file descriptors for read and write operations. The file descriptor, along with an event handler, `handle_event`, defined on line 19, is passed to ring buffer initialisation helper. Once the program is properly loaded, the ring buffer `rb` is set up to poll for data every 100 *ms*. Upon `cleanup`, the ring buffer is deallocated. For completion, the `submit_rb_event` function on kernel-space code, simply reserves a sample from `rb` ring buffer and submits data as shown in listing 15.

```c
...
#include "common.h"

const char * state_lookup[state_max];
const char * prop_lookup[prop_max];
const char * verdict_lookup[verdict_max];

static void init_lookup() {
    state_lookup[q0] = "q0";
    ...
    prop_lookup[empty] = "empty";
    ...
    verdict_lookup[inconclusive] = "INCONCLUSIVE";
    ...
    return;
}

static int handle_event(void *ctx, void *data, size_t data_sz) { ... }

int main(int argc, char **argv)
{
    struct ring_buffer *rb = NULL;
    struct monitor_bpf *skel;
    int err;
    ...
    init_lookup();
    ...
    /* Set up ring buffer polling */
    rb = ring_buffer__new(bpf_map__fd(skel->maps.rb), handle_event, NULL, NULL);
    if (!rb) {
        err = -1;
        fprintf(stderr, "Failed to create ring buffer\n");
        goto cleanup;
    }
    ...
    while (!exiting) {
        err = ring_buffer__poll(rb, 100 /* timeout, ms */);
        /* Ctrl-C will cause -EINTR */
        if (err == -EINTR) { ... }
        if (err < 0) {
            printf("Error polling perf buffer: %d\n", err);
            break;
        }
    }

cleanup:
    ring_buffer__free(rb);
    monitor_bpf__destroy(skel);
    return err < 0 ? -err : 0;
}

```

Listing 14: Generated user-space code.

```c
...
static inline void submit_rb_event(state* state, proposition* prop,
                                    verdict* verdict, int* entry_id)
{
    struct event* e;

    /* reserve sample from BPF ringbuf */
    e = bpf_ringbuf_reserve(&rb, sizeof(*e), 0);
    if (!e) return;

    /* populate event e with data */
    ...

    bpf_ringbuf_submit(e, 0);
}
...
```

Listing 15: Generated `submit_rb_event` function.

Chapter 7

Case Study I: Simple Stack

For a simple demonstration of *dottobpf* in practice, drawing inspiration from [46], verification of a simple stack data structure is an adequate entry point. The stack provided supports a *push* operation that inserts an element at the top of the stack, the *pop* operation extracts and returns the topmost element of the stack to the user, and the *empty* operation indicates whether the stack is empty or not.

The goal for this case study is to demonstrate eBPF monitor generation from an automaton specification and pertaining configuration file.

7.1 LTL and Automata

Consider the following property:

$$\Box((push \land \Diamond empty) \rightarrow (\neg empty\, U\, pop)) \tag{7.1}$$

This property conforms to the LTL_3 specification, as defined by Bauer et al. [6], and can in normal language be expressed as

for any stack that has been pushed to and is eventually found empty,
a pop event must have occurred before the empty event.

Do note, however, that this property does not assert whether the *should* be empty or not at a give point time but rather, that it *cannot* be empty until a pop has occur ed. Thus, valid for verifying a correct implementation of the stack structure. In order to perform runtime verification on this property with `dottobpf`, the following steps must be completed:

- Generate automaton specification in DOT language (from the provided formula) with LamaConv
- Write a configuration file, providing associations between atomic properties with eBPF event sources
- Generate monitor with `dottobpf`
- Compile and execute program. Reap results.

In order to obtain the automaton in DOT from the property specified in eq. (7.1), invoke LamaConv with following command:

```
$ java -jar rltlconv.jar \
    "LTL=G( (push && F empty) -> (!(empty) U pop) )" \
    --formula \
    --moore \
    --min \
    --dot
```

Listing 16: Invocation of LamaConv.

The resulting automaton is shown in fig. 7.1. Yellow states indicate an *inconclusive* verdict, while the red state signifies a violation of the specified property and yields a *rejected* verdict.

```
digraph G {
  q0 -> q0 [label="?"];
  q2 -> q2 [label="\"pop\""];
  q2 -> q1 [label="\"push\""];
  q1 -> q0 [label="\"empty\""];
  q1 -> q2 [label="\"pop\""];
  q2 -> q2 [label="\"empty\""];
  q1 -> q1 [label="\"push\""];
  start [shape=none, style=invis];
  start -> q2 [label="START"];
  q2 [style=filled, fillcolor=yellow]
  q1 [style=filled, fillcolor=yellow]
  q0 [style=filled, fillcolor=red]
}
```

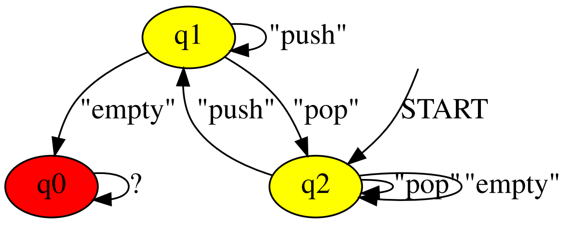


Figure 7.1: Resulting automaton in DOT, with an illustration, from the invocation provided in listing 16.

7.2 Monitor generation

With the automaton readily available, the pertaining configuration file in YAML format is required. As noted in section 5.1 all eBPF programs must have be provided a license under which to operate. To allow for flexibility in this regard, the licence type is left to the discretion of the user and is made mandatory in the configuration file, under the bpf_license key. For the purposes of this thesis, the 'Dual BSD/GPL' license is used exclusively. The following associations' field is an array which informs dottobpf how to associate formula's atomic propositions to generated eBPF programs. Each array element will eventually result in a single eBPF program. Key event_source specifies eBPF program configuration with a required field spec and optional args. event_source.spec defines the attachment point for eBPF program, and is passed directly to the SEC annotation of a program definition. event_source.arg key specifies an array of arguments that will be passed as arguments to the eBPF programs function.

Propositions are associated with the eBPF program by the propositions key under each associations element. The field is also an array, enabling users to define multiple propositions under the same event_source. The required key of each propositions array element is name, which must correspond to the propositions in the provided DOT file. The semi-optional field is predicate, functioning as a filter for the event_source and predicating the assertion of atomic propositions. In that case that several propositions are associated to the same event_source, only one can omit predicate specification, effectively put under an else condition.

The occurrence of *push* and *pop* events are traced with the uprobe facility, and their respective propositions are asserted true whenever the probe fires. Assertion of *empty* proposition, however, requires additional consideration. The empty function, which determines whether the stack is empty or not, returns 1 if it is empty, and 0 otherwise. Consequently, rather than tracing every invocation of the function, its return result is of greater interest. To attach to a return address, a `uretprobe` is employed, and the return value is obtained by passing a single argument of the same type to the eBPF program function. In this case, the argument `int retval` is supplied. Return value of interest is `1`, indicating an empty stack, and is specified as the predicate for this specific association.

```yaml
---
# Required
bpf_license: Dual BSD/GPL

# Optional - Filename of generated ebpf programs.
# filename: myname # defaults to 'monitor'

# Required - Array that associates propositions to eBPF event sources.
# Propositions reference LTL propositions
associations:
  # Optional - File name in which to write the eBPF program.
  # Stores this specific eBPF in that filename. Has precedence over filename key above
  # filename: myfile - defaults to 'monitor'

  # Required - eBPF event source specification
  - event_source:
      # Required - Event specification
      spec: uretprobe//usr/local/bin/stack:empty

      # Optional - Arguments to provide to the generated function
      args:
        - int retval

    # Required - Array of propositions that should be asserted by the eBPF program
    # Propositions should be in the set of automatas atomic propositions
    propositions:
      - name: empty

      # Optional
      # The proposition is asserted only when the predicate is satisfied.
      # Examples:
      # - 1
      # - 0
      # - retval == 1
      # Here the 'retval' variable must be a C type variable readily available
      # in the eBPF program function. Either as a passed argument, or supplied manually
      # after script generation.
        predicate: retval == 1
  - event_source:
      spec: uprobe//usr/local/bin/stack:push
    propositions:
      - name: push
  - event_source:
      spec: uprobe//usr/local/bin/stack:pop
    propositions:
      - name: pop
```

With the automaton specification and the configuration file at hand, `dottobpf` can be invoked as follows. `-o` option specifies the output directory, and `-tf [staticarray|bpfmap]` specifies the type of generated transition function as discussed in section 6.4.1. The build procedure will try to use system-wide libbpf if `LIBBPF_SRC` is not provided. It is recommended to clone the project and build it, in order to ensure reproducibility.

```
$ dottobpf stack.dot stack.yml -o generated -tf staticarray
$ cd generated
$ make LIBBPF_SRC=~/rv-with-bpf-deliverables/libbpf/src
$ sudo ./monitor
```

7.3 Verification of properties

Prior to the verification, consider the push function of the stack, provided in listing 17. Notice that the the buffer index pointer is never incremented upon a function call. Consequently, `empty` will return 1 (i.e. `true`), even though elements (should) have been pushed. The generated monitor should detect this obvious violation of the specified property. With the monitor running, the stack is executed from another shell.

```
void push(int number, int * i)
{
    buffer[*i] = number;
    // *i += 1;
}
```

Listing 17: Faulty `push` function.

```
$ stack < 100commands.in
NO POP
YES
YES
PUSHED 1
NO
NO POP
zsh: killed     stack < 100commands.in
```

Listing 18: Faulty stack execution.

The output of the monitor is listed below. Note that the stack has received a `KILL` signal (listing 18). This is an example of how a reaction to the detected violation may be implemented.

```
...
libbpf: elf: symbol address match for 'empty' in '/usr/local/bin/stack': 0x1235
libbpf: elf: symbol address match for 'push' in '/usr/local/bin/stack': 0x11c9
libbpf: elf: symbol address match for 'pop' in '/usr/local/bin/stack': 0x11f8
Successfully started! Please run `sudo cat /sys/kernel/debug/tracing/trace_pipe`
to see output of the BPF programs.
TIME      COMM      PID       PROPOSITION       STATE     VERDICT         ENTRY_ID
22:40:30  stack     880464    empty             q2        INCONCLUSIVE    0
22:40:30  stack     880464    empty             q2        INCONCLUSIVE    0
22:40:30  stack     880464    push              q1        INCONCLUSIVE    0
22:40:30  stack     880464    empty             q0        REJECTED        0
```

Listing 19: Violation of the push property.

Chapter 8

Case Study II: Distributed System

While the previous example illustrated use of `dottobpf` for a simple single process, case study presented in this section considers verification of properties across a distributed system. More specifically, the objective is to verify protection of application server from unauthorised access. Goals for this case study are to illustrate

1. The process of finding relevant event sources
2. How eBPF can be used to trace third party software
3. The limitations of `dottobpf` for monitor generation
4. The differences between eBPF frontends, specifically BPF CO-RE and bpftrace

To that end, a distributed system composed of a Node.js application server, Authelia - an authentication server, and NGINX as a web and reverse proxy server is deployed. NGINX, is well known open-source software, used for various purposes such as web serving, reverse proxying, caching, load balancing, media streaming etc [56]. Authelia, also an open-source project, and functions as a companion to reverse proxies, providing authentication and authorisation as an extension to proxy services, and a login portal [3].

As shown in the architecture diagram in fig. 8.1, Authelia is only connected to the reverse proxy server and never directly in contact with the application backends. Payloads sent by clients of the protected API will therefore never reach Authelia - save for authentication components, such as the `Authorization` header. In tandem, the reverse proxy servers are configured to generate an authorisation request to Authelia for incoming requests. Response from Authelia instructs the reverse proxy to either allow or block the incoming request due to insufficient privileges.

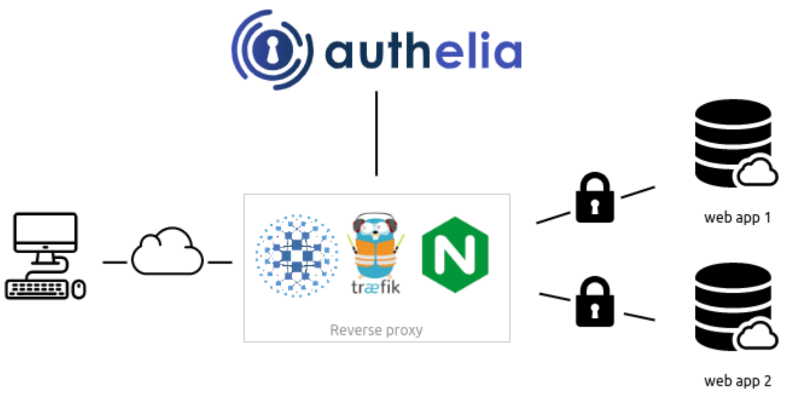


Figure 8.1: Authelia with reverse proxy system. Adapted from [3].

