SAARLAND UNIVERSITY
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

MASTER'S THESIS

Model Checking Timed Hyperproperties

Author
Jens Heinen

Advisor
Christopher Hahn

Supervisor
Prof. Bernd Finkbeiner, Ph.D.

Reviewers
Prof. Bernd Finkbeiner, Ph.D.
Prof. Dr.-Ing. Holger Hermanns

Submitted: 06th June 2018

Statement in Lieu of an Oath
I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Declaration of Consent
I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Eidesstattliche Erklärung
Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Einverständniserklärung
Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird

Date/Datum	Signature/Unterschrift

Abstract

Timing attacks have become fairly known in context of the Meltdown exploit, where attackers used the vulnerability of CPU caches against timing attacks on kernel space data. The threat of timing attacks on privacy-critical systems raises the need of time dependent analysis and checking specific properties on these systems. A well known example of a timing attack is the extraction of the password from a bad password checking algorithm, which stops processing the input on the first index where the saved password and the user input differ. We can create a simple MITL property for a timing-attack free system ("$G^{[2,2]}$ answer"). The property ensures that the password checker always has to give the answer exactly after two time units, independently of the correctness of the input. However, this excludes traces where the password checker can give the answer earlier.

To improve the response time of the password checker while remaining timing-attack free, an improved property can be formulated as follows: "Every pair of execution traces agree on the answer and give an answer at least in two time steps.". This cannot be expressed in MITL as it is a hyperproperty. A hyperproperty is a set of sets of traces and hence relates multiple computation traces. HyperLTL is a temporal logic to express hyperproperties. The example property additionally has a timing constraint, which in turn cannot be formalized in HyperLTL.

In this thesis a new logic for timed hyperproperties is constructed. Properties to prevent timing attacks on password checking systems and cryptographic systems are formulated and checked on suitable models. The corresponding model checking problem for Timed HyperLTL is studied together with its complexity.

Acknowledgements

I am very thankful to my advisor Chris for always being helpful and patient with any of my problems. His support during the last year when working on this thesis was above all of any expectation.

I am also very grateful to Prof. Finkbeiner for offering me this thesis and for the great discussions about time.

Special thanks go to Dominic, Michael and Julian for proofreading this work. I would also like to thank Prof. Hermanns for reviewing this thesis.

The last and biggest thanks go to an unknown person. Thank you for your support in all matters.

Contents

1 Introduction

The analysis of security critical systems is a matter of enormous interest, as our daily life is widely influenced by the fast development of information and communication technologies. However, coming up with formal correctness guarantees for systems satisfying security policies is not trivial. A main issue of secure information flow is noninterference between high input and low input, that shall be accomplished by the implementation. Checking whether information flow in a system is correct, secure and invulnerable to a specific attack is important. This ensures systems to deny flow of information or data that do not meet the boundaries of the policy.

Some policies for security critical systems can already be formalized by *trace properties*. Trace properties are sets of execution traces. They define the traces that a system, satisfying the property, may permit. Trace properties can be distinguished in two categories: safety and liveness. Safety properties state, that something bad will never happen. Liveness properties force, that something good will happen eventually. An example for a safety property is the following:

"A critical section can only be entered, if the the program holds the lock on the data."

This trace property can formally be written in Linear-time Temporal Logic (LTL):

$$\square(enter \rightarrow lock)$$

LTL was introduced by Amir Pneuli in 1977 [23]. LTL formulas can describe the future of paths referring to abstract time terms. Every LTL formula describes a trace property. Trace properties are sets of traces, that can only prescribe individual traces. However, trace properties are restricted to only one execution trace at a time. Many important security policies, especially information flow policies, cannot be formalized as a trace property. Observational determinism is an example for this:

"Every pair of traces with the same initial low observation remain indistinguishable for low users."

Whether one specific trace is allowed by the property depends on the other trace of the pair. This implies that observational determinism cannot be expressed by LTL.

HyperLTL [9] has been introduced to fill this gap. The logic raises the expressiveness of LTL to a higher level by adding explicit trace quantification. With this, one can quantify about the traces and thus reason about the relation of two or multiple traces at a time. The properties expressed by HyperLTL are *hyperproperties*. In contrast to trace properties, hyperproperties are sets of trace properties, i.e. sets of sets of execution traces. They have been introduced by Michael Clarkson and Fred Schneider [8]. With HyperLTL, one can formalize observational determinism:

$$\forall \pi_1 . \forall \pi_2 . \Box (in_{\pi_1} \leftrightarrow in_{\pi_2}) \rightarrow \Box (out_{\pi_1} \leftrightarrow out_{\pi_2})$$

Observational determinism is an information flow property as it connects a system's input and output with each other. HyperLTL is a simple logic in which many information flow properties can be stated [9].

However, systems satisfying hyperproperties, e.g. observational determinism, are not completely secure against attackers. In the last few years many attacks have become publicly known, which were used to break into critical systems. Many of them were timing attacks, which misuse faulty behaviour of the system to have different runtime for different inputs. By measuring the difference of runtime for multiple inputs, an attacker can extract information about the quality of the sent queries and then compute the real secrets used in the system. Hyperproperties in general cannot express time of system executions, which are crucial for preventing timing attacks. In this thesis, we introduce a logic for *timed hyperproperties*, which are an extension of hyperproperties by adding time constraints to the temporal expressions. To describe timed hyperproperties, Timed HyperLTL is presented and its expressiveness is shown on several example properties.

Especially password checking algorithms have a high risk to be vulnerable to timing attacks. If the respective algorithm is vulnerable, an attacker can infer the secret password by multiple queries and measuring the response time for the query. A timed hyperproperty, which a password checker has to satisfy to be secure against a timing attack, can be formalized as follows:

$$\forall \pi_1, \pi_2 .\ \Box^{[0,\infty]}\, resp_{\pi_1} \rightarrow \Diamond^{[0,4]}\, resp_{\pi_2}$$

The property allows a system to have a delay of maximal four time units between the responses on the first and the second execution trace. If the respective timing attack is not efficient, i.e. if the responses only have a difference lower than a given threshold, then the property described above effectively protects systems against the attacks.

Another well known timing attack on OpenSSL has been described by Boneh and Brumley [7]. They built an environment in the SSL handshake protocol, where an attacking client sends decryption queries to the server. The queries are initially guessed and iteratively improved to approximate the secret by measuring the time needed to decrypt the queries. The essential part of the handshake model

$$
\begin{aligned}
\forall \pi_1, \pi_2 . \square^{[0,\infty]}(&CKeyExchange_{\pi_1} \leftrightarrow CKeyExchange_{\pi_2}) && (1)\\
&\wedge(CKeyExchange_{\pi_1} \rightarrow (\square^{[0,0]} \neg fail_{\pi_1} \wedge \Diamond^{[t,\infty]} SFinish\pi_1)) && (2)\\
&\wedge(CKeyExchange_{\pi_1} \rightarrow (\square^{[0,\infty]} decrypt_{\pi_1} \leftrightarrow decrypt_{\pi_2})) && (3)
\end{aligned}
$$

Property 1.1: Safety property to prevent SSL timing attack

is depicted in Figure 1.0.1. A possible restriction to prevent a timing attack is described by Property 1.1.

Property 1.1 consists of multiple parts, which are explained more detailed in the following:

(1) The first part requires the system to synchronously end the *ClientKey-Exchange* run, such that both traces can start the decryption method at the same time.

(2) The second part indicates that the system must go into the *ServerFinish* state some time after the *ClientKeyExchange* is done.

(3) Finally, the last part enforces two traces to synchronously decrypt the input query and send the result back. Decrypting and responding is seen as one atomic action in this property. Atomic actions may need more than one time unit for being performed. This can also be stated more granular, what makes the prevention property more complex.