The sequence diagram in fig. 8.2 illustrates the workflow. Upon an 401 unauthorised response, the reverse proxy provides a redirection URL, leading to Authelia login portal. Upon a successful login, a cookie is set for the session, authorisation for the requested content is verified, and the original request forwarded to the application backend. Installation and configuration guides are provided in appendix B. Application configuration for the case at hand is as follows:

• `hello.com/world`	–	Open Access
• `hello.com/user`	–	Only authorised users
• `hello.com/admin`	–	Only authorised admins
• `auth.hello.com/verifyauth`	–	Authelia verification request API
• `auth.hello.com`	–	Authelia login portal

Installation steps and configuration for required software is provided in appendix B.

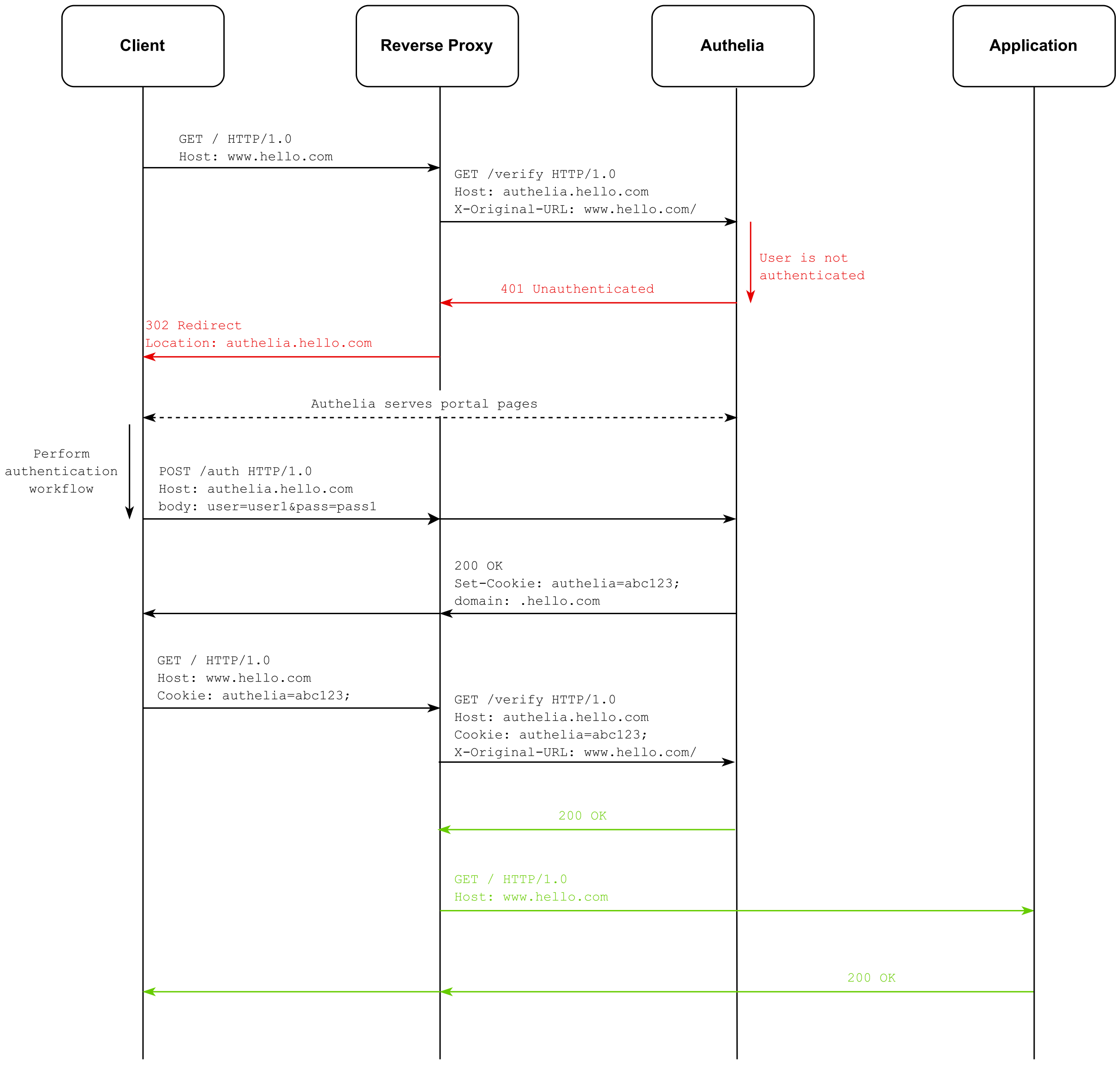


Figure 8.2: Authentication sequence with Authelia and NGINX.

8.1 Property specification

Before turning to formulation in LTL, a brief consideration of system behaviour provides aid in determining which events to trace and how to do so.

The sensitive component of the SUS is the application backend and the objective is to protect it from unauthorised access. Authelia documentation, along with the sequence diagram, informs that every incoming request generates an authorisation request. Securely forwarding client requests to the application server then implies that a successful authorisation must have occurred. For the authorisation event to occur, an authorisation request issued by NGINX to Authelia must precede it.

- Client request to sensitive content is received on NGINX
- NGINX issues an authorisation request to Authelia
- Authelia `200 OK` authorisation verdict
- Client request is forwarded to the application backend upon successful authorisation, or redirected to the Authelia login portal

Numerous alternatives to monitor such behaviour present themselves - solely on NGINX, on all deployed servers, or even on network level. Moreover, monitoring may be performed on individual client requests, or by differentiating client connections. For the purpose of illustrating distributed tracing with eBPF, as well as limits of dottobpf, a combination of monitoring across servers and network level functions are selected. Items 2 and 4 are substituted with TCP events, rather than application level events.

- Client request is received on Nignx
- TCP connect event from NGINX to Authelia
- Authelia `200 OK` authorisation verdict
- TCP accept event on application server upon successful authorisation.

Note the absence of 'redirection to Authelia portal' in item 4. A redirection implies a new client request, which is covered by item 1. Having reasoned about behaviour, LTL formulation is at hand. Following abbreviations are used for the sake of simplicity:

•	*req*	–	new client request on NGINX
•	*tcpconnectauthelia*	–	TCP conenct *system call* to Authelia (from NGINX)
•	*authed*	–	Authorisation response `200 OK`
•	*tcpaccepthello*	–	TCP accept event on the Node.js application server

For better comprehension, desired properties are first expressed in natural language.

1. For every request that eventually is TCP accepted on the application server, a successful authentication event must occur
2. Every successful authentication event must be preceded only by a TCP connect call to Autehlia
3. Every successful authentication event must be succeeded only by a TCP accept on application server
4. Every TCP accept event on application server must be succeeded only by a new client request. footnote

These properties are characterised as precedence chains [20], and can be derived as follows:

1. $(req \wedge \Diamond tcpaccepthello) \rightarrow (\neg tcpaccepthello\, W\; authed)$

The second property specifies the precedence of *tcpconnectauthelia* to an *authed* event.

2. $(req \wedge \Diamond tcpaccepthello) \rightarrow (\neg tcpaccepthello\, W\; (tcpconnectauthelia \wedge \bigcirc\, authed)$
3. $authed \rightarrow \bigcirc\, tcpaccepthello$

Then, precedence of *tcpaccepthello* to a new request must be provided

4. $authed \rightarrow \bigcirc\, (tcpaccepthello \wedge \bigcirc req)$

Global conjunction of items 1 and 4 then specifies desired behaviour *once req* proposition is true. However, before *req* is asserted, *all* other propositions also satisfy the formula. Therefore, further precedence needs to be applied to the *req* proposition. The final formulation is provided in 8.1 and the generated automaton in 8.3.

$$\begin{aligned}
\Box\Big(&\\
&\big(\,(req \wedge \Diamond tcpaccepthello)\\
&\qquad \rightarrow (\neg tcpaccepthello\, W\; (tcpconnectauthelia \wedge \bigcirc\, auth)\,\big)\\
&\wedge (authed \rightarrow \bigcirc\, (tcpaccepthello \wedge \bigcirc req)\\
\Big) &\wedge \neg(authed \vee tcpconnectauthelia \vee tcpaccepthello)\; W\; req
\end{aligned} \tag{8.1}$$

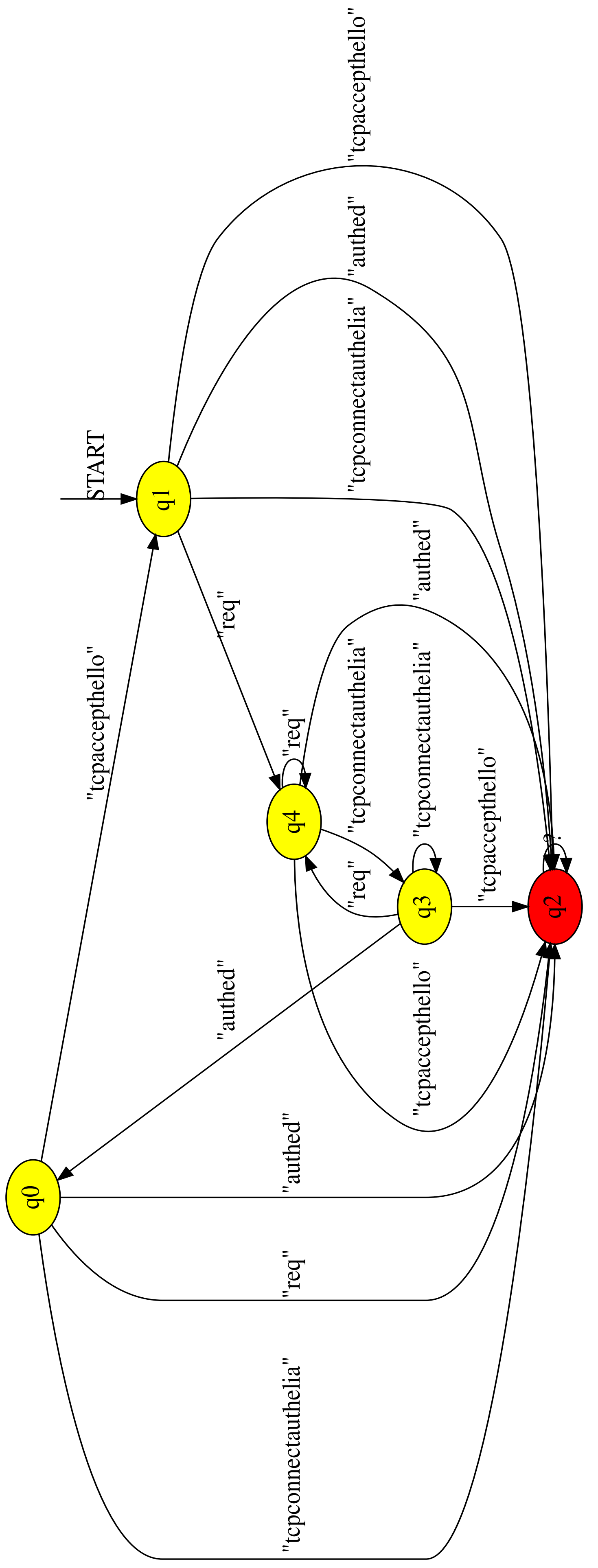


Figure 8.3: Automaton generated by LamaConv with formula provided in eq. (8.1).

8.2 Locating event sources

The next step to monitor generation is to locate event sources with which to associate atomic propositions. As discussed previously, eBPF programs can attach to statically defined tracepoints, raw tracepoints, kernel and user-space events. Dynamically tracing user-space functions requires their symbols to be represented in binary's ELF. `readelf` command is built for this purpose, however, `objdump` is used here for truncated output. Events regarding request handling are of interest with respect to the *req* proposition.

```
$ objdump -tT /usr/sbin/nginx | grep request
...
0000000000066370 g    DF .text  0000000000000197  Base        ngx_http_process_request_uri
00000000000642c0 g    DF .text  000000000000009f  Base        ngx_http_create_request
0000000000062f00 g    DF .text  000000000000009b  Base        ngx_http_filter_finalize_request
...
```

Listing 20: `objdump` sample on NGINX binary.

The output informs that there is indeed a function `ngx_http_create_request` to which an eBPF can attach. *tcpconnectauthelia* proposition can be traced either by TCP connect handling by NGINX, or TCP connect system call. For the sake of diversity, the latter is chosen. Dynamically tracing kernel events requires a similar approach to that of user-space events - function symbols must be located. Kernel function symbols are represented in the `/proc/kallsyms` file. Listing 21 illustrates several alternatives. Considering that SUS only relies on IPv4, `tcp_v4_connect` is sufficient for this case.

```
$ sudo cat /proc/kallsyms | grep tcp | grep connect
...
ffffffffbd2fc050 t tcp_connect_queue_skb
ffffffffbd2fe210 t tcp_connect_init
ffffffffbd300670 T tcp_connect
ffffffffbd306ff0 t tcp_v4_pre_connect
ffffffffbd30a850 T tcp_v4_connect
ffffffffbd312330 T tcp_fastopen_defer_connect
ffffffffbd3c8360 t tcp_v6_pre_connect
ffffffffbd3ca530 t tcp_v6_connect
ffffffffbd422ff0 t mptcp_disconnect
...
```

Listing 21: `/proc/kallsyms` system file.

The third proposition, namely *authed*, represents a successful authorisation verdict from Authelia. Unfortunately, the invocation of `objdump -tT /usr/sbin/authelia`, returns only third party openSSL symbols along with some arbitrary Go functions, which implies the installed binary as been stripped of debug symbols. The issue may be resolved building Authelia binary from source and keeping debug symbols. Nonetheless, this approach yields unsatisfying results[1]. Alternatively, Authelia authorisation response can be traced on NGINX. Inspecting the source code reveals that the response to an upstream request is initially handled by `ngx_http_upstream_finalize_request` function. However, being a static function its symbol is compiled away, rendering it only visible to the functions within the same file, or more precisely, its translation unit. `ngx_http_finalize_request` function on the other hand, wraps `ngx_http_upstream request`, and is not static. Invocation:

[1]Libbpf makes assumptions about ELF symbols being valid C identifiers, which Go violates. A workaround is proposed in [53]. Bpftrace scripts, however, do attach.

```
objdump -tT /usr/sbin/nginx | grep ngx_http_finalize_request
```

indeed verifies that the function symbol is represented in the ELF. Pursuing the possibility further, the first argument is of `ngx_http_request_t` type, containing pertinent information about the request. URI and response code can be obtained by object parsing. While certainly possible in theory, it proves rather difficult in practice. Inclusion of system-wide types in kernel, necessary for object parsing, introduces a plethora of type redefinition errors in BPF CO-RE. This too, is discussed further in section 10.4. The remaining alternative is parsing the request object in NGINX source code itself, and making information easily available by either new functions or static tracepoints. To that end, a patch implementing static tracepoints with SystemTap, publishing status code of authorisation response, is supplied. Steps for patching, building from source and installing NGINX are documented in Appendix A. Upon successful installation the tracepoint can be listed by:

```
$ readelf -n /usr/sbin/nginx
...
stapsdt              0x0000004b   NT_STAPSDT (SystemTap probe descriptors)
    Provider: ngx_http_upstream
    Name: auth_res
    Location: 0x000000000007522c, Base: 0x00000000000dab94, Semaphore: 0x0000000000000000
    Arguments: 8@200(%rax) 8@648(%rbp)
...
```

Listing 22: SystemTap probes in NGINX ELF note segments.

The proposition *tcpaccepthello*, can either be traced as TCP accept system call or the TCP accept handling in Node.js. Aiming for diversity, the latter option is selected. The binary's synbols are once again inspected using `objdump`, and yields several possibilities. After filtering out irrelevant results,

```
$ objdump -tT /usr/bin/node | grep accept

000000000164ced0 l     F .text  0000000000000062                uv__accept
0000000000000000       F *UND*  0000000000000000                accept4@@GLIBC_2.10
0000000001658e80 g     F .text  0000000000000194                uv_accept
0000000000000000       F *UND*  0000000000000000                accept@@GLIBC_2.2.5
0000000001961af0 g     F .text  0000000000000083                OCSP_accept_responses_new
000000000184bcd0 g     F .text  00000000000001c2                BIO_accept
000000000184d710 g     F .text  0000000000000052                BIO_new_accept
00000000017d33a0 g     F .text  00000000000000bb                SSL_accept
0000000000000000      DF *UND*  0000000000000000 (GLIBC_2.10) accept4
0000000000000000      DF *UND*  0000000000000000 (GLIBC_2.2.5) accept
000000000184bcd0 g    DF .text  00000000000001c2  Base          BIO_acceptaverified
000000000184d710 g    DF .text  0000000000000052  Base          BIO_new_accept
00000000017d33a0 g    DF .text  00000000000000bb  Base          SSL_accept
00000000017d39c0 g    DF .text  0000000000000098  Base          SSL_set_accept_state
000000000184d700 g    DF .text  0000000000000006  Base          BIO_s_accept
0000000001658e80 g    DF .text  0000000000000194  Base          uv_accept
```

Listing 23: `objdump` sample on Nodejs binary.

only two options remain in the listing 23. Functions starting with `BIO_*` and `SSL_*` can be ignored, as they belong to OpenSSL library. The remaining functions are `accept4@@GLIBC_2.*` and `uv_accept`. While GLIBC functions are essentially system calls, `uv_accept` is a function from the `libuv` library, which provides support for asynchronous I/O [54]. Therefore, it is most likely that `uv_accept` relies on the GLIBC accept function to handle TCP accept events. Thus, it appears to be the most promising candidate for tracing TCP accept events in Node.js. This assumption can be easily verified using `bpftrace`.

```
$ # With node server running ...
$ sudo bpftrace -e 'uprobe:/usr/sbin/node:uv_accept { printf(%s\n, probe); }'
Attaching 1 probe...
# curl localhost:3000/world from another shell or browser
uprobe:/usr/sbin/node:uv_accept
```

Listing 24: `bpftrace` simple tracing `uv_accept`.

```
---
bpf_license: Dual BSD/GPL
filename: distributed
associations:
  - event_source:
      spec: uprobe//usr/sbin/nginx:ngx_http_create_request
    propositions:
      - name: req
  - event_source:
      spec: kretprobe/tcp_v4_connect
    propositions:
      - name: tcpconnectauthelia
        predicate: dport == 9091
  - event_source:
      spec: uprobe//usr/bin/node:uv_accept
    propositions:
      - name: tcpaccepthello
  - event_source:
      spec: usdt//usr/sbin/nginx:ngx_http_upstream:auth_res
      args:
        - int cnum
        - int http_status
    propositions:
      - name: authed
        predicate: http_status == 200
```

Listing 25: Configuration file for formula specified in eq. (8.1).

8.3 Monitor generation

The remaining piece prior to monitor generation with dottobpf, are the configuration file associations. Consider the association for the *authed* proposition; the eBPF program for the proposition receives two arguments: `int cnum` and `int http_status`. Then a predicate `http_status == 200` is applied, informing the eBPF program not to assert association *authed* once this event fires, unless the predicate is satisfied. Similarly, the *tcpconnectauthelia* event receives a predicate `dport == 9091`. This is due to the `tcp_v4_connect` probe which will fire for all TCP connect system calls. However, only TCP connect call to Authelia running on port 9091, are of interest. The predicate ensures that the eBPF program returns as quickly as possible if destination port is not 9091, and thus not registering the proposition as true.

A keen reader may note that the `kretprobe` on `tcp_v4_connect` is selected over its entry counterpart. Furthermore, they may recall the `tcpconnect` example from section 5.1, listing 2, which explains that a TCP connect systemcall call fills in socket information during the call but returns `void`. To effectively examine the populated socket, its address must first be recorded, saved in a map, and then retrieved and inspected upon function return. The `dport` variable provided to the predicate is neither declared nor initialised. Additionally, as only TCP connect calls made by NGINX are pertinent, programs on both the entry and return of the call must return as quickly as possible to avoid unnecessary performance latency. This example illustrates the limits of `dottobpf`, where manual intervention is necessary for a fully functional monitor. A patch that addresses these issues is supplied. Execution of the monitor is shown below.

```
$ dottobpf cs_distributed.dot cs_distributed.yml -o generated -tf staticarray
$ patch generated/distributed.bpf.h  <distributed.patch
$ cd generated && make LIBBPF_SRC ~/rv-with-bpf-deliverables/libbpf/src
$ sudo ./distributed
...
Successfully started! Please run `sudo cat /sys/kernel/debug/tracing/trace_pipe`
to see output of the BPF programs.
TIME       COMM       PID        PROPOSITION          STATE      VERDICT         ENTRY_ID
18:53:11   nginx      571132     req                  q4         INCONCLUSIVE    0
18:53:11   nginx      571132     tcpconnectauthelia   q3         INCONCLUSIVE    0
18:53:11   nginx      571132     authed               q0         INCONCLUSIVE    0
18:53:11   node       570316     tcpaccepthello       q1         INCONCLUSIVE    0
18:53:21   node       570316     tcpaccepthello       q2         REJECTED        0
```

Listing 26: Patching of the generated code for distributed system study.

Simulating a falsifying event, as seen on last line in listing above, can be done by making a request directly to the application server, bypassing NGINX[2].

8.4 Limitations

While this study illustrates tracing across multiple and distributed systems, it is limited by the ability to only handle sequential client requests. Concurrent requests from multiple clients may yield false positives. Another limitation of the case, is that the monitor is oblivious to which requests are in fact authorised and which were not. Consider the sequence:

$$\texttt{req} \rightarrow \texttt{req} \rightarrow \texttt{req} \rightarrow \texttt{tcpconnectauthelia} \rightarrow \texttt{authed} \rightarrow \texttt{tcpaccepthello}$$

Which of these requests is eventually authorised? Are these requests directed to the protected application server at all? The monitor does not differentiate incoming requests. One solution is the provide counters for *req*, *authed* and *tcpaccepthello* propositions in order to get an inkling of what the system is doing. Suppose, on the other hand, that incoming requests may be differentiated by the monitor, how then to differentiate which requests are forwarded to the application server? The next case study discusses parameterisation of properties in order to answer these questions.

[2]Try, for example, `curl localhost:3000/world`.

Chapter 9

Case Study III: Concurrent Connections

Examples provided so far have only discussed sequential event handling for a single, global automaton, that is, colloquially, a single state value despite possibly concurrent or, even worse, unrelated events. This case study aims to illustrate how parameterisation of automata with `dottobpf`, can solve such problems. The study uses the same authentication system from the previous section. A reconsideration of desired behaviour and how to monitor for it, is necessary.

Parameterisation inherently relies on unique identifiers of phenomena one wishes to study, across the entire domain. Evaluating the previous case study from this perspective, quickly reveals the chosen domain — spanning two servers and system calls — makes it challenging to find unique identifiers applicable to the entire system. Suppose each client request could be identified on NGINX, the question as to how to identify and relate these requests to pertaining TCP system calls and Node.js responses. On the transport layer, this is unfeasible, simply due to requests being application level object, of which the kernel, and TCP, are entirely unaware of. In the case of correctly associating requests on NGINX and Node.js, one could indeed parse request headers for some unique identifier, such as cookies. Maintenance of some lookup table would be required, in that case, in order to associate an initial incoming client request with the cookie value it receives from Authelia server. While certainly plausible, this solution poses the fundamental challenge encountered throughout this thesis: parsing of complex user-space object. For the sake of simplicity, the scope in thus study is limited only to NGINX server. Moreover, each request is obviously preceded by an established TCP connection. To make things more exciting, at least from an operational perspective, requests can be monitored in relation to the client connection pertaining to the request, thus allowing parameterised grouping of requests by their connections.

9.1 Parameterising Automata

For each new client connection, NGINX initialises a new `ngx_connection_t` object, to which a unique connection identifier is atomically assigned. The intent of this variable is to allow identification of connection for debugging or analysis purposes, fitting splendidly for the task at hand. Each request object on NGINX references a client connection object, allowing following formulation:

- Each new connection must registered.
- Each request shall be associated to a registered connection.
- Each request to access the application server must first be forwarded to Authelia verification API.
- Each request forwarded to the application server must be granted access by Authelia verification API.
- Forwarding requests to the Authelia server is allowed.

- Each connection is (eventually) closed.