In company of the new logic, we also present a model checking algorithm for Timed HyperLTL, that is build upon the well-known and widely used real-time

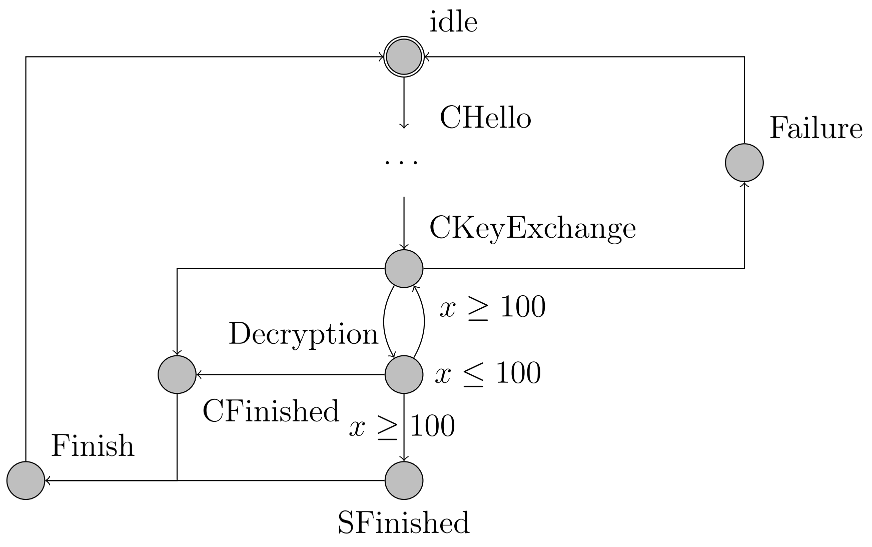

Figure 1.0.1: Section of the UPPAAL model for SSL Handshake protocol, secure against the timing attack of [7]

model checker Uppaal for timed automata [4]. It is generally based on the model checking algorithm for HyperLTL. The approach builds an automaton for the formula to be checked and applies a composition of the automaton and the system to create a new Büchi automaton with states as tuples of atomic propositions. The algorithm then maps the trace properties to the respective positions of the state tuples. To check whether the property is satisfied or not, the algorithm checks the language of the resulting automaton on emptiness: if the language is non-empty, the property holds.

Our model checking algorithm works similarly: We compute a self-composition of the real-time system in dependency of the number of trace quantifiers in the formula to be checked. This results in a new system with states as tuples of sets of atomic propositions, where each position is a projection of the respective trace quantifier. Then, we can handle this timed system to UPPAAL and adapt the Timed HyperLTL to the UPPAAL verifier language.

Related Work

Nielson et al. [21] describe timed automata as a 'formalism for modelling and analysing the real-time safety aspects of cyberphysical systems.' A language-based approach has been implemented, called Timed Command Language, with which information flow control can be defined in a type system for Timed Automata. Similar to MITL, the authors only consider non-zeno and time-divergent trace behaviours, as only those traces reflect relevant real-world behaviour. Information-flow control is defined over mappings from lattices to program variables, where partial order of a complete lattice is given by the directed partial order of the lattice. The paper uses *Timed Commands*, build upon the Guarded Commands of Dijkstra combined with clocks, to develop a type system for a programming language. Given a program written in Timed Commands, the next step of the approach translates the commands into a a timed automata, which is done along several translation rules for the elementary commands.
In contrast to this, Timed HyperLTL does not include a type system for the formulas, but uses the standard syntax and semantics of the already presented and well-studied linear-time temporal logics LTL and HyperLTL.

Focardi et al [14] propose an extension to a prior work of the authors to formulate noninterference properties in a real-time setting. The *Security Process Algebra* (SPA) represents the basis of this approach. For this, the authors define the new logic tSPA, which is a timed variant of SPA. The semantics of tSPA are defined on a LTS. tSPA uses discrete time where actions are atomic and durationless. Additionally, it uses an *idling* operator that allows processes to wait indefinitely. With the new logic the authors lift the noninterference property into a real-time framework.
The real-time setting of tSPA is similar to our approach: We also use atomic actions and delay actions that elapse time. However, a difference is the explicit synchronization step done in tSPA by the *tick* action, where the global clock is

synchronized whenever all processes agree on the *tick* action. Timed HyperLTL defines the synchronizing on runs by the synchronous elapsing of time in the respective runs. Hence, the logic does not need to have an extra synchronize step in the runs of a system.

Köpf et al. [16] worked on analysing information flow in hardware circuits in a timing-sensitive manner. They created a decision procedure for the security of hardware implementations. The goal is to find possible ways of timing attacks on the system and how to prevent them by posing policies on the system behaviour. The paper concentrates on analysing information flow of synchronous timed hardware circuits. A first and important point is the relation between runtime of the system and the used secrets.
A similar experiment is done for Timed HyperLTL as we will see in Chapter 5, where we investigate the model checking of a system depending on the size of the secret. Additionally, our work is flexible enough to be applicable to more settings than just hardware circuits as it is a generic approach to any environment.

Nguyen et al. [20] take a somewhat similar approach to describe timed systems as it is done in this work. The authors take STL (Signature Temporal Logic) and add existential $\exists$ and universal $\forall$ trace quantifiers to relate multiple execution traces. The work concentrates on the signal domain of cyberphysical systems (CPSs) and expresses safety and security hyperproperties in the resulting logic, HyperSTL. In companion to the new logic, the authors present a falsifying algorithm for checking the hyperproperties for CPSs. The authors also consider side-channel attacks on CPS's, whereas they also take anomalies in power consumption and heat generation into account, as well as execution time.
While the work of Nguyen focus on the cyberphysical part, our work concentrates on specifying properties for software in embedded real-time systems. We can only check for anomalies in execution time as we cannot measure power consumption or heat generation by a timed automata.

The work of Finkbeiner et al. [13] shows the enormous potential of the temporal logics for hyperproperties. It introduces the model checking algorithm for the temporal logics HyperLTL and HyperCTL*, which builds the basis for our work. Their work is an automata-based algorithm to check hyperproperties on finite state systems. They also distinguish the alternation free fragment from formulas with an alternation depth of at least one. The authors present an efficient model checking algorithm for HyperCTL*.

Outline

The thesis is structured as follows: In Chapter 2, we will introduce the required definitions and constructions needed for building the new logic and implement the model checking algorithm. Chapter 3 will introduce the new logic by presenting syntax and semantics of Timed HyperLTL. Several example policies for security relevant issues created from the new logic will also be presented. After that, Chapter 4 will discuss the model checking algorithm for Timed HyperLTL.

In Chapter 5, we will show experimental results obtained from running our model checker on timed hyperproperties described so far and some real-world examples. Finally, Chapter 6 concludes our findings and gives an outlook on possible future work.

2 Background

In this chapter, we outline the background definitions for this thesis. In Section 2.1, we introduce timed automata and Kripke structures as a description for the systems to be modelled. Furthermore, we introduce the temporal logics LTL, MITL and HyperLTL, which is used to describe hyperproperties, in Section 2.2. We establish the formal definition of hyperproperties in Section 2.3. Based on this, we point out the differences of the logics with respect to their expressibility and why we need to combine them to achieve the formalization of timed hyperproperties.

2.1 Preliminaries

A timed automata is an automaton with an additional finite set of real-time clocks. During a run of the automaton, clock-values can be compared to time constants via *clock constraints*.

Definition 2.1 (Timed Automata). *A Timed Automata is a tuple* $T = (Loc, Act, C, \hookrightarrow, Loc_0, Inv, AP, L)$*, where*

- *Loc is a finite set of locations, Loc_0 is a set of initial locations,*
- *Act is a finite set of actions,*
- *C is a finite set of clocks,*
- *$\hookrightarrow \subseteq Loc \times CC(C) \times Act \times 2^C \times Loc$ is a transition relation, where $CC(C)$ is the set of clock constraints,*
- *$Inv : Loc \to CC(C)$ is an invariant-assignment function,*
- *AP is a finite set of atomic propositions,*
- *$L : Loc \to 2^{AP}$ is a labeling function for the locations.*

The transition $l \xhookrightarrow{c',a,X} l'$ is taken with action a if clock constraint c evaluates to true under clock evaluation η. If the transition is taken, all clocks $x \in X$ are reset to 0.

A state of a timed automaton is given as $s = \langle l, \eta \rangle$, where l is a location of the timed automaton and η is the current clock evaluation of state s. The semantics of temporal logics like LTL or CTL can be defined over the paths of a Kripke structure.

Definition 2.2 (Kripke Structure). *A Kripke structure $K = (S, s_o, \delta, AP, L)$ is given by the set of states S, the initial state s_0, a transition function $\delta : S \to 2^S$, the set AP of atomic propositions and the labelling function $L : S \to 2^{AP}$. It is additionally required that: $\forall s \in S : \delta(s) \neq \emptyset$.*

A Kripke structure [17] is a general representation for programs. The nodes represent reachable states of the system and the transitions are the possible state transitions. One can describe terms of temporal logics by traces of a Kripke structure. The requirement $\forall s \in S : \delta(s) \neq \emptyset$ ensures that all runs of the Kripke structure result in infinite traces.