Before preceding with derivation of atomic properties and reqiuired parameters for the required formula, review the fact that every NGINX request objects references a connection. Then, for the sake of simplicity and scope limitation, observation of the request object instance can be substituted by the observation the connection number of the connection the request object references. Practically, this means that, instead of (ideally) recording the specific request object instance, connection identifiers pertaining to that request are recorded, thereby alleviating the necessity of parameterising request object instances. Now, in order to monitor for behaviour expressed above, following atomic properties are derived:

•	*httpconn*	—	new connection established
•	*httpclose*	—	(existing) connection closed
•	*authed*	—	authorisation response `200 OK`
•	*req_dest*	—	request destination

With the intent to reason about distinct connection objects, automata are identified by unique connection identifiers. Moreover, reasoning about request destination and authorisation verdicts requires evaluations of these domain variables as well. In the realm of NGINX (with configuration presented in chapter 8), the value domain of these parameters can be defined as:

1. Connection identifier

$$c \in C\,, \text{ where } C = \{\, i_1, \dots, i_n \mid i, n \in \mathbb{N} \wedge 1 \le n \le 2^{64} - 1 \,\} \tag{9.1}$$

2. Request URL

$$r \in R\,, \text{ where } R = \{\, \mathit{hello.com}/* \,,\, \mathit{auth.hello.com}/* \,\} \tag{9.2}$$

Then, in tandem with desired behaviour expressed above and parameters in the NGINX value domain, following parameters can be assigned to propositions above.

$$AP = \begin{cases} \mathit{httpconn}(c), & \mathit{req\_dest}(c, r) \\ \mathit{auth}(c), & \mathit{httpclose}(c) \end{cases} \tag{9.3}$$

Instantiation of automata for each of $2^{64} - 1$ possible domain objects, is as impractical as inefficient. Given the fundamental principle of runtime verification is reasoning about observed traces, and in order to limit memory footprint, each automaton instantiation is a consequence of an observed domain value (e.g. c in eq. (9.3)). Thereby, allowing to limit the domain to the subset of values actually necessary to verify a run. During the matching process against a parameterised proposition, which may contain both quantified and bound variables, every unbound variable within the proposition may receive a value drawn from the underlying domain[52, p. 36]. Consider the function `update_state`, listing 27, in this light. The function attempts a `state_map` lookup. If lookup is unsuccessful for the provided `entry_id`, a new state element is instantiated. As `update_state` is called by each generated eBPF program, each proposition effectively functions as an *automaton constructor*. Moreover, supplied function arguments may also be applied as parameters to the propositions. Atomic propositions in monitors generated by `dottobpf` can be expressed as defined in [52, p. 36]:

Definition 9.1.1 (Parameterised propositions)
For the set of proposition names PN where $p \in PN^{(n)}$ denotes a proposition with arity $n \in \mathbb{N}$. Then the set of propositions P, given a value domain D and a set of variables V, are defined as:

$$P = \bigcup_{n \in \mathbb{N}} \bigcup_{p \in PN^{(n)}} p(v_1, \dots, v_n) \mid v_k \in D \cup V, 1 \ \ge\ k \ \ge n$$

```
static inline state update_state(int* entry_id, proposition* prop)
{
    state* curr_state = bpf_map_lookup_elem(&state_map, entry_id);
    // Initialize state val
    if (!curr_state) {
        int res = bpf_map_update_elem(&state_map, entry_id,
                                      &aut.initial_state, BPF_NOEXIST);
        if (res < 0) return -1;
        curr_state = bpf_map_lookup_elem(&state_map, entry_id);
        if (!curr_state) return -1;
    }
    ...
}

static inline void handle_verdict(verdict* vd, int* entry_id)
{
    if (*vd == reject) {
        bpf_printk("Sending KILL signal\n");
        bpf_send_signal(9);
        // bpf_map_delete_elem(&state_map, entry_id);
    } else if (*vd == accept) {
        bpf_printk("Deleting accepted element\n");
        bpf_map_delete_elem(&state_map, entry_id);
    }
}
```

Listing 27: `update_state` and `handle_verdict` helper functions.

Take also note of the implementation of `handle_verdict` function in listing 27, which is also called by every generated eBPF program. In the case of an event leading to an *accept* state, which is a sink state, the state value is deleted from the `state_map` — a decision made to reduce monitors memory footprint. Then, each proposition *may* perform the function of a *destructor*, expressed analogously to definition 9.1.1

Now, returning to the current study, and in light of domain limitation, the bounds of connection identifier parameter c in eq. (9.3) can be redefined from eq. (9.1) to domain D_C, such that:

$$c \in D_C\,,\ \text{where } D_C \subset C,\ \ C = \{\ i_1, \dots, i_n | i, n \in \mathbb{N} \mid 1 \leq n \leq 2^{64} - 1\ \} \tag{9.4}$$

In satisfying the behaviour expressed at the beginning of this section, three patterns URL should be distinguished by the '*req_dest* proposition: `hello.com/*` — requests to the application server, `auth.hello.com/verifyauth` — requests outgoing to the Authelia verification API, and `auth.hello.com/(?!verifyauth\b)\b\w+` — requests to all other URI to Authelia server. Achieving this with code generation, proves to be tricky to. Request URLs are essentially *char* pointers in C and handling of user-space pointers in eBPF requires silky gloves, especially with complex types[1]. In addition to the commentary in section 9.2.1 the solution is approached by hardcoding parameter filtering in the NGINX source code, as shown in listing 28. Once a pattern is matched, request instance's connection id is published on a new event source. Authentication result is made available in a similar fashion.

[1]See section 10.4 for further discussion.

```
static void
ngx_http_upstream_connect(ngx_http_request_t *r, ngx_http_upstream_t *u)
{
    ...
    if (strstr((char *) r->headers_in.server.data, "auth") != NULL) {
        DTRACE_PROBE1(ngx_http_upstream, authelia, r->connection->number);
    } else if (strncmp((char *) r->uri.data, "/verifyauth", 10) == 0) {
        DTRACE_PROBE1(ngx_http_upstream, verifyauth, r->connection->number);
    } else {
        DTRACE_PROBE1(ngx_http_upstream, hello, r->connection->number);
    }
    ...

}

static void
ngx_http_upstream_finalize_request(ngx_http_request_t *r,
    ngx_http_upstream_t *u, ngx_int_t rc)
{
    if (!strncmp((char *) r->uri.data, "/verifyauth", 10)) {
        DTRACE_PROBE2(ngx_http_upstream, auth_res, r->connection->number,
                      r->headers_out.status);
    }
    ...
}
```

Listing 28: Applied parameter filtering in NGINX source code.

Propositions are reformulated as shown in eq. (9.5), where *upstreamhello*, *verifyauth*, and *upstreamauthelia* signify a request forwarding to `hello.com/*`, `auth.hello.com/verifyauth`, `auth.hello.com/(?!verifyauth\b)\b\w+`, respectively, for some connection id c. Finally, the LTL property for the desired behaviour can be formulated as in eq. (9.6).

$$AP = \begin{cases} \mathit{httpconn}(c), & \mathit{verifyauth}(c), \quad \mathit{upstreamauthelia}(c), \\ \mathit{upstreamhello}(c), & \mathit{auth}(c), \quad \mathit{httpclose}(c) \end{cases} \tag{9.5}$$

Considering the figure of the automaton by the end of this section, it becomes evident that this formula is not ideal. Inward edges after first assertion of *authed* (state $q5$), for propositions *verifyauth*, *upstreamhello* and *authed* imply that unauthorised access by privilege escalation would go undetected. Nonetheless, it captures the required intent well enough to illustrate parameterisation of propositions. Due to the fact that LamaConv does not, at the time writing support, parameterisation, the parameter is removed from the invocation, which is performed as performed as seen previously.

$$
\begin{aligned}
\forall c \mid c \in D_C :\ & \mathit{httpconn}(c)\ \land \bigcirc \Big(\\
& \quad \mathit{verifyauth}(c)\ \land \bigcirc\ (\\
& \qquad \mathit{verifyauth}(c)\ U\ (\\
& \qquad\quad \mathit{authed}(c)\ \land \bigcirc\ (\\
& \qquad\qquad \mathit{upstreamhello}(c)\ U\ \mathit{httpclose}(c) \\
& \qquad\qquad\quad R\ \neg(\mathit{httpconn}(c) \lor \mathit{upstreamauthelia}(c)) \\
& \qquad\quad) \\
& \qquad)\ \lor\ \mathit{httpclose}(c)) \\
& \quad) \lor\ \mathit{upstreamauthelia}(c)\ U\ \mathit{httpclose}(c) \\
& \Big)
\end{aligned}
\tag{9.6}
$$

9.2 Parameterising with `dottobpf`

Combating the limits of what can (and should) be specified through the DOT file, parameterisation of automata in `dottobpf` is achieved through the configuration file. Provided parameters need to made available to eBPF sources, i.e. parameters must be variables available on the eBPF program stack. Users are able to provide eBPF function arguments, as well as a predicate in relation to the provided variables, in order to assert parameterised atomic propositions. A truncated version of the configuration file for this study illustrates this in listing 29. The provided configuration defines, under

```
1  state_entry_id_arg: int cnum
2  associations:
3    - event_source:
4        spec: usdt//usr/sbin/nginx:ngx_http_request:http_init_connection
5        args:
6          - int cnum
7      propositions:
8        - name: httpconn
9    - event_source:
10       spec: usdt//usr/sbin/nginx:ngx_http_upstream:auth_res
11       args:
12         - int cnum
13         - int status
14     propositions:
15       - name: auth
16         predicate: status == 200
```

Listing 29: Configuration file with automata parameterisation

the key `state_entry_id_arg`, line 4, that the unique identifier of each automaton instance should be the argument `int cnum`. This argument is then made available to event sources, by simply defining it as one of arguments to the event source program function, under the array field `event_source.args`. At this point, the curious reader might wonder why the *authed* proposition is not parameterised, when it clearly can be, especially as variable holding the verdict is made available to the eBPF program. While this is certainly true, the decision not to do so is motivated by the initial expressions of desired behaviour, specifically the statement — Each request forwarded to the application server must be granted access by Authelia verification API. Monitoring unauthorised access is done simply by verifying whether the *upstreamhello* assertion is preceded by *authed* assertion. Predicating *authed* on a 200 response code value is enough, relieving the complexity of further parameterisation. However, the functionality to do so is readily available. Upon generating monitors and compiling generated code, execution of monitors yields results as illustrated below. Note the different values in the column ENTRY_ID, on the right

```
1  Successfully started! Please run `sudo cat /sys/kernel/debug/tracing/trace_pipe`
2  to see output of the BPF programs.
3  TIME      COMM    PID     PROPOSITION        STATE   VERDICT        ENTRY_ID
4  00:15:04  nginx   754876  httpconn           q7      INCONCLUSIVE   1
5  00:15:04  nginx   754876  verifyauth         q0      INCONCLUSIVE   1
6  00:15:04  nginx   754876  authed             q5      INCONCLUSIVE   1
7  00:15:04  nginx   754876  upstreamhello      q6      INCONCLUSIVE   1
8  00:15:04  nginx   754876  httpclose          q3      ACCEPTED       1
9  00:15:21  nginx   754876  httpconn           q7      INCONCLUSIVE   4
10 00:15:21  nginx   754876  httpclose          q3      ACCEPTED       4
11 00:15:27  nginx   754876  httpconn           q7      INCONCLUSIVE   5
12 00:15:27  nginx   754876  verifyauth         q0      INCONCLUSIVE   5
13 00:15:27  nginx   754876  authed             q5      INCONCLUSIVE   5
14 00:15:27  nginx   754876  upstreamhello      q6      INCONCLUSIVE   5
15 00:15:27  nginx   754876  verifyauth         q5      INCONCLUSIVE   5
16 00:15:27  nginx   754876  authed             q5      INCONCLUSIVE   5
17 00:15:27  nginx   754876  upstreamhello      q6      INCONCLUSIVE   5
18 00:15:35  nginx   754876  verifyauth         q5      INCONCLUSIVE   5
19 00:15:35  nginx   754876  httpconn           q7      INCONCLUSIVE   11
20 00:15:35  nginx   754876  httpclose          q3      ACCEPTED       11
21 00:15:47  nginx   754876  httpconn           q7      INCONCLUSIVE   12
22 00:15:47  nginx   754876  upstreamauthelia   q2      INCONCLUSIVE   12
23 00:15:47  nginx   754876  upstreamauthelia   q2      INCONCLUSIVE   12
24 00:15:47  nginx   754876  upstreamauthelia   q2      INCONCLUSIVE   12
25 00:15:47  nginx   754876  upstreamauthelia   q2      INCONCLUSIVE   12
26 00:15:47  nginx   754876  upstreamauthelia   q2      INCONCLUSIVE   12
27 00:15:47  nginx   754876  upstreamauthelia   q2      INCONCLUSIVE   12
28 00:15:47  nginx   754876  upstreamauthelia   q2      INCONCLUSIVE   12
29 00:15:54  nginx   754876  upstreamauthelia   q2      INCONCLUSIVE   12
30 00:15:55  nginx   754876  verifyauth         q5      INCONCLUSIVE   5
31 00:15:55  nginx   754876  authed             q5      INCONCLUSIVE   5
32 00:15:55  nginx   754876  upstreamhello      q6      INCONCLUSIVE   5
33 00:16:11  nginx   754876  httpclose          q3      ACCEPTED       5
34 00:16:11  nginx   754876  httpclose          q3      ACCEPTED       12
```

Listing 30: Output of parameterised monitoring.

hand side. These indicate different observed connection identifiers to which each automaton instance is associated. The listing examplifies three attempts at access content at `https://hello.com/world`. The first executed with `curl -k https://hello.com/world`, identified by entry id 1. The next two are requested via the browser, first to `https://hello.com/world`, identified by entry id 4. The connection is immediately closed by the browser due to the self signed certificate. Once the risks have been accepted, a new connection is initiated, identified by entry id 5. Two successive requests are made by the browser — one for page content and the other for `favicon.ico`. The attempt to access restricted content at `https://hello.com/user` starts on line 19. Entry id 11 denotes a *new* internal connection from NGINX to Authelia server. Authelia respons with `401 Unauthorized`, and the connection is closed. In the provided configuration, NGINX is instructed to redirect the user to the login portal for `401` response code.

```
error_page 401 =302 https://auth.hello.com/?rd=$target_url;
```

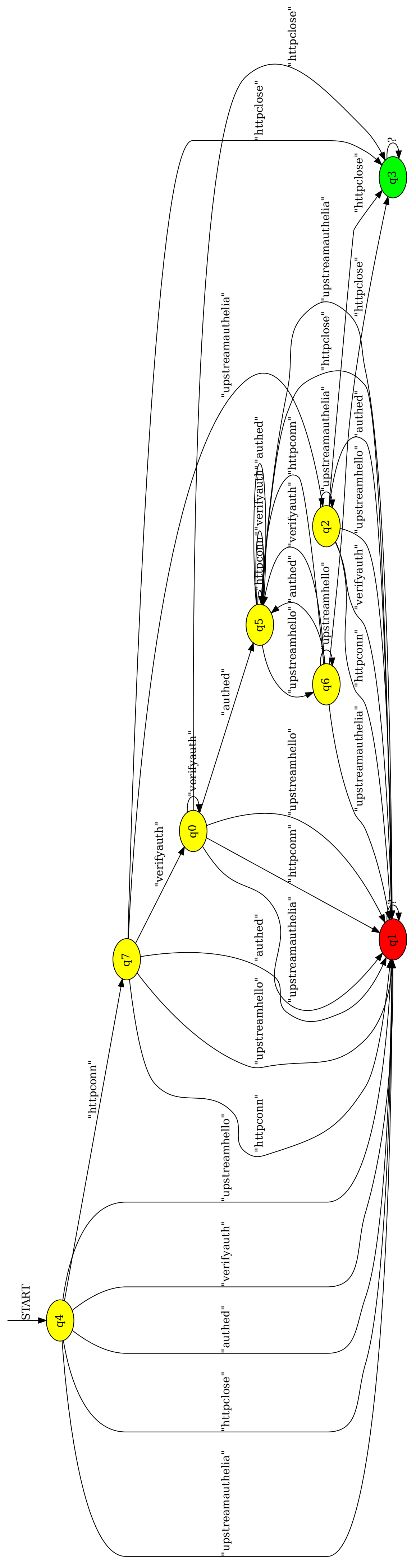
Therefore a new connection from NGINX to Authelia is created, entry id 12, which is requesting login form, from `auth.hello.com` an returning this to the user. Upon successful login, the client identified by entry id 5, is granted access on line 31.

9.2.1 Comments on multiple parameters

So far, parameterised propositions with single parameters have been demonstrated. Parameterisations with arity > 1 are possible with `dottobpf` to some degree. Instead of specifying primitive types, composite types could be provided:

```
state_entry_id_arg: struct very_complex* stuff
```

The state value is stored in a hash map, and so an entries with unique combinations would be identified. However, such configuration would require manual revisions to the generated script, both for type definitions and assignment of struct members. More importantly, considering the fact that eBPF verifier rejects programs that have uninitialised variables, all propositions are required to have the same parameters. This is due to the fact that all states are stored in a single state map. A possible workaround could include dummy values, but the efficacy of the approach is debatable. Another map type that could facilitate more flexible multiple parameterisation is of nested types, aptly named `BPF_MAP_TYPE_HASH_OF_MAPS`, which is presented as suggestions for future work in section 10.7.


START
q4
q7
q0
q5
q6
q2
q1
q3
"httpconn"
"verifyauth"
"authed"
"httpclose"
"upstreamauthelia"
"upstreamhello"
"verifyauth"
"authed"
"upstreamhello"
"httpconn"
"upstreamauthelia"
"httpclose"
"upstreamauthelia"
"upstreamhello"
"httpconn"
"upstreamauthelia"
"authed"
"upstreamhello"
"verifyauth"
"httpclose"
"httpconn"
"verifyauth"
"authed"
"upstreamhello"
"authed"
"verifyauth"
"httpconn"
"upstreamhello"
"upstreamauthelia"
"httpclose"
"upstreamauthelia"
"httpclose"
"authed"
"upstreamhello"
"verifyauth"
"httpconn"
"upstreamauthelia"
"httpclose"
"httpclose"
?
?

Chapter 10

Evaluation

Having demonstrated various capabilities of `dottobpf` and how it can, in conjunction with LamaConv, be applied for runtime verification, an evaluation of its limits and design choices, as well as challenges encountered throughout, is due.

10.1 Taxonomy

Leveraging the taxonomy of runtime verification tools given by [23], `dottobpf` is evaluated in light of 7 major concepts, as illustrated in fig. 10.1.

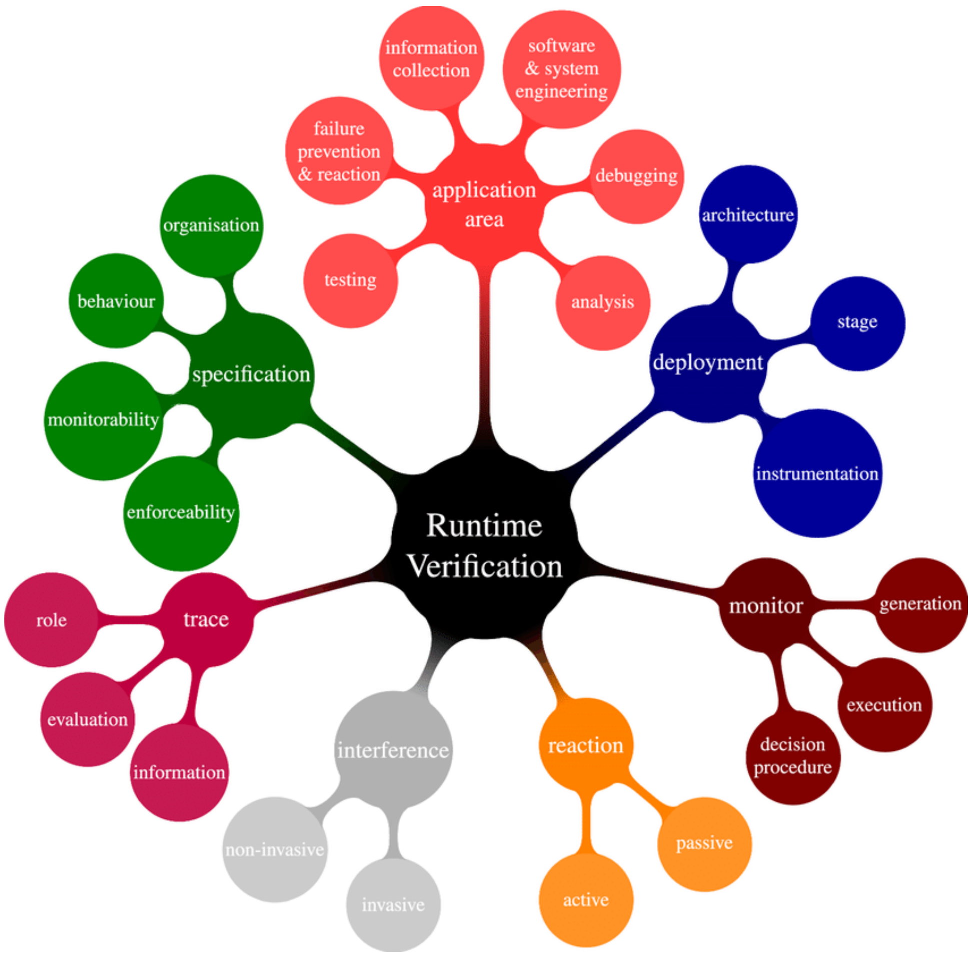


Figure 10.1: Taxonomy of runtime verification tools. Adapted from [23].

Specification indicates the intended system behaviour. `dottobpf` monitor generation relies on *explicit* specification, by expecting users to provide the *system model* specification in a DOT file, alongside an eBPF configuration. The specification is classified *declarative* as the system model is obtained from a temporal logic specification where *time* is handled as a logical ordering of discrete events. Notably, there are no *implicit* specifications currently present in `dottobpf`.

Monitor's *decision procedure* is generated *explicitly* from the provided system model, and is classified as *operational*, by being based on automata formalisms. Soundness and completeness properties of decision procedure cannot be absolutely guaranteed, due to lack of thorough testing for unforeseen circumstances or errors that may arise in the underlying domain.

Deployment. Instrumentation is applied at the *software* level. Considering that eBPF is designed to consume traces of running systems, the *stage* of deployment is *online*. The notion of *placement* is related to online monitoring and classified as *inline*, as eBPF programs attach to the address space of the event source (of the SUS). Characterising architecture of *dottobpf* monitors warrants some deliberation. eBPF programs can be generated by *dottobpf* in either the same or in separate files. Communication and synchronisation is facilitated by map and ring buffer utilities, which are global constructs in eBPF. In that sense, the monitor may be classified as decentralised. Tracing programs residing in containers is possible if tracing is applied from the host level. Absolute path, throughout the mount, to the event source is then specified as seen from the host. However, currently, all programs must run on the same machine. Distribution across multiple machines is currently not supported.

Reaction in `dottobpf` monitors is only implemented by an issued `KILL` signal upon detection of a rejecting state, to the traced event source. Serving only as an example, this is hardly an *active* reaction — any other reaction would have to be manually implemented by the user. There`dottobpf` generated monitors are therefore classified as *passive*, with the possibility to be made active. This is addressed further in section 10.7.

Trace *role* refers to the *observed* trace and the trace *model*. Trace models for `dottobpf` monitors are expressed LTL, and consequently *infinte*. The observed trace is obtained by *event triggered sampling*. The information obtained from the SUS, which are evaluated along a trace, are occurring system *events* and possibly some internal *state*, i.e. SUS domain values.

Interference is *invasive*, considering that eBPF programs attach to SUS's events, perform logic and relinquish control upon finished execution.

Application Area, considering that thesis has focused exclusively on eBPF tracing programs, can classified as *analysis*, *debugging*, *engineering* and *information collection*.

10.2 Concurrency

Case study in chapter 8 has demonstrated the capability to monitor concurrent connections by parameterising automata. It is important to note however, that operations *within* each connection are sequential. `dottobpf` monitors do not support concurrency, and race-conditions may occur, with the limiting factor being concurrency support in eBPF tracing programs [8][2].

Implementation of spinlocks in Linux 5.1, was the first to introduce concurrency controls in eBPF. From the point of view of an eBPF program, they behave much like the ordinary kernel spinlock[1] , as shown in listing 31. As eBPF programs run in a restricted environment, a long list of rules regulate the use of spinlocks. One such rule prohibits use of spinlocks in tracing programs. With tracing, parallel threads can look up and update eBPF map fields in parallel, causing corruption where one thread overwrites the update from another. This is also known as the "lost update" problem where concurrent reads and writes overlap, causing lost updates. Tracing frameworks, bpftrace and BCC address concurrency by using per-CPU hash and array map types, where possible to avoid such

[1]For a concise introduction of spinlocks in eBPF see [18].

```
struct hash_elem {
    int cnt;
    struct bpf_spin_lock lock;
};

struct hash_elem *val = bpf_map_lookup_elem(&hash_map, &key);
if (val) {
    bpf_spin_lock(&val->lock);
    val->cnt++;
    bpf_spin_unlock(&val->lock);
}
```

Listing 31: Spinlocks in eBPF.

corruptions [29, ch. 2.3.7]. Per-CPU maps create instances for each logical CPU to use, preventing parallel threads from updating a shared location. However, unless a system model can be specifically defined from a single CPU's perspective, such a solution is insufficient for runtime verification with `dottobpf`.

10.3 In comparison to `dot2bpf`

`dot2bpf` is an (unpublished) alternative to `dot2k`, and generates eBPF code from a DFA represented in DOT language, thereby addressing the same issue as `dottobpf` [43]. In an ideal scenario, `dottobpf` would have been designed as an extension of the features provided by `dot2bpf`[2]. Unfortunately, this tool was discovered much too late in the course of this thesis, rendering it impossible to make a meaningful contribution within the allocated time.