Definition 2.3 (Self-composition). *The self-composition of a Kripke Structure $K = (S, s_0, \delta, AP, L)$ is computed by applying the cross product on S and adding new transitions according to the tuples of states. The* ***n-times self-composed*** *Kripke Structure $K' = (S', \boldsymbol{s_0}, \delta', AP', L')$ is defined as follows:*

- $S' = \{\boldsymbol{s} \mid \boldsymbol{s} = (s_1, s_2, \dots, s_n)\} = S \times^n S$,
- $\boldsymbol{s_0} = (s_0, s_0, \dots)$,
- $\delta' : S' \to 2^{S'}$,
- $AP' = AP$,
- $L' : S' \to 2^{AP'}$

For a state tuple $\boldsymbol{s} = (s_1, s_2, \dots, s_n)$ the transition function is defined as follows:

$$\delta'(\boldsymbol{s}) = (s_1, s_2, \dots, t_i, \dots, s_n) \text{ iff } \exists t_i \text{ s.t. } \delta(s_i) = t_i \text{ and}$$
$$\delta'(\boldsymbol{s}) = \boldsymbol{t} \text{ iff } \forall s_i \in \boldsymbol{s}\ \exists t_i \in \boldsymbol{t} \text{ s.t. } \delta(s_i) = t_i.$$

The transition function δ' returns the set of tuples of states that are reachable from the state tuple $\boldsymbol{s}$, by applying the standard transition function δ to the single states of the tuple $\boldsymbol{s}$.

As shown by Yovine [27], a timed automaton with one clock is a special case of a Kripke structure. This induces that self-composition of a Kripke structure can be applied to timed automata in a similar way. The clock constraints of transitions and states are added according to the respective positions of state tuples and the new transitions. For a new transition t' of the self-composed system a clock constraint is added if:

$$t' = \boldsymbol{l} \xhookrightarrow{c',a,X} \boldsymbol{l'} \text{ iff } \exists t \in \delta_S \text{ with } t = l \xhookrightarrow{c',a,X} l' \quad \text{where } l \in \boldsymbol{l} \wedge l' \in \boldsymbol{l'}.$$

Remember that ***l*** and ***l'*** are tuples of sets of S.

2.2 LTL – HyperLTL – MITL

LTL and HyperLTL are both linear-time temporal logics in contrast to CTL* and HyperCTL*, which are branching-time temporal logics. In this section, we give an overview of these temporal logics and discuss the differences in their expressiveness especially regarding security relevant properties. Additionally, we show MITL as an extension of LTL with timing constraints.

LTL

We first introduce LTL (Linear-Time Temporal Logic). It has been invented by Amir Pnueli for the formal verification of computer programs [23]. LTL formulas are based on the grammar depicted in Figure 2.2.1.

$$\psi ::= a \quad | \quad \psi \wedge \psi \quad | \quad \psi \vee \psi \quad | \quad \neg\psi \quad | \quad \bigcirc\psi \quad | \quad \psi \,\mathcal{U}\, \psi$$

Figure 2.2.1: Syntax of LTL

a is an atomic proposition, the logical operators are defined as usual. The temporal operators are $\bigcirc$ *next* and $\mathcal{U}$ *until*. From these, we can derive other operators, namely *eventuall (finally)*:

$$\Diamond\psi \equiv \text{true}\ \mathcal{U}\,\psi$$

and *globally*

$$\Box\psi \equiv \neg\Diamond\neg\psi \equiv \neg\,(\text{ true } \mathcal{U}\,\neg\psi)$$

The semantics of a LTL formula is defined over traces. A trace is a sequence of atomic propositions. For an infinite trace $t = s_0 s_1 s_2 \ldots$ and index $i \in \mathbb{N}$, we use the following index notations:

$$\begin{aligned} t[i] &\equiv s_i, \\ t[..i] &\equiv s_0 s_1 \ldots s_i, \\ t[i..] &\equiv s_i s_{i+1} \ldots \end{aligned}$$

A set of traces modelling a system S is called *executions* of S. The following equations define the semantic of LTL:

Definition 2.4 (LTL semantics).

$$\begin{array}{lll}
t \vDash a & \textit{iff} & a \in t[0] \\
t \vDash \psi_1 \wedge \psi_2 & \textit{iff} & t \vDash \psi_1 \wedge t \vDash \psi_2 \\
t \vDash \neg\psi & \textit{iff} & t \nvDash \psi \\
t \vDash \bigcirc \psi & \textit{iff} & t[1,\infty] \vDash \psi \\
t \vDash \psi_1 \mathcal{U} \psi_2 & \textit{iff} & \exists i \geq 0 \ : t[i,\infty] \vDash \psi_2 \wedge \forall 0 \leq j < i : \ t[j,\infty] \vDash \psi_1
\end{array}$$

LTL formulas define ω-regular languages: Each LTL formula can be translated into an equivalent Büchi automaton over the alphabet $\Sigma = 2^{AP}$ that accepts precisely the traces that satisfy the formula [19, 26].

HyperLTL

HyperLTL, introduced by Clarkson et al. in [9], is an extension of LTL by adding explicit trace quantification. The syntax shown in Figure 2.2.2 generates formulas of HyperLTL with the initial symbol φ.

$$\begin{array}{lllll}
\varphi ::= \exists\pi.\varphi \mid & \exists\pi.\psi & \mid \neg\varphi & & \\
\psi ::= \ a_\pi & \mid \ \psi \vee \psi & \mid \ \neg\psi & \mid \ \bigcirc\psi & \mid \ \psi \ \mathcal{U} \ \psi
\end{array}$$

Figure 2.2.2: HyperLTL syntax

The grammar defines HyperLTL formulas by a quantifier prefix with at least one quantifier and a following subformula, which is a standard LTL formula.The all-quantifier is defined as $\forall\pi.\psi \equiv \neg\exists\pi.\neg\psi$. This relation between the existential and universal quantifier is essential for the model checking algorithm of HyperLTL and Timed HyperLTL, as we will see in Chapter 4. Note that for HyperLTL formulas without quantifier alternation, we use the following abbreviations:

$$\forall\pi_1, \forall\pi_2, \ldots, \forall\pi_n \equiv \forall\pi_1, \ldots, \pi_n \text{ and}$$
$$\exists\pi_1, \exists\pi_2, \ldots, \exists\pi_n \equiv \exists\pi_1, \ldots, \pi_n.$$

Definition 2.5 (HyperLTL semantics).

$$\begin{array}{lll} \Pi \vDash \exists \pi.\, \psi & \textit{iff} & \exists t \in Traces(K, s_0) : \Pi[\pi \mapsto t] \vDash \psi \\ \Pi \vDash a_\pi & \textit{iff} & a \in \Pi(\pi)(0) \\ \Pi \vDash \neg\psi & \textit{iff} & \Pi \nvDash \psi \\ \Pi \vDash \psi_1 \vee \psi_2 & \textit{iff} & \Pi \vDash \psi_1 \textit{ or } \Pi \vDash \psi_2 \\ \Pi \vDash \bigcirc \psi & \textit{iff} & \Pi[1, \infty] \vDash \psi \\ \Pi \vDash \psi_1 \mathcal{U} \psi_2 & \textit{iff} & \exists i \geq 0 : \Pi[i, \infty] \vDash \psi_2 \textit{ and } \forall 0 \leq j < i : \Pi[j, \infty] \vDash \psi_1 \end{array}$$

The semantics of HyperLTL are defined over a Kripke structure. A Kripke structure K satisfies a HyperLTL formula ψ : $K \vDash \psi$ iff $s_0 \vDash \psi$. A *path* p of K is a sequence $s_0 s_1 s_2 \cdots \in S^\omega$ with s_0 as initial state and it holds: $\forall i \in \mathbb{N} : s_{i+1} = \delta(s_i)$. A *trace* t of a path $p = s_0 s_1 \ldots$ is the sequence of the state labeling : $t = \sigma_1 \sigma_2 \ldots$ with $\sigma_i = L(s_i)$ $\forall i \in \mathbb{N}$. $Traces(K, s)$ is the set of all traces of Kripke structure K, starting in state s.

As Alur et al. showed in [2], the property describing observational determinism is not a regular tree property, so it is not expressable in LTL. Additionally, LTL is obviously a sublogic of HyperLTL, so it is subsumed. You can write every LTL formula as a HyperLTL formula with a single leading forall quantifier. However, observational determinism is expressible in HyperLTL as we can relate multiple execution traces.