The primary difference between these tools emerges from the choice of logic used to model system behaviour, which ultimately influences the application of runtime verification to SUS. With `dot2bpf` accepting DFA as system specification, reliance on regular expressions for system modelling is implied. `dottobpf`, on the other hand, accepts a Moore machine, and approaches runtime verification by system modelling with LTL_3. Additionally, a minor difference pertains to the expectation of the end user. Whereas `dottobpf` attempts to minimise the necessity for manual involvement after script generation, `dot2bpf` expects users to manually modify generated eBPF programs with necessary event sources.

10.4 Challenges

A persisting challenge throughout this thesis has been parsing the complex user-space objects in kernel-space.Recall that data from NGINX request and connection objects were necessary to assert propositions. The matter is further complicated that these objects are exclusively passed by reference, and eBPF enforces all memory reads through helper calls `bpf_probe_read_kernel` and `bpf_probe_read_user` in order to ensure kernel safety. The first attempted solution was to include necessary NGINX headers in kernel-code, with something as shown in listing 32:

However, such a solution inevitably leads to type definition errors with BTF, provided by `vmlinux.h`. Omitting inclusion of BTF is disadvantageous, especially in the domain of BPF CO-RE, by severely limiting type inference across kernel versions, and thereby, portability. Pursuing the solution, nonetheless, seems to confuse Clang and compilation errors arise by an attempt to include `gnu/stubs-32.h` despite `amd64` architecture. An immediate solution was not apparent [26].

Abandoning inclusion of system-wide headers, another possible solution was approached by copying the necessary types from NGINX source code into a separate header and including that

[2]Especially considering that its kernel module variant `dot2k` has been merged in the kernel [19].

```
...
#include "vmlinux.h"
#include "ngx_http.h"

SEC("uprobe//usr/sbin/nginx:ngx_http_finalize_request")
int BPF_KPROBE(handle_ngx_http_finalize_request,
               struct ngx_http_request_s* r, ngx_int_t rc)
{
    u_char sptr;
    u_char str[16];

    sptr = BPF_PROBE_READ_USER(r, request_line.data);
    int err = bpf_probe_read_user_str(str, sizeof(str), sptr);
    if (err) {
      bpf_printk("Error %d\n", err);
    }
    ...
    return 0;
}
```

.

Listing 32: Reading `char*` with BPF CO-RE. `BPF_CORE_READ` and `BPF_CORE_READ_USER` allow for simpler syntax for dereferencing pointer chains.

```
$ make
...
In file included from /usr/include/x86_64-linux-gnu/sys/types.h:25:
In file included from /usr/include/features.h:510:
/usr/include/x86_64-linux-gnu/gnu/stubs.h:7:12: fatal error: 'gnu/stubs-32.h' file not found
 # include <gnu/stubs-32.h>
```

Listing 33: Compilation errors occur when BTF is not included.

instead. However, these structs are composite, dependencies also needed to be defined[3]. Although the approach compiled successfully, only gibberish data was obtained. The return code `-EFAULT` from the `bpf_probe_read_user` call, suggests two potential reasons: either provided `sptr` value is incorrect, or a *page fault occurs, which uprobe programs cannot follow* [30]. If the pointer value is in fact, incorrect it must be due to improper type definition provided. Regardless of the effort to define the necessary types without colliding with kernel typles, faithfully representing these structures was elusive. Alternatively, if the returned `-EFAULT` is due to a page fault, the ability to block eBPF program is necessary. Fortunately, support for *sleepable uprobes* was introduced in Linux 6.0, allowing programs to sleep during execution. Pursuing the solution out of sheer interest, the kernel was recompiled to `>v6.0`, and sleeping uprobes implemented. Unfortunately, the results were only gibberish values. The only remaining solution was to implement parsing in the NGINX source, and making information available by publishing on USDT's.

What is surprisingly peculiar in this regard, is that inclusion of system-wide types is achievable with bpftrace. The script below is able to correctly parse objects and retrieve correct data, indicating that it in fact is possible to successfully work with system-wide types in eBPF. Whether this is a sensible idea, is another discussion. Figuring out exactly why, and how to achieve this with BPF CO-RE, prompted the endeavour above. A hypothesis that the differences arise due to the used compilation strategy was considered. BPF CO-RE has portability as its central tenet, and achieves this by careful integration with BTF, Clang, libbpf and the kernel [39]. Bpftrace is less stringent in this regard, and its documentation

[3]See for instance `struct ngx_http_request_s` in [42].

makes note [15]:

> [. . .] that BTF types are not available to a bpftrace program if it contains user-defined type, that redefines some BTF type. Here, ‘user-defined types’ are also types introduced via included headers.

However, do note that for tracing kernel events will most likely require kernel headers. Then, combination of user and kernel headers in the same bpftrace script will also lead to type redefinition errors.

```
#!/usr/bin/env bpftrace

#include "/usr/include/stdint.h"
#include "ngx_http.h"

uprobe:/usr/sbin/nginx:ngx_http_finalize_request
{
  $r = (struct ngx_http_request_s *)arg0;
  printf("Uri %s\n", str($r->uri.data));
}
```

Listing 34: Bpftrace accepts system-wide and user headers.

10.5 BPF CO-RE vs bpftrace

Having presented some differences between BPF CO-RE and bpftrace, it begs the question whether BPF CO-RE was more suitable option for code generation by `dottobpf`. As with all comparative questions in computer science, the answer is unequivocally ‘it depends’.

Admittedly, bpftrace is breeze to work with in comparison – it offloads concerns regarding compilation, program loading, provides a simple and efficient syntax, and enables quick assessment of solutions. All case studies in could certainly be easily replicated in bpftrace, and thereby all present functionality provided with dottobpf. However, an inherent limitation of bpftrace is that maps, by design, cannot be shared among different bpftrace scripts [1]. This, in turn enforces centralisation of the generated monitors, and limits possible extensions discussed in section 10.7. Moreover, bpftrace is by design oriented towards tracing, and the possibility of extending `dottobpf` generation to reactive systems would be limited as well. Another key area to consider is that of communication with user-space programs. While ring buffer is supported internally in eBPF, bpftrace documentation makes no mention on application of these, thus assuming that the API does not support ring buffer. This again, may be due to it being designed for tracing. On the other hand, bpftrace provides a lot of nice utilities such as writing socket data into PCAP files, providing easily accessible histograms, stacktraces, and so on. These are useful tools from an operational perspective but do not weigh as much with respect to runtime verification. Finally, it would be negligent failing to consider BPF CO-RE’s portability. bpftrace necessitates installation of LLVM, Clang and kernel header dependencies, which can consume over 100 Mbytes of storage. Eliminating these dependencies during deployment makes BPF more practical for embedded Linux environments and facilitate its adoption more broadly, as Gregg notes in [28].

Conclusively, a compelling argument that bpftrace would have been a better choice can be made, considering the current functionality `dottobpf` generated monitors provide. However, using BPF CO-RE provides a foundation for further work and research.

10.6 Performance

A critical aspect absent from this thesis is the thorough benchmarking of the overhead induced by tracing with eBPF. But alas, being merely human and ultimately subjugated to the relentless tyranny of time, a thorough insight into performance of eBPF has to be forgone. Nonetheless, light profiling was conducted to gain an overview of the landscape. For this purpose, the builtin eBPF statistics gathering has been applied. Statistics have to be enabled manually by `sudo sysctl -w kernel.bpf_stats_enabled=1`, and impact performace approximately 10-30 nanoseconds per run [10]. Once enabled they can be retrieved with `bpftool`. Profiling was performed on a vritual machine Ubuntu 22.04, 2GB RAM.

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 60
model name      : Intel Core Processor (Haswell, no TSX)
cpu MHz         : 2399.996
cache size      : 16384 KB
address sizes   : 46 bits physical, 48 bits virtual
```

```
MemTotal:        2004260 kB
MemFree:          633912 kB
MemAvailable:    1292672 kB
Buffers:           86380 kB
Cached:           606812 kB
SwapCached:            0 kB
```

Figure 10.2: Processor and memory information.

Profiling the stack solution was approached by substituting the I\O loop, for a loop that runs for a set number of iterations. Two runs were executed, one for a transition function using a static array and another using eBPF hashmap. Each run was executed with 1 000 000 000 iterations, divided equally amongst `push` and `pop` operations. Average running time was obtained by dividing the total run time with the number of firings. Results are shown in table 10.1.

	static array	hashmap
`handle_push`	221	241
`handle_pop`	222	245

Table 10.1: Average running time in nanoseconds stackcasestudy

As expected, using eBPF hash maps for transition function induce an average of 20 nanoseconds overhead per invocation. Light performance testing was done on the case of distributed system monitoring, by issuing requests to the the open `hello.com/world` endpoint. Tests were performed in three rounds, with 10k, 100k and 1m requests. Results are summarised below.

request count	10k	100k	1m
`handle_req`	7295	6982	7228
`handle_tcp_v4_connect_entry`	738	689	742
`handle_tcpconnectauthelia`	948	894	902
`handle_tcpaccepthello`	2921	2335	2618
`handle_authed`	6800	7266	7455

Table 10.2: Average running time in nanoseconds for distributed case study.

Although these tests do not permit drawing any conclusions, several observations can be made. The execution time of the probes appears to be consistent as the number of requests increases. While `handle_tcp_v4_connect_entry` and `handle_tcpaccepthello` seem to have better performance at first glance, it should be noted that these programs are triggered for all TCP connections in the system, and only one in three connections was made to Authelia[4]. Interestingly, both programs that instrument NGINX, `handle_authed` and `handle_req`, have the longest average running time. The reasons underlying such results are not immediately apparent and inspire further investigation.

Given additional time, it would be beneficial to conduct performance benchmarking using tools like `ab` or Siege to evaluate the response time latencies for each request. This assessment should be conducted under various loads to detect potential variability in execution time. Subsequently, profiling of the eBPF programs and comparison of their average runtimes to the induced latencies in response time, could provide a more solid basis for drawing conclusions.

10.7 Future Work

One area for improvement in `dottobpf` is related to parameterising propositions. To address concerns regarding variable arity in parameterised propositions, one could consider utilising a map in map types to store the state variable and propositions parameters.

To tackle the issue of potential race conditions during trace verification, one could draw upon the solution presented in [46] where similar issues were encounterd in DTrace. Rosenberg suggests separating the act of producing a trace from the act of verification. When considering that kernel-based verification may provide better performance than user-space verification by avoiding costly copies to user-space, it becomes necessary to determine precisely where and how the verification should occur in the kernel. The solution could be approached by utilising kernel's tracepoint mechanism, where 'observers' would publish data and another eBPF program would attach to the tracepoint in question, consume data and yield a verdict. Then, one should also consider how to the act of consumption of verification of the trace should be tackled in case of parameterised automata.

The case of performing runtime verification on truly distributed systems, i.e. multiple machines, with eBPF would be very exciting extension to `dottobpf`. As eBPF programs are at the time unable to make network calls, one could first consider efficient ways to do so. Subsequently, on the receiver side, one could explore the possibility using the possibility of XDP for filtering network packets, locating the packets pertaining to runtime verification at hand and consequently tail call other eBPF programs.

The scope of this thesis is exclusively focused on the tracing capabilities of eBPF. Although tracing live systems serves as the fundamental basis for runtime verification, investigating the potential of eBPF for formal runtime enforcement within the domains of security and networking, thus creating truly reactive systems from formal specification are intriguing topics.

In order to tackle problems encountered with complex user-space types in the presence of BTF, investigation on how to solve these issues could do much to facilitate tracing of user-space systems, and make eBPF into a powerful runtime verification platform.

[4]One TCP connect call from `curl`, and another towards app server.

Chapter 11

Conclusion

This thesis has introduced the tool `dottobpf`, which generates eBPF tracing programs from LTL_3 specifications, producing LTL_3-based automata for runtime verification. The capability of `dottobpf` to perform runtime verification and successfully detect property violations in SUS has been demonstrated through three case studies with varying degrees of complexity. Additionally, restrictions of LamaConv with respect to parameterisation have been tackled by the ability to provide such information through configuration. The extent of currently possible parameterisation are discussed, and suggestions for improvement provided. Moreover, limitations of eBPF encountered upon throughout the work, particularly in the context of concurrency and tracing complex user-space objects, have been presented and discussed. Such limitations naturally affect capabilities of `dottobpf`. In regard to race-conditions, possible solutions have been suggested, however, tackling the problem of user-space types requires further research.

The final form of this thesis has been shaped by the challenges faced during the project, specifically regarding tracing user values. In hindsight, admitting defeat earlier would have provided more time to explore flexibility in parameterisation and capabilities for reactive systems, or performance benchmarking. Wisdom comes easy in hindsight. Nevertheless, the exploration of these challenges has yielded valuable insight into the current state of eBPF and its applicability for runtime verification. The relevance of this work is confirmed by the increasing significance of both runtime verification and eBPF, exemplified by the emergence of other tools addressing similar problems. As such, `dottobpf` serves as a proof of concept, demonstrating both possibilities and limitations for performing runtime verification with eBPF.

Part III

Appendices

Appendix A

Installation instructions

This chapter provides installation instructions required for reproducing case studies in this thesis. Tested and verified on Ubuntu 22.04, with kernels 5.17, 5.19, 6.0, 6.2 Installation is tested on Debian 11 as well and should be similar.

A.1 Ubuntu 22.04 Setup

First check whether the kernel is compiled with `CONFIG_DEBUG_INFO_BTF`. Output should be `CONFIG_DEBUG_INFO_BTF=y`

```
$ zgrep CONFIG_DEBUG_INFO_BTF=y /boot/config-`uname -r`
```

If that does not work, try:

```
$ zgrep CONFIG_DEBUG_INFO_BTF=y /proc/config.gz
```

Should that not work, a recompilation of kernel is required. If the kernel is compiled with BTF information, proceed with installing required packages with `apt`. Packages `linux-tools-*` and `linux-headers-*` can be omitted for Debian.

```
$ sudo apt upgrade
$ sudo apt update
$ sudo apt install -y               \
    libbpf0                         \
    libbpf-dev                      \
    linux-tools-common              \
    linux-tools-generic             \
    linux-tools-`uname -r`          \
    linux-headers-`uname -r`        \
    bpftool                         \
    bpftrace                        \
    build-essential                 \
    git                             \
    gnupg                           \
    graphviz                        \
    libgraphviz-dev                 \
    lsb-release                     \
    libpcre3                        \
    libpcre3-dev                    \
    libssl-dev                      \
    pkg-config                      \
    python3-pip                 \
    software-properties-common  \
    systemtap                   \
    systemtap-sdt-dev           \
    unzip                       \
    wget
```

Install LLVM and Clang.

```
$ sudo bash -c "$(wget -O - https://apt.llvm.org/llvm.sh)"
```

The previous command install latest stable versions. At the time of writing version `15`. Verify by running `grep`. Then create a symbolic link to `clang`. The symbolic link is for facilitating the `make` command and not strictly necessary. In that case, build can be invoked with `make CLANG=<clang-cmd> LLVM_STRIP=<llvim-strip-cmd>`.

```
$ ls -al /usr/bin | grep clang
$ sudo ln -s /usr/bin/clang-<version-number> /usr/local/bin/clang
$ # verify
$ clang --version
Debian clang version 15.0.7
Target: x86_64-pc-linux-gnu
Thread model: posix
InstalledDir: /usr/local/bin
$ sudo ln -s /usr/bin/llvm-strip-<version-number> /usr/local/bin/llvm-strp
```

A.2 Install LamaConv

Install Lamaconv from [49], with following command. Do note that the URL string is broken into two lines for readability and format of the page. Remove the backslash in the URL string.

```
$ wget -O lamaconv.zip \
    https://www.isp.uni-luebeck.de/sites/default/files/content/\
        projects/lamaconv/lamaconv-snapshot-2016-09-07.zip \
    && unzip lamaconv.zip && rm lamaconv.zip
```

Jar `rltlconv.jar` should be available on your path. Invoke with:

```
$ java -jar rltlconv.jar ...
```

A.3 Dottobpf

For installation of `dottobpf` clone the repository `rv-with-bpf-deliverables` from Github or download from Zenodo.

Github

```
$ git clone --recurse-submodules --depth 1 \
    https://github.com/nela/rv-with-bpf-deliverables.git ~/rv-with-bpf-deliverables
```

Zenodo
DOI: 10.5281/zenodo.7935692.

```
$ wget -q -O- https://zenodo.org/record/7935692/files/rv-with-bpf-deliverables-mthesis.tar.gz | tar -xz
```

Note that the directory name will be `rv-with-bpf-deliverables-mthesis`. Edit accrodingly in the following steps.

Install `dottobpf`

```
$ cd dottobpf
$ make install
```

A.3.1 Usage

Dottobpf should be invoked with at least two arguments: `file.dot` and `file.yml`.

Other available options are:

- `-tftype | -tf staticarray | bpfmap` - Generates automatons transition function either as a static array or a bpf map. Defaults to static array.
- `-output | -o outputdir` - Creates a directory if it does not exist, in which to write files. Otherwise the results are printed to `stdout`.
- `-format | -f` - Formats code with `clang-format`, which should be available on your path in order for it to work.

An example invocation

Once the code is generated navigate to the directory where files are, in order to build the executable. The libbpf project has been added as submodule to the `rv-with-bpf-deliverables` and it is recommended to pass the path to this directory when invoking make. Additionally, clang and llvm-strip may be overridden with `CLANG` and `LLVM_STRIP`, respectively. Example invocation:

```
$ dottobpf stack.dot stack.yml -tf bpfmap -o generated
$ cd generated
$ make LIBBPF_SRC=~/rv-with-bpf/libbpf/src
$ # Executable monitor should now be successfully built.
$ sudo ./monitor
```

```
$ cd generated
$ make install
```

Appendix B

Setup Authentication System

This section provides installation instructions for the required software and patching in order to reproduce studies described in chapters 8 and 9.

B.1 Node.js and Hello app

Install Node.js first, then navigate to the `hello-app` directory to install necessary dependencies.

```
$ curl -sL https://deb.nodesource.com/setup_18.x | sudo -E bash -
$ sudo apt install nodejs
$ cd ~/thesis-deliverables/auth-system/hello-app
$ npm install
```

In order to start the server, from `hello-app` directory run `npm start`.

B.1.1 Install NGINX

NGINX version 1.18.0 has been used in the thesis and is recommended that the same version is used for replication. NGINX download, patching and installation can be performed with the scripts provided in `rv-with-bpf-deliverables/auth-system/nginx` directory.

The script `ngx_download_patch_configure.sh` downloads, patches runs source configuration for the server. The script `ngx_configure.sh`, first generates a self signed SSL certificate (Authelia requires an SSL certificate in order to function properly), and then copies server configuration from `rv-with-bpf-deliverables/auth-system/nginx/conf` to NGINX configuration directory at `/etc/nginx`. It will also update the `/etc/hosts` for resolution of `hello.com` and `auth.hello.com` by assigning these names to `127.0.0.1`.

```
$ cd ~/rv-with-bpf-deliverables/auth-system/nginx
$ sudo ./ngx_download_patch_configure.sh
$ make
$ make install
$ sudo ./ngx_configure.sh
```

Then verify that USDTs have been properly added. The output should be similar to the following.

```
$ readelf -n /usr/sbin/nginx
...
```

In order to start the server, run `sudo /usr/sbin/nginx`.

B.2 Authelia

Install from a `.deb` package (without debug symbols) with the following command.

```
$ wget \
    https://github.com/authelia/authelia/releases/download/v4.37.5/authelia_v4.37.5_amd64.deb \
    -O authelia_v4.37.5_amd64.deb \
    && sudo dpkg -i authelia_v4.37.5_amd64.deb \
    && rm authelia_v4.37.5_amd64.deb
```

In order to start the server, navigate to directory where configuration files reside and run:

```
$ cd ~/rv-with-bpf-deliverables/auth-system/authelia
$ authelia --config authelia_configuration.yml
```

Authentication information can be seen in `/rv-with-bpf-deliverables/auth-system/authelia`, but shortly `user1:user1` to access the restricted page on `https://hello.com/user`.

B.2.1 Building from source

First install `pnpm` and restart shell.

```
$ wget -qO- https://get.pnpm.io/install.sh | sh -
```

Then
Then navigate to `authelia` directory, where a bootstrap script is waiting for you. The script will install golang, prior to cloning and building authelia. Remove these lines if you already have go `>v.1.18`.

```
$ cd ~/rv-with-bpf-deliverables/auth-system/authelia
$ sudo ./bootstrap_build_from_source.sh
```

You may need to change permissions to your user since the script needs to be executed in sudo mode. Protip. Running authelia when building from source might compromise internal database schema, due to different dependency versions. If schema errors occur, simply delete the generated `db.sqlite3` file.

Appendix C

Reproduction Instructions for Case Study I: Simple Stack

Instructions provided here will guide through necessary steps to reproduce the results of case study in chapter 7. The guide assumes all steps in appendix A have been completed successfully.

C.1 Install Stack

Navigate to `stack-implementation` directory and run make. This will create the stack executable `.bin/stack` in the same directory and create a symbolic link to `/usr/local/bin/stack`.

```
$ cd stack-implementation
$ make
[sudo password for .. ]
[ -d /Users/nela/master/code/stack/bin ] || mkdir /Users/nela/master/code/stack/bin
gcc -o /Users/nela/master/code/stack/bin/stack -Wall -Wextra -O0 stack.c
sudo ln -sf /Users/nela/master/code/stack/bin/stack /usr/local/bin/stack
```

Run `make clean` to remove binary and symbolic link.

C.2 Generate monitors with `dottobpf`

Navigate to `rv-with-bpf-deliverables/cs_stack` where necessary configuration for `dottobpf` is provided. Invoke `dottobpf`, preferably with `LIBBPF_SRC` path. Note that directories `generated_map` and `generate_array` already exist and contains the executable `monitor`. You can even try running the executable and test the portability of BPF CO-RE yourself! Or just skip generation and build with make directly.

```
$ dottobpf stack.dot stack.yml -tf bpfmap -o mygenerated
$ cd mygenerated
$ make LIBBPF_SRC=~/rv-with-bpf/libbpf/src
$ # Executable monitor should now be successfully built.
$ sudo ./monitor
```

Then, start up the stack manually and try commands such as

- `push 1`
- `empty`
- `pop`

Or navigate back to the `stack-implementation` and redirect output from example files.

```
$ cd stack-implementation
$ stack <1000commands.in
```

The running monitors should print out results. In order to try correct implementation of stack, edit the push function in `stack-implementation/stack.c` by removing the comment on line 13 which should update the buffer index. Run `make` and start the monitor again.

C.3 Generate automata with LamaConv

For replicating automata generation with LamaConv the file `cs_stack/stack.ltl` contains the necessary string that can be copied. To convert to a png, the `dot` command can be used. Or to get a figure directly, invoke `rltlconv.jar` with -png option instead of `-dot`.

```
$ java -jar rltlconv.jar \
    "LTL=G( "LTL=G( (push && F empty) -> (!(empty) U pop) )" \
    --formula \
    --moore \
    --min \
    --dot > mystack.dot

$ # make png
$ dot -Tpng:cairo mystack.dot -o mystack.png
```

Appendix D

Reproduction Instructions for Case Study II: Distrubuted System

Instructions provided here will guide through necessary steps to reproduce the results of case study in chapter 8. The guide assumes all steps in appendices A and B have been completed successfully. Start up the servers as described.

D.1 Generate monitors with `dottobpf`

Navigate to `rv-with-bpf-deliverables/cs_distributed` where necessary configuration for `dottobpf` is provided. Invoke `dottobpf`, preferably with `LIBBPF_SRC` path. Then, apply the provided patch required for successful tracing of TCP connect system calls. Note that directory `generated` and `generated_patched` already exists, and the `generated_patch` contains the executable `distributed`. You can even try running the executable and test the portability of BPF CO-RE yourself! Or build with make directly.

```
$ dottobpf distributed.dot distributed.yml -tf staticarray -o mygenerated
$ patch mygenerated/distributed.bpf.c <distributed.patch
$ cd mygenerated
$ make LIBBPF_SRC=~/rv-with-bpf-deliverables/libbpf/src
$ sudo ./distributed
```

Then try running requests to `https://hello.com/world` and `https://hello.com/user`. Simulate fault by accessing the `hello-app` directly on `localhost:3000`.

D.2 Generate automata with LamaConv

For replicating automata generation with LamaConv the file `cs_distributed/distrbuted.ltl` contains the necessary string that can be copied. To convert to a png, the `dot` command can be used. Or to get a figure directly, invoke `rltlconv.jar` with `-png` option instead of `-dot`. Note that the string is broken into several lines for readbility and formatting. Remove backslashes from the string.