MITL

MITL (Metric Interval Temporal Logic) is another extension of LTL, where timing constraints in form of intervals are added to the temporal operators [3]. MITL is a fragment of MTL (Metric Temporal Logic), where punctuality is not inherited. Thus, MITL is a decidable fragment, while model checking and the satisfiability for MTL is in general undecidable [22].

MITL formulas are generated by the following grammar:

$$\psi ::= a \quad | \quad g \quad | \quad \neg\psi \quad | \quad \psi \vee \psi \quad | \quad \psi\, \mathcal{U}^{[a,b]}\, \psi$$

The other boolean connectives and temporal operators can be defined by standard conventions. For this work, we stipulate $a, b \in \mathbb{N}$ as it is specified for the decidable fragment. It is also possible and straightforward to choose $\mathbb{R}$ as domain for time intervals.

Definition 2.6 (MITL semantics). *Let* $TA = (Loc, Act, C, \hookrightarrow, Loc_0, Inv, AP, L)$ *be a timed automaton,* $a \in AP, g \in ACC(C)$ *and* $[a, b] \subseteq \mathbb{N}_{>0}$. *With* $s = \langle l, \eta \rangle \in$

$TS(TA)$ and the MITL state formula ψ and path formula ϕ, we define the semantics of MITL as follows:

$$\begin{array}{lll}
s \vDash a & \textit{iff} & a \in L(l) \\
s \vDash g & \textit{iff} & \eta \vDash g \\
s \vDash \neg\psi & \textit{iff} & s \nvDash \psi \\
s \vDash \psi_1 \vee \psi_2 & \textit{iff} & s \vDash \psi_1 \vee s \vDash \psi_2 \\
\pi \vDash \phi_1 \mathcal{U}^{[a,b]} \phi_2 & \textit{iff} & \exists i \geq 0. s_i + d \vDash \psi_2 \textit{ for some } d \in [0, d_i] \textit{ with} \\
& & \quad \sum_{k=0}^{i-1} d_k + d \in [a,b] \quad \textit{and} \\
& & \forall j \leq i. s_j + d' \vDash \psi_1 \vee \psi_2 \textit{ for any } d' \in [0, d_j] \textit{ with} \\
& & \quad \sum_{k=0}^{j-1} d_k + d' \leq \sum_{k=0}^{i-1} d_k + d
\end{array}$$

For $s_i = \langle l_i, \eta_i \rangle$ and $d \geq 0$ we define $s_i + d = \langle l_i, \eta_i + d \rangle$.
We can rewrite the timed temporal operators *eventually* and *globally* similar to LTL in this way:

$$\Diamond^{[a,b]} \psi \equiv \text{ true } \mathcal{U}^{[a,b]} \psi \qquad \Box^{[a,b]} \psi \equiv \neg \Diamond^{[a,b]} \neg\psi$$

For time-divergent path $\pi = s_0 \xrightarrow{d_0} s_1 \xrightarrow{d_1} \ldots$, the following two equations hold

- A timed *eventually* formula $\Diamond^{[a,b]} \psi$ is satisfied if ψ is true at some time instant $t \in [a,b]$.

$$\pi \vDash \Diamond^{[a,b]} \psi \quad \text{iff} \quad \exists i \geq 0.\ s_i + d \vDash \psi \text{ for } \textbf{some } d \in [0, d_i] \text{ with } \sum_{k=0}^{i-1} d_k + d \in [a,b].$$

- A timed *globally* formula $\Box^{[a,b]} \psi$ is satisfied if ψ holds on all time instants $t \in [a,b]$.

$$\pi \vDash \Box^{[a,b]} \psi \quad \text{iff} \quad \exists i \geq 0. s_i + d \vDash \psi \text{ for } \textbf{any } d \in [0, d_i] \text{ with } \sum_{k=0}^{i-1} d_k + d \in [a,b].$$

2.3 Hyperproperties

Trace properties are sets of traces. Thus, a system satisfying a trace property ϕ has at least all traces τ that satisfy the property ϕ. A hyperproperty is a set of sets of traces, so a set of trace properties. Security policies, for example nonintеference, are not policies of individual traces, but of unspecified trace variables since the hyperproperty claims restrictions on all possible runs of the system.

If a security policy is given as a hyperproperty H, then H is the set of all systems, satisfying the security policy, as each element of the hyperproperty is a trace property that specifies the allowed executions of a system. A *trace property* P is defined by a set of traces T that consists of the traces fulfilling the property:

$$T \vDash P \equiv T \subseteq P$$

This is a first step of formalizing the satisfaction of a hyperproperty H by a trace property T:

Definition 2.7. *Given a trace property T and a hyperproperty H. Then it holds:*

$$T \vDash H \text{ iff } T \in H$$

As trace properties are subsumed by hyperproperties, it is also possible to express every trace property by a hyperproperty as it is shown by [8].

Theorem 2.1 (Lift of trace properties [8]). *The lift of a trace property P to a hyperproperty H is given as the power-set of the trace property:*

$$[P] \equiv \mathbb{P}(P)$$

where $\mathbb{P}$ denotes the powerset function.

This can be extended to get a relation between the satisfaction of hyperproperties and trace properties by a system, as the following Theorem 2.2 shows.

Theorem 2.2. *A system satisfying a hyperproperty H that contains the trace properties $H = \{P_1, \ldots, P_n\}$ also satisfies all trace properties of the hyperproperty:*

$$S \vDash H \to \forall i = 0, \ldots, n \; S \vDash P_i$$

Trace properties can always be written as intersection of a *liveness* and a *safety* property [1]. This also holds for every hyperproperty, which can be split into a safety hyperproperty (*hypersafety*) and a liveness hyperproperty (*hyperliveness*) as shown by Clarksond et al. [8].

3 Timed HyperLTL

The intended use of the new logic is the formalization of timed hyperproperties. For this, Timed HyperLTL formulas are built upon the syntax of HyperLTL and MITL. Our logic extends HyperLTL with timing constraints on temporal operators, which are defined in MITL. Therefore, we can express hyperproperties, where parts of the property are valid from a time point onwards or in the bounds of a time interval.

We define the syntax and semantics of Timed HyperLTL in Section 3.1 and 3.2 and describe how the standard temporal operators can be derived in the time constraint environment. Additionally, we examine how to express the untimed temporal operators with the new logic. In Section 3.3, we present example timed hyperproperties which can be expressed by Timed HyperLTL.

3.1 Syntax

Timed HyperLTL formulas are defined by the following grammar:

$$\begin{array}{llllllllll} \varphi = & \forall \pi.\, \varphi & \mid & \exists \pi.\, \varphi & \mid & \psi & & & & \\ \psi = & a_\pi & \mid & \psi \wedge \psi & \mid & \psi \vee \psi & \mid & \neg \psi & \mid & \psi\ \mathcal{U}^{[a,b]}\ \psi \end{array}$$

Figure 3.1.1: Syntax of Timed HyperLTL

The formula $\psi_1\ \mathcal{U}^{[a,b]}\ \psi_2$ means that ψ_1 has to hold from time step a until time step b. From time step $b+1$ onwards, ψ_2 has to hold. $\forall$ and $\exists$ have the usual meanings as universal and existential trace quantifiers, derived from HyperLTL. The same holds for the logical operators, which have their classical definition. The atomic proposition a_π holds if the first state in π is true for a, i.e. $a \in \pi[0]$.

We only consider one global clock c for all execution traces in a system checked with a Timed HyperLTL formula. This results in running all traces of the system synchronously. Note that the time of one clock tick may vary depending on the environment. For our later examples we will use different notions of time unit, e.g. one time unit may be equivalent of comparing two bits with each other, see

Section 5.1. This is the reason why we omit the *next* operator $\bigcirc a$ of HyperLTL. The definition of a "next step" is not clear in the real-time setting and may change. A possible definition of $\bigcirc a$ could be true $\mathcal{U}^{[1,1]}\, a \equiv \diamondsuit^{[1,1]}\, a$, to simulate the next step with an interval of one time unit. This means, that a has to hold exactly one time unit after the current state. However, we use the interval definitions of MITL which prohibits singular intervals, i.e. intervals $I = [a, b]$ with $a = b$, because the language then gets undecidable.