```
$ java -jar rltlconv.jar \
    "LTL=G( ((req && F tcpaccepthello) -> (!tcpaccepthello W (tcpconnectauthelia && X authed) ) )  \
        && (authed -> X (tcpaccepthello && X req) ) ) \
        && !(authed || tcpconnectauthelia || tcpaccepthello) W req" \
    --formula \
    --moore \
    --min \
    --dot > distributed.dot
```

```
$ # make png
$ dot -Tpng:cairo distributed.dot -o distributed.png
```

Appendix E

Reproduction Instructions for Case Study II: Concurrrent Conenctions

Instructions provided here will guide through necessary steps to reproduce the results of case study in chapter 9. The guide assumes all steps in appendices A and B have been completed successfully. Start up the servers as described.

E.1 Generate monitors with `dottobpf`

Navigate to `rv-with-bpf-deliverables/cs_distributed` where necessary configuration for `dottobpf` is provided. Invoke `dottobpf`, preferably with `LIBBPF_SRC` path. Then, apply the provided patch required for successful tracing of TCP connect system calls. Note that directory `generated` and contains the executable `connections`. You can even try running the executable and test the portability of BPF CO-RE yourself! Or skip generation and build with make directly.

```
$ dottobpf connections.dot connections.yml -tf staticarray -o mygenerated
$ cd mygenerated
$ make LIBBPF_SRC=~/rv-with-bpf-deliverables/libbpf/src
$ sudo ./connections
```

Then try running requests to `https://hello.com/world` and `https://hello.com/user`.

E.2 Generate automata with LamaConv

For replicating automata generation with LamaConv the file `cs_connections/connections.ltl` contains the necessary string that can be copied. To convert to a png, the `dot` command can be used. Or to get a figure directly, invoke `rltlconv.jar` with -png option instead of `-dot`. Note that the string is broken into several lines for readbility and formatting. Remove backslashes from the string.

```
$ java -jar rltlconv.jar \
    "LTL=httpconn && X(verifyauth && X ( verifyauth U \
        (authed && X (upstreamhello U httpclose R (!httpconn && !upstreamauthelia ) ) \
            || httpclose ) ) || upstreamauthelia U httpclose)" \
    --formula \
    --moore \
    --min \
    --dot > connections.dot

$ # make png
$ dot -Tpng:cairo connections.dot -o connections.png
```

Bibliography

[1] *Access map from different bpftrace scripts*. [Online; accessed 14. May 2023]. 2023. URL: https://github.com/iovisor/bpftrace/discussions/2495.

[2] *Atomicity of Map operation helpers*. [Online; accessed 13. May 2023]. 2018. URL: https://github.com/iovisor/bcc/issues/1521#issuecomment-680185387.

[3] *Authelia. Architecture - Overview*. [Online; accessed 12. May 2023]. 2022. URL: https://www.authelia.com/overview/prologue/architecture.

[4] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. Cambridge, Massachusetts; London, England: The MIT Press, 2008. ISBN: 026202649X.

[5] Ezio Bartocci et al. 'Introduction to Runtime Verification'. In: *Lectures on Runtime Verification*. Vol. 10457 LNCS. Lecture Notes in Computer Science. Cham, Switzerland: Springer Verlag, 2013, pp. 1–34.

[6] Andreas Bauer, Martin Leucker and Christian Schallhart. 'Runtime Verification for LTL and TLTL'. In: vol. 20. 2011, 14:1–14:64.

[7] Dragan Bosnacki and Anton Wijs. 'Model checking: recent improvements and applications'. In: *International Journal on Software Tools for Technology Transfer* 20 (2018).

[8] *bpf-helpers(7) - Linux manual page*. [Online; accessed 13. May 2023]. 2023. URL: https://www.man7.org/linux/man-pages/man7/bpf-helpers.7.html.

[9] *bpftool — ManPage: tool for inspection and simple manipulation of eBPF programs and maps*. [Online; accessed 15. Apr. 2023]. URL: https://www.mankier.com/8/bpftool.

[10] *bpftool features from Quentin Monnet*. [Online; accessed 14. May 2023]. 2023. URL: https://gist.github.com/rafaeldtinoco/5c23b32f5409a955387e6c369d42fc40.

[11] *bpftool-btf — ManPage: tool for inspection of BTF data*. [Online; accessed 15. Apr. 2023]. URL: https://www.mankier.com/8/bpftool-btf.

[12] *bpftool-gen — ManPage: tool for BPF code-generation bpftool*. [Online; accessed 15. Apr. 2023]. URL: https://www.mankier.com/8/bpftool-gen.

[13] *bpftool-map: tool for inspection and simple manipulation of eBPF maps*. [Online; accessed 15. Apr. 2023]. URL: https://www.mankier.com/8/bpftool-map.

[14] *bpftool-prog — ManPage: tool for inspection and simple manipulation of eBPF progs*. [Online; accessed 15. Apr. 2023]. URL: https://www.mankier.com/8/bpftool-prog.

[15] *Bpftrace Reference Guide - BTF Support*. [Online; accessed 13. May 2023]. 2023. URL: https://github.com/iovisor/bpftrace/blob/master/docs/reference_guide.md#btf-support.

[16] J. Richard Büchi. 'On a Decision Method in Restricted Second Order Arithmetic'. In: Stanford, CA.: Stanford University Press, 1960, pp. 1–12.

[17] David Calavera and Lorenzo Fontana. *Linux Observability with BPF: Advanced programming for Performance Analysis and Networking*. Sebastopol, Ca.: O'Reilly Media, Inc., 2019.

[18] Jonathan Corbet. *Concurrency management in BPF*. [Online; accessed 13. May 2023]. 2021. URL: `https://lwn.net/Articles/779120`.

[19] *Deterministic Automata Monitor Synthesis — The Linux Kernel documentation*. [Online; accessed 14. May 2023]. URL: `https://docs.kernel.org/trace/rv/da_monitor_synthesis.html`.

[20] Matthew B. Dwyer. *Temporal Specification Patterns*. [Online; accessed 12. May 2023]. URL: `https://matthewbdwyer.github.io/psp`.

[21] *eBPF for Windows*. [Online; accessed 10. Apr. 2023]. URL: `https://github.com/microsoft/ebpf-for-windows`.

[22] Jake Edge. *Unifying kernel tracing*. [Online; accessed 9. Apr. 2023]. 2019. URL: `https://lwn.net/Articles/803347`.

[23] Yliès Falcone et al. 'A taxonomy for classifying runtime verification tools'. In: *International Journal on Software Tools for Technology Transfer* 23 (2021).

[24] Michael Fisher. *An Introduction to Practical Formal Methods Using Temporal Logic*. Chichester, West Sussex, PO19 8SQ, United Kingdom: John Wiley & Sons, Ltd, 2011. DOI: `https://doi.org/10.1002/9781119991472`.

[25] Francesco Fuggitti. *LTLf2DFA*. Version 1.0.0.post0. Mar. 2019. DOI: `10.5281/zenodo.3888410`. URL: `https://doi.org/10.5281/zenodo.3888410`.

[26] *gnu/stub-32.h required on amd4*. [Online; accessed 13. May 2023]. 2023. URL: `https://github.com/libbpf/libbpf-bootstrap/discussions/133`.

[27] Valentin Goranko and Antje Rumberg. *Temporal Logic*. Last Updated: Feb 2020. 1999. URL: `https://plato.stanford.edu/entries/logic-temporal/#ForModTim`.

[28] Brendan Gregg. *BPF binaries: BTF, CO-RE, and the future of BPF perf tools*. [Online; accessed 9. May 2023]. Nov. 2020. URL: `https://brendangregg.com/blog/2020-11-04/bpf-co-re-btf-libbpf.html`.

[29] Brendan Gregg. *BPF Performance Tools: Linux System and Application Observability*. 1st. Addison-Wesley Professional, 2019.

[30] *How to access user space function arguments in bpf?* [Online; accessed 13. May 2023]. 2023. URL: `https://stackoverflow.com/questions/74703910/how-to-access-user-space-function-arguments-in-bpf`.

[31] *Inline (Using the GNU Compiler Collection (GCC))*. [Online; accessed 10. Apr. 2023]. URL: `https://gcc.gnu.org/onlinedocs/gcc/Inline.html`.

[32] IoVisor. *BCC: tcpconnect.py*. [Online; accessed 13. Apr. 2023]. 2022. URL: `https://github.com/iovisor/bcc/blob/master/tools/tcpconnect.py`.

[33] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. Notes Series NS-01-1. Available from `http://www.brics.dk/mona/`. BRICS, Department of Computer Science, University of Aarhus. Jan. 2001.

[34] Fred Kroger. *Temporal logic of programs*. Berlin; New York: Springer-Verlag, 1987. ISBN: 3540170308.

[35] Orna Kupferman and Moshe Vardi. 'Model Checking of Safety Properties'. In: vol. 19. 2001, pp. 291–314. DOI: `https://doi.org/10.1023/A:1011254632723`.

[36] Orna Kupferman, Moshe Y. Vardi and Pierre Wolper. 'An Automata-Theoretic Approach to Branching-Time Model Checking'. In: Washington, DC, USA: IEEE Computer Society Press, 1986, pp. 332–345.

[37] Martin Leucker. ‘Teaching Runtime Verification’. In: *Runtime Verification*. Vol. 7186 LNCS. Lecture Notes in Computer Science. 2nd International Conference on Runtime Verification, RV 2011 ; Conference date: 27-09-2011 Through 30-09-2011. Springer Verlag, 2012, pp. 34–48.

[38] Martin Leucker and Christian Schallhart. ‘A brief account of runtime verification’. In: *The Journal of Logic and Algebraic Programming* (2009), pp. 293–303. URL: `https://www.sciencedirect.com/science/article/pii/S1567832608000775`.

[39] Andrii Nakryiko. *BPF CO-RE (Compile Once — Run Everywhere)*. [Online; accessed 13. May 2023]. Feb. 2020. URL: `https://nakryiko.com/posts/bpf-portability-and-co-re`.

[40] Andrii Nakryiko. *BPF CO-RE reference guide*. [Online; accessed 10. Apr. 2023]. 2021. URL: `https://nakryiko.com/posts/bpf-core-reference-guide`.

[41] Andrii Nakryiko. *Building BPF applications with libbpf-bootstrap*. [Online; accessed 13. Apr. 2023]. 2020. URL: `https://nakryiko.com/posts/libbpf-bootstrap`.

[42] *Nginx*. [Online; accessed 13. May 2023]. 2023. URL: `https://github.com/nginx/nginx/blob/master/src/http/ngx_http_request.h#L377`.

[43] Daniel Bristot de Oliveira. ‘dot2bpf’. [Online; accessed 8. April 2023]. 2023. URL: `https://bristot.me/efficient-formal-verification-for-the-linux-kernel/`.

[44] Daniel Bristot de Oliveira, Tommaso Cucinotta and Rômulo Silva de Oliveira. ‘Efficient Formal Verification for the Linux Kernel’. In: *Software Engineering and Formal Methods*. Cham: Springer International Publishing, 2019, pp. 315–332.

[45] Amir Pnueli. ‘The temporal logic of programs’. In: *18th Annual Symposium on Foundations of Computer Science*. 1977, pp. 46–57. DOI: `10.1109/SFCS.1977.32`.

[46] Carl Martin Rosenberg. ‘Leveraging DTrace for Runtime Verification’. University of Oslo, 2016.

[47] Michael Royal. *eBPF-Guide*. [Online; accessed 10. Apr. 2023]. URL: `https://github.com/mikeroyal/eBPF-Guide`.

[48] *rv — The Linux Kernel documentation*. [Online; accessed 14. May 2023]. URL: `https://docs.kernel.org/6.2/tools/rv/rv.html`.

[49] Torben Scheffel et al. *LamaConv—Logics and Automata Converter Library*. [Online; accessed 9. May 2023]. 2016. URL: `https://www.isp.uni-luebeck.de/lamaconv`.

[50] Erik Seligman, Tom Schubert and M V Achutha Kiran Kumar. ‘Formal Verification’. In: Boston: Morgan Kaufmann, 2015. ISBN: 978-0-12-800727-3. DOI: `https://doi.org/10.1016/B978-0-12-800727-3.00010-1`.

[51] Michael Sipser. *Introduction to the theory of computation*. Boston, Ma.: Cengage Learning, 1996.

[52] ‘Temporal assertions for sequential and concurrent programs’. PhD thesis. 2007, pp. 1–133.

[53] *Unable to attach to Go programs*. [Online; accessed 13. May 2023]. 2023. URL: `https://github.com/libbpf/libbpf-bootstrap/discussions/179#discussioncomment-5888941`.

[54] *uv_stream_t – Stream handle – libuv documentation*. [Online; accessed 13. May 2023]. 2023. URL: `http://docs.libuv.org/en/v1.x/stream.html`.

[55] Moshe Y. Vardi. ‘An automata-theoretic approach to linear temporal logic’. In: *Logics for Concurrency: Structure versus Automata*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 238–266.

[56] *What Is NGINX? - NGINX*. [Online; accessed 4. May 2023]. 2023. URL: `https://www.nginx.com/resources/glossary/nginx`.