3.2 Semantics

Similar to HyperLTL, Timed HyperLTL defines the semantics over a set T of traces. $\Pi : \mathcal{V} \to T$ is a trace assignment with $\mathcal{V}$ as the set of trace variables. We write $\Pi[\pi \mapsto \tau]$ for the updated trace assignment, where π is mapped to τ. For the validity judgement, we write the trace set T as subscript since the validity depends on the trace mapping. For the sake of decidability, we assume that $x < y$. The semantics of Timed HyperLTL are defined in Figure 3.2.1.

$$\begin{array}{ll}
\Pi \vDash_T a_\pi & \text{iff } a \in \Pi(\pi)[0] \\
\Pi \vDash_T \psi_1 \wedge \psi_2 & \text{iff } \Pi \vDash_T \psi_1 \wedge \Pi \vDash_T \psi_2 \\
\Pi \vDash_T \psi_1 \vee \psi_2 & \text{iff } \Pi \vDash_T \psi_1 \vee \Pi \vDash_T \psi_2 \\
\Pi \vDash_T \neg\psi & \text{iff } \Pi \nvDash_T \psi \\
\Pi \vDash_T \forall\pi.\ \psi & \text{iff } \forall\tau \in T : \Pi[\pi \mapsto \tau] \vDash_T \psi \\
\Pi \vDash_T \exists\pi.\ \psi & \text{iff } \exists\tau \in T : \Pi[\pi \mapsto \tau] \vDash_T \psi \\
\Pi \vDash_T \psi_1 \mathcal{U}^{[x,y]}\, \psi_2 & \text{iff } \exists z \in \mathbb{R}^+ :\ \Pi \vDash_{T+z} \psi_2 \text{ such that } z \in [x, y] \\
 & \quad \text{and } \exists z' \in \mathbb{R}^+ : x \leq z' < z\ : \Pi \vDash_{T+z'} \psi_1
\end{array}$$

Figure 3.2.1: Semantics of Timed HyperLTL

A Kripke structure K, defined in Chapter 2, satisfies a Timed HyperLTL formula ϕ iff all traces of K, denoted by $Traces(K)$, satisfy ϕ, i.e. $Traces(K) \vDash \phi$. $Traces(K)$ denotes the set of all traces starting in the initial state s_0 of K and $\forall i \geq 0\ s_{i+1} \in \delta(s_i)$. We use $\tau[i]$ to denote element i of trace τ, where $i \in \mathbb{N}$. A trace position $\tau[i]$ is composed of a subset of atomic propositions which hold in this position and the current delay value: $\tau[i] = (X_i, d_i)$, where $X_i \subseteq AP$ and $d_i \in \mathbb{R}^+$ such that $d_i = \sum_{j=0}^{i} d_j$ for $d_j \in \tau[j]$. This means for the validity of an atomic proposition $\Pi \vDash_T a_\pi$ iff $a \in X$. We need the following definition:

$$\tau + z = (X, d) + z = (X, d + z)$$

With this, we can define $T + z$ as $\forall\tau \in T : \tau + z$. We then get $\Pi \vDash_{T+z} \psi$ iff $\forall\tau \in T : \tau + z \vDash \psi$. The above definition of adding time to trace set is crucial as we declare at this point to synchronize all traces. Another definition of the $\mathcal{U}$ operator could allow comparison of asynchronous runs of the system, which is shortly discussed in Chapter 6.

Deriving temporal operators

With the semantics of $\mathcal{U}^{[x,y]}$ we can derive the typical temporal operators:

$$\begin{aligned}
\Diamond^{[x,y]}\,\psi &\equiv \mathit{true}\;\mathcal{U}^{[x,y]}\,\psi\\
\Box^{[x,y]}\,\psi &\equiv \neg\,\Diamond^{[x,y]}\,\neg\psi\\
\psi_1\,\mathcal{W}^{[x,y]}\,\psi_2 &\equiv \psi_1\,\mathcal{U}^{[x,y]}\,\psi_2 \vee \Box^{[x,y]}\,\psi_1\\
\psi_1\,\mathcal{R}^{[x,y]}\,\psi_2 &\equiv \neg\left(\neg\psi_1\,\mathcal{U}^{[x,y]}\,\neg\psi_2\right)
\end{aligned}$$

The *Finally* and *Globally* operators differ slightly in their semantic on an equal interval:

$$\begin{aligned}
\Pi \models \Diamond^{[0,4]}\,\psi &\quad \text{iff } \exists z \in [0,4] :\; \Pi \models_{T+z} \psi\\
\Pi \models \Box^{[0,4]}\,\psi &\quad \text{iff } \forall z \in [0,4] :\; \Pi \models_{T+z} \psi
\end{aligned}$$

It is crucial to carve out the differences of formulas depending on different time intervals of temporal operators.

$$\begin{aligned}
\Pi \models \Diamond^{[0,0]}\,\psi &\quad \text{iff } \Pi \models_T \psi\\
\Pi \models \Box^{[0,0]}\,\psi &\quad \text{iff } \Pi \models_T \psi
\end{aligned}$$

which implies the following:

$$\Diamond^{[0,0]}\,\psi \quad\equiv\quad \Box^{[0,0]}\,\psi \quad\equiv\quad \psi.$$

We can also connect the new timed temporal operators and the standard temporal operators of HyperLTL:

$$\begin{aligned}
\psi_1\,\mathcal{U}\,\psi_2 &\equiv \psi_1\,\mathcal{U}^{[0,\infty]}\,\psi_2\\
\Box\,\psi &\equiv \Box^{[0,\infty]}\,\psi\\
\Diamond\,\psi &\equiv \Diamond^{[0,\infty]}\,\psi
\end{aligned}$$

The equations given above underline that HyperLTL is a sub-fragment of Timed HyperLTL. We can express every HyperLTL formula as a Timed HyperLTL formula by replacing every temporal operator with an operator equipped with the interval $[0,\infty]$.

3.3 Example Policies

In this section, we show the potential of Timed HyperLTL as well as some timed hyperproperties related to information flow policies. Most of the hyperproperties discussed by [8] can be extended by a timing constraint on their respective temporal operators and thus, are expressible in Timed HyperLTL. We perform this for the classical information flow properties as observational determinism, noninterference and declassification.

Timed Noninference

Noninference states that any change in the high input may not influence the low-observable output. This is expressed by HyperLTL Property 3.1:

$$\forall \pi_1 . \exists \pi_2 . (\square\, \lambda_{\pi_2}) \land O_{\pi_1} \leftrightarrow O_{\pi_2}$$

Property 3.1: Standard Noninference

With the first variant of timed noninference, we force the public output not to be influenced by dummy inputs for at least t time units. After that, the output may differ on one trace to the output on the other trace.

$$\forall \pi_1 . \exists \pi_2 . \square\, \lambda_{\pi_2} \land \Diamond^{[0,t]}\, O_{\pi_1} \leftrightarrow O_{\pi_2}$$

Property 3.2: TNI1 - Timed Noninference, Variant 1

This first variant of timed noninference allows a delay in the output behaviour of the system of t time units. The following trace τ would satisfy the first variant of timed noninference. λ denotes the dummy input and in the standard input. As the property has two trace variables, we have 2-tuple states after applying the self-composition onto the system. We depict the full structure of the trace here.

$$\overbrace{(\{in\},0),(\{\lambda\},0)}^{\tau[0]}\ \overbrace{(\{in\},1),(\{\lambda\},1)}^{\tau[1]}\ldots\overbrace{(\{in\},t),(\{\lambda\},t)}^{\tau[t]}\ \overbrace{(\{b\},t+1),(\{a\},t+1)}^{\tau[t+1]}$$

In this case, the delay steps are in measures of one time unit. Therefore, the delays agree with the position indexes of the trace. For the rest of the thesis, we will leave out the delay values in the trace tuples if they can be intuitively derived.

We can also give another variant of timed noninference, where we restrict the time span for dummy inputs. This is shown in Property 3.3. The second variant of timed noninference would not be satisfied by the above trace, because the outputs

$$\forall \pi_1 . \exists \pi_2 . (\square^{[0,4]} \lambda_{\pi_2}) \land \square^{[0,\infty]} O_{\pi_1} \leftrightarrow O_{\pi_2}$$

Property 3.3: Timed Noninference, Variant 2

of π_1 and π_2 do not coincide on all positions. This violates variant two of timed noninference. Trace τ_1 satisfies Property 3.3.

$$\tau_1 = (\{in\}, \{\lambda\})(\{in\}, \{\lambda\})(\{in\}, \{\lambda\})(\{in\}, \{\lambda\})(\{\mathbf{a}\}, \{\mathbf{a}\})(\{\mathbf{a}\}, \{\mathbf{a}\}) \dots$$

We give a third variant of timed noninference to demonstrate how the meaning of the property is influenced by the scope of the time constraint. Property 3.4 depicts the corresponding Timed HyperLTL formula.

$$\forall \pi . \exists \pi_2 . \square^{[0,4]} ((\lambda_{\pi_2}) \land O_{\pi_1} \leftrightarrow O_{\pi_2})$$

Property 3.4: Timed Noninference, Variant 3

Neither of the traces seen so far satisfy this Timed HyperLTL property. We give a satisfying trace below.

$$\tau_2 = (\{in\}, \{\lambda\})(\{in\}, \{\lambda\})(\{in\}, \{\lambda\})(\{in\}, \{\lambda\}) \dots$$

The crucial point is, that the scope of the $\square^{[0,4]}$ operator ranges over the complete LTL fragment of the Timed HyperLTL formula. So a system satisfying the last property has to agree in its observable output on all execution traces for four time steps. As the above trace only has input atomic propositions, this is trivially true. The below trace satisfies Property 3.4, too:

$$\tau_3 = (\{in\}, \{\lambda\})(\{in\}, \{\lambda\})(\{a\}, \{a\})(\{a\}, \{a\})(\{b\}, \{a\}) \dots$$

The above trace shows, that the timing constraint has either limited the possible prefixes for traces or, for the output case, any extensions of traces of the system. Both cases together illustrate a natural extension of the result shown in [8] for hyperproperties. They can always be written as the intersection of hypersafety and hyperliveness properties. The same holds for timed hyperproperties.

Theorem 3.1. *Timed HyperLTL properties can be formulated as intersections of liveness and safety timed hyperproperties.*

Proof. This results from reformulating the timing constraints on the temporal operators as clock constraints. Then, the new formula is essentially a HyperLTL formula for which the theorem is already shown [8]. ■

Timed Observational Determinism

Observational Determinism forces a system to behave deterministic in the low-security view. The observable output of such a system must be deterministic depending on the public observable input. Therefore, no trace pair (π_1, π_2) may exist with same low input but different low output. The HyperLTL property is depicted in Property 3.5.

$$\forall \pi_1, \pi_2.\ \Box(I_{\pi_1} = I_{\pi_2}) \rightarrow \Box(O_{\pi_1} = O_{\pi_2})$$

Property 3.5: Standard Observational Determinism

To satisfy the hyperproperty, the traces for π_1 and π_2 have to agree on the low input and output respectively. The respective trace τ_4 of the self-composed system may look like the following:

$$\tau_4 = (\{in\}, \{in\}) \dots (\{in\}, \{in\})(\{b\}, \{b\})(\{b\}, \{b\})(\{\mathbf{a}\}, \{\mathbf{a}\})(\{\mathbf{a}\}, \{\mathbf{a}\}) \dots$$

With the timed variant of observational determinism shown in Property 3.6, we admit the output to be equal once in a four seconds interval.

$$\forall \pi_1, \pi_2 \Box^{[0,0]}(I_{\pi_1} \leftrightarrow I_{\pi_2}) \rightarrow \Diamond^{[0,4]}(O_{\pi_1} \leftrightarrow O_{\pi_2})$$

Property 3.6: TOD1 - Timed Observational Determinism

We illustrate the satisfying trace τ_5 below.

$$\tau_5 = (\{in\}, \{in\})(\{a\}, \{b\})(\{b\}, \{b\})(\{b\}, \{a\}) \dots$$

The property forces the system to give an output at most 4 time steps after the input arrived. Compared to the standard Observational Determinism, this is a weaker property with respect to restrictiveness, because the standard hyperproperty forces the output to be globally equal, whereas the timed version is satisfied if this happens at one position of the trace.

To carve out the differences between $\Box^{[0,4]}$ and $\Diamond^{[0,4]}$, a second variant for timed observational determinism is considered in Property 3.7.

$$\forall \pi_1, \pi_2 \Box^{[0,0]}(I_{\pi_1} \leftrightarrow I_{\pi_2}) \rightarrow \Box^{[0,4]}(O_{\pi_1} \leftrightarrow O_{\pi_2})$$

Property 3.7: Timed Observational Determinism, Variant 2

Now the output has to agree on all positions of the execution traces for the time span of four units, illustrated by the trace below:

$$\tau_6 = (\{in\}, \{in\})(\{a\}, \{a\})(\{b\}, \{b\})(\{b\}, \{a\}) \dots$$

Trace τ_6 again shows the relation between the structure of the time intervals and the structure of the allowed traces of the system. Note that the property is satisfied, although the atomic propositions in the last shown position (b, a) do not agree. This does not harm the satisfiability since the position is out of the time scope.

Timed Declassification

A password checker may reveal whether the entered password is correct or incorrect. A classical variant of declassification can be expressed by the following HyperLTL formula:

$$\forall \pi_1, \pi_2.\ (I_{\pi_1[0]} = I_{\pi_2[0]} \wedge \bigcirc(pw_{\pi_1} \leftrightarrow pw_{\pi_2})) \rightarrow O_{\pi_1} \leftrightarrow O_{\pi_2}$$

Property 3.8: Standard Declassification

As described in Section 3.2, there is no unique translation of the next operator $\bigcirc$ to Timed HyperLTL as in a continuous setting. We allow a delay in revealing the correctness of the password. This is given by the Timed HyperLTL formula in Property 3.9. A system satisfying this property is still preserving declassification

$$\forall \pi_1, \pi_2 \left(I_{\pi_1[0]} = I_{\pi_2[0]} \wedge \Diamond^{[0,4]}\, declass^{[0,1]}_{(\pi_1,\pi_2)}(pw) \right) \rightarrow O_{\pi_1} \leftrightarrow O_{\pi_2}$$

Property 3.9: TD1 - Timed Declassification

if two traces declassify the correctness of the password both in a time interval of one time unit. This is a much weaker property as the classical version because it permits leaking the correctness of the password in a non-atomic way. If the threshold for a possible timing attack on the password checker is known, the bounds of the intervals can be adapted. The definition of the proposition *declass* is given below.

$$declass^{[x,y]}_{(\tau_1,\tau_2)}(a) \equiv \left((a_{\tau_1} \rightarrow \Diamond^{[x,y]}\, a_{\tau_2}) \wedge a_{\tau_1} \right)$$

Property 3.10: Propositional Formula for Declassification of a

The trace τ_7, given below, would satisfy Property 3.9. Let "{_}" denote any possible atomic proposition occurring in the system and pw the declassification of the entered password.

$$\tau_7 = (\{in\},\{in\})(\{\boldsymbol{pw}\},\{\_\})(\{\_\},\{\boldsymbol{pw}\})(\{\_\},\{\_\})(\{b\},\{a\})(\{out\},\{out\})\ldots$$

As pw appears on π_1 earlier, pw must hold on π_2 at most one time step after, because of the time constraint $[0, 1]$ on the declassification proposition. However, Property 3.9 also features another constraint, namely $\diamondsuit^{[0,4]}$ at the beginning of the *declass* scope. Therefore, we can give more traces satisfying the property.

$$\tau_8 = (\{in\},\{in\})(\{\boldsymbol{pw}\},\{\_\})(\{\_\},\{\boldsymbol{pw}\})(\{\_\},\{\_\})(\{out\},\{out\})\dots$$
$$\tau_9 = (\{in\},\{in\})(\{\_\},\{\_\})(\{\boldsymbol{pw}\},\{\_\})(\{\_\},\{\boldsymbol{pw}\})(\{out\},\{out\})\dots$$
$$\tau_{10} = (\{in\},\{in\})(\{\boldsymbol{pw}\},\{\boldsymbol{pw}\})(\{\_\},\{\_\})(\{\_\},\{\_\})(\{out\},\{out\})\dots$$

By comparing the above traces, we see the role of the second time constraint $I_1 = [0,1]$ in relation to $\diamondsuit^{[0,4]}$ as I_1 is relative to $[0,4]$. $[0,1]$ does not result in the absolute point zero in time but in the range zero to one time steps after the starting point of $\diamondsuit^{[0,4]}$. This is interesting as we only have one global clock in the synchronous setting, but still allow these kinds of constraints. To realize the correct checking of the timed hyperproperties we need to consider the relative bounds of all constraints for the global clock. This will further be discussed in Chapter 6.

Of course τ_{10} satisfies the property also, as we do not prohibit the declassification to occur synchronously. The definition of $\diamondsuit^{[0,1]}$ allows the immediate start of *declassify* on trace π_1.

$$\tau_{11} = (\{in\},\{in\})(\{\boldsymbol{pw}\},\{\_\})(\{\_\},\{\_\})(\{\_\},\{\boldsymbol{pw}\})(\{b\},\{a\})(\{out\},\{out\})\dots$$

Note that trace τ_{11} does also satisfy Property 3.9. The declassification of the password on trace π_2 did not appear in the allowed time bounds of one time step after declassification on π_1. But therefore the premise of the timed hyperproperty is false, which makes the complete implication true. We give a short explanation why this is intuitive:
It is not important for the security of the system to have equal outputs on two traces if the inputs have been different. Neither it is important if the declassification of the password did not appear in the given interval. The same argument holds for the following trace τ_{12} satisfying Property 3.9, too.

$$\tau_{12} = (\{in\},\emptyset)(\{\boldsymbol{pw}\},\{in\})(\{\_\},\{\_\})(\{\_\},\{\boldsymbol{pw}\})(\{b\},\{a\})(\{out\},\{out\})\dots$$

The trace has a delay for input on trace π_2 later than occurring on trace π_1. The property forced $I_{\pi_1[0]} = I_{\pi_2[0]}$, so the trace makes the premise of the implication false and the complete implication true.

In context of this setting, one could think of relaxing the declassification premise by allowing delays of inputs.

Timing attacks

We describe some timing attacks on security systems in the following section. More detailed experiments are depicted in Section 5.2. This section essentially contains the timed hyperproperties used to check if systems satisfy the properties and hence, are not vulnerable to specific attacks.

Cache-timing attacks

Timing attacks may not only be based on the different execution times of algorithms, but also on the behaviour of the memory. A special case are cache-timing attacks where the runtime of a system depends on cache hits and cache misses.

A special attack is described by Bernstein [5] on the AES specification. AES (Advanced Encryption Standard) is a block cipher used worldwide for encryption tasks. It uses multiple lookup tables in the encryption definition. The attack described by the author bases on the relation between a single array lookup and the whole AES computation time depending on all table lookups. By measuring the time for lookups with different indexes, one can retrieve information about the used index, which partly consists of the encryption key, i.e. $k[0] \oplus n[0]$, where n is the input. The same argument holds for the other parts of the key $k[1] \oplus n[1], k[2] \oplus n[2]$ and so on. The attacker then can compute $k[0]$ by observing the suitable value for $k[0] \oplus n[0]$ where the overall AES computation time is maximal, and additionally by approximating the value for $n[0]$ to get the maximal computation time.

The different computation times of table lookups depend on the cache behaviour and the RAM, as it is possible to have lower lookup time for different inputs. The runtime also depends, whether the table lookup is done on cache lines or in the RAM. Therefore, the author suggests in his conclusion to use a constant time encryption standard, where *constant* means the AES implementation to be independent of key k and input n. This is connected to widely more problems than just resolving the timing issue of AES. It would be a huge trade-off for high-class and middle-class hardware if a constant time encryption standard was widely used.

At this point we can leverage the problem of stating timing restrictions on systems implementing the AES encryption. We do this by providing timed hyperproperties for systems to prevent the described timing attack above. The crucial point of the cache-timing attack is that different indexes for table queries result in different result times. The timed hyperproperty shown in Property 3.11 requires a system to retrieve all results for a table lookup in a given time interval for t units.

$$\forall \pi_1 . \forall \pi_2 .\ \square (query_{\pi_1} \leftrightarrow query_{\pi_2}) \rightarrow \Big(res_{\pi_1} \rightarrow \Diamond^{[0,t]}\, res_{\pi_2} \Big)$$

Property 3.11: Preventing cache-timing attack

The atomic proposition *query* is satisfied if the system gets an input query with an index of the form $k[i] \oplus n[i]$. *res* stands for the event to have finished the computation and sending the response to the inquirer.

The huge difference between demanding a constant-time encryption mechanism and restricting the system to a minimal allowed delay is the amount of trade-off between efficiency and security. In our case, this trade-off depends on the dimensions of the response time. Assume the attacker is only capable to retrieve useful information if the response time is in units of 10 milliseconds. Then our property can set $t = 5$ms such that many systems, satisfying the timed hyperproperty, are still considered to be secure.

Attack on RSA & Diffie-Hellmann

Diffie-Hellmann and RSA (Rivest-Shamir-Adleman) are methods used in private-key encryption systems. They both use exponentiation operations to hide the plaintext of data and compute the corresponding ciphertext. We will present a model of the SSL handshake protocol, which uses an implementation of RSA, in Section 5.2 and will explain the RSA algorithm in more detail.

Diffie-Hellmann is an approach for secure key exchange between Alice and Bob on a public channel. It works as follows:

- Choose two numbers p prime number and g a primitive root of p.
- Alice chooses an arbitrary integer a and computes $A = g^a$. Then Alice sends (g, p, A) to Bob.
- Bob chooses an arbitrary integer b and computes $B = g^b$. Then Bob sends (B) to Alice.
- After that, Bob additionally computes $K = A^b$, whereas Alice computes $K = B^a$. Both agree on the secret key $K = g^{a \cdot b}$.

The essential part of the key exchange is the computation of A and B by exponentiation of a prime number. Finding the exponent by an inverse attack is in general computationally expensive. However, if the implementation of those schemes is vulnerable, the attacks can be efficiently performed. The aim of an attacker is extracting the secrets a, b from the interaction between Alice and Bob by eavesdropping their public communication. g and p are public values, which the attacker hence knows. The attacker can measure the response time of sending (g, p, A) to Bob and thus, is able to deduce the exponents a and b bitwise. For an attacker knowing bits $0 \ldots k-1$ of a it is possible to efficiently compute bit k and so the complete exponent.

The attack shown here is taken from Kocher [15]. The paper contains more detailed information on the structure of the attack and the application environment.

Based on this attacking description, we can already define another timed hyperproperty, that protects the system against the described attack.

"Any two executions of modular exponentiation in Diffie-Hellmann encryption has to take the same amount of time."

We formalize the property in Timed HyperLTL as depicted in Property 3.12.

A system satisfying Property 3.12 must execute the encryption tasks, i.e. the exponentiation described above, synchronously and in time lower than t time units, on all runs. This ensures for all runs to have the same runtime for the

$$\forall\pi_1.\forall\pi_2.\ (in_{\pi_1} \leftrightarrow in_{\pi_2}) \rightarrow \square^{[0,t]}(encryption_{\pi_1} \leftrightarrow encryption_{\pi_2})$$

Property 3.12: Prevent the timing attack on Diffie-Hellmann

exponentiation. Obviously, the untimed version of the hyperproperty also prevents the timing attack. However, the extension with the time constraint narrows the possible runs of the system. This results in less runs, which conversely carry out the exponentiation only in the given time span.

At this point we want to explicitly show the difference of $\square^{[0,t]}$ and $\square$, where the last operator is equivalent to the timed variant $\square\,[0,\infty]$. We therefore consider two traces: The first one, τ_1 satisfies Property 3.12, and the second one fulfills Property 3.12, where the timed operator is replaced by $\square$.

$$\tau_1 = (\{in\},\{in\})(\{encrypt\},\{encrypt\})\overbrace{\ldots}^{t-2\text{ times}}((\{encrypt\},t-1),(\{encrypt\},t-1))\ldots$$
$$\tau_2 = (\{in\},\{in\})(\{encrypt\},\{encrypt\})\overbrace{\ldots}^{t\text{ times}}((\{encrypt\},t+1),(\{encrypt\},t+1))\ldots$$

In the second trace, the encryption task is allowed to run longer than t time units. Although the traces satisfying the property still have to synchronously be in *encryption* state, the overall runtime of encryption is longer than in the time constraint variant.

The above explanation almost translates to the $\diamondsuit$ operator as depicted in Property 3.13.

$$\forall\pi_1.\forall\pi_2.\ (in_{\pi_1} \leftrightarrow in_{\pi_2}) \rightarrow \diamondsuit^{[0,t]}(encryption_{\pi_1} \leftrightarrow encryption_{\pi_2})$$

Property 3.13: Timed and untimed Finally operator

The following two traces outline the different meaning of the above property when replacing $\diamondsuit^{[0,t]}$ by $\diamondsuit$.

$$\tau_1 = (\{in\},\{in\})(\{\_\},\{\_\})\overbrace{\ldots}^{t-2\text{ times}}((\{encrypt\},t-1),(\{encrypt\},t-1))(\{\_\},\{\_\})\ldots$$
$$\tau_2 = (\{in\},\{in\})(\{\_\},\{\_\})\overbrace{\ldots}^{t\text{ times}}((\{\_\},t+1),(\{\_\},t+1))(\{encrypt\},\{encrypt\})\ldots$$

The timed variant $\diamondsuit^{[0,t]}$ shortens the possible positions where the encryption proposition may appear in the trace.

4 Model Checking

The model checking problem of Timed HyperLTL is strongly related to the one of HyperLTL and MITL. Therefore, we first present the model checking algorithms for both underlying logics in Section 4.1. Based on this we investigate how to check Timed HyperLTL formulas in Section 4.2. Furthermore, we prove the complexity of the model checking problem.

4.1 HyperLTL – MITL

As Timed HyperLTL is built upon HyperLTL and MITL, we first introduce the corresponding model checking algorithms together with a short description of their complexity.

HyperLTL

We will focus on the model checking approach for the alternation free fragment of HyperLTL in this section. Model checking HyperLTL is based on an automata-theoretic approach for LTL. The question to answer is whether for a given Kripke structure K and a HyperLTL formula ψ it holds that $K \vDash \psi$ [11]. It is first assumed, that all quantifiers occurring in the formula ψ are existential quantifiers. If they are universal, $K \vDash \psi$ is equivalent to K not satisfying the negated formula:

$$K \vDash \forall \pi_1 \ldots \forall \pi_n . \psi \text{ iff } K \nvDash \exists \pi_1 \ldots \exists \pi_n . \neg \psi .$$

The model checking of HyperLTL proceeds in three steps:

1. Create an ψ-equivalent Büchi automaton A_ψ with alphabet $(2^{AP})^n$, where each letter of A_ψ is a n-tuple of states of atomic propositions. Position i of such a tuple contains all atomic propositions of trace π_i.

2. Intersect the automaton A_ψ with the K-equivalent automaton and reduce the alphabet by one exponent of 2^{AP} to eliminate one existential quantifier. Repeat this until all existential quantifiers are removed.

3. Emptiness-check. $K \vDash \psi$ iff $L(A_\psi) \neq \emptyset$

The complexity of model checking the alternation free fragment of HyperLTL is essentially the same as for LTL [12]. This is due to the reduction of the HyperLTL model checking to LTL model checking.

Theorem 4.1 ([12]). *The model checking problem for the alternation free fragment of HyperLTL is PSPACE-complete in the size of the formula and NLOGSPACE-complete in the size of the Kripke structure.*

Model checking gets undecidable for the complete fragment of HyperLTL [12].

Theorem 4.2 ([12]). *Model checking the complete fragment of HyperLTL is in general undecidable.*

MITL

The model checking algorithm for MITL has been proposed by Alur et al. [3]. It is based on the satisfiability problem of the language and works as follows: The model checking problem for MITL is to decide whether or not all timed state sequences that are accepted by a given timed automaton A satisfy a given MITL formula ϕ:

$$L(A) \subseteq L(\phi)$$

The authors use the construction for testing the satisfiability of MITL formulas to solve the model checking problem. First, they construct the timed automaton $B_{\neg\phi}$, that accepts precisely the models of the negated formula $\neg\phi$. Then, the model checking problem reduces to the emptiness check of the disjunction of all accepted traces of the timed automaton A and the created automaton $N_{\neg\phi}$. Hence, the model checking problem can be reformulated as follows:

$$L(A) \subseteq L(\phi) \quad \text{iff} \quad L(A) \cap L(B_{\neg\phi}) = \emptyset$$

Thus, the model checking problem for MITL is in general hard [3].

Theorem 4.3 ([3]). *Model checking of MITL formulas is EXPSPACE-complete.*

4.2 Algorithm

As mentioned above, we combine the essential approaches of the model checking algorithms for HyperLTL and MITL to build the algorithm for Timed HyperLTL. A main difference consists in the setting of formula and system. We use a timed automaton A_t as specification for a timed system and handle the timed formula to UPPAAL [4] to check it against the system. Hence, it is not necessary for us to create a ψ-equivalent automaton for the Timed HyperLTL formula ψ. As we have to reduce the explicit trace quantification of Timed HyperLTL to MITL, we do a self-composition on the timed automaton A_t with iteration depth depending on the number of quantifiers. Through the self-composition of the system we obtain a mapping of trace variables to state positions.

1. Given are the system to be checked as a timed automaton A and the Timed HyperLTL formula ϕ with n leading universal quantifiers without alternation. First, do a self composition of the automaton $n-1$ times as described in Definition 2.3. As result, get a new system A' with states consisting of n-tuples of atomic propositions, where one can match position i of the state tuple to the respective trace variable π_i.
2. To check the given formula on the system, convert it to Uppaal syntax and adapt the trace variables to match the self-composed state tuples. This is done by a fast-forward implementation of replacing the trace variables by their equivalent mapped trace instances. Save whether the input formula is an universal quantified property or an existential quantified property, as the universal one has to be converted to a formula with existential quantifier prefix.
3. Check the formula for the resulting automaton by calling the *verifyTA* routine of Uppaal. Distinct two cases:
   - If the input formula is universally quantified, then the formula given to Uppaal is in existential normal form. If the converted formula is satisfied, the input formula is not satisfied.
   - Otherwise, the input formula is existentially quantified and the result of Uppaal model checker translates to the overall result.

The algorithm is shown in Algorithm 1 as pseudocode.

Algorithm 1 Model checking algorithm of Timed HyperLTL

```
Input: $K = (S, s_0, \delta, AP, L)$,
       Timed HyperLTL formula $\phi, n = \#$ trace quantifiers in $\phi$
Output: Yes if $K \vDash \phi$ else No
$A' = Self - composition(K, n)$
$\phi' = Convert(\phi)$
result = Uppaal.verifyTA($A', \phi'$)
if $\phi$ starts with $\forall\pi_1.\forall\pi_2\ldots$ then
    output($\neg$ result)
else
    output(result)
end if
```

The self-composition is a crucial part of the model checking, and we therefore explain it now in more detail. By applying the self-composition once, we obtain a system, where each state s' consists of a pair of states of the initial system: $s' = (s_1, s_2)$. Each of the positions of this pair is again a tuple of $X \subseteq AP$ and the current delay value d. Hereby, we get a projection of the trace variables π_1 and π_2 to the positions in the tuple of s'. After the self-composition is applied, the initial formula has to be adapted with the projection of the trace variables. The trace variables have to be removed from the formula and replaced by the concrete projected sequences of state tuple positions.

This is a slightly different approach as for HyperLTL, where an additional automaton is created from the formula and then composed with the system to eliminate the quantifier. However, the last step is quite the same because the model checking of HyperLTL results in an emptiness check of the language of the resulting automaton. The language emptiness check can be translated into a reachability check for accepting states of the system. For Timed HyperLTL, we have a search for a specific trace in the last step, which does not satisfy our timed hyperproperty.

Note on the usage of UPPAAL

To get a more detailed view on the model checking algorithm implemented in UPPAAL by Larsen et. al [18], we describe the *verifyTA* routine in the following section. The query language for the verifier is a fragment of MITL extended by path quantifier and clock constraints. UPPAAL implements a reachability check, i.e. whether a given combination of clock constraints and data is reachable from the initial configuration of the timed system. This reachability check is built as goal directed search for a diagnostic trace σ. If a diagnostic trace exists for the system A and the formula ϕ, then $\phi \nvDash A$ because $\exists \tau \in A.\ \tau \nvDash \phi$. The search performs by applying several inference rules on the timed system. The rules are shown in detail on p. 7 of [18]. The search has two termination criteria:

1. *Success*: If a diagnostic trace is found, then the search stops by outputting this trace as counterexample.

2. *Fail*: If the backtracking does not find a diagnostic trace, the formula is satisfied on the system: $\phi \vDash A$.

If the call to UPPAAL returns a counterexample, our model checking algorithm currently does not forward this counterexample, as it is not applicable to the input system. This is due to the fact, that the projection from the trace of self-composed system to the initial one is not intuitive.

We already mentioned the usage of two time constraints in one formula, e.g. $\Box^{[0,t]}(a_{\pi_1} \to \Diamond^{[0,1]} a_{\pi_2})$. UPPAAL allows the check of subtractions of two clocks, i.e. $3 \leq x - y \leq 5$. This feature allows a constant translation of the above usage of timing constraints in Timed HyperLTL. We therefore do not need to compute multiple clock constraints to match on one clock. We can just use two clocks in a synchronous manner, i.e. starting at the same time. Then we can carry out the respective operation directly on the two clocks.

Complexity

The complexity of the model checking problem for Timed HyperLTL depends on the fragment of Timed HyperLTL to be checked. The bigger part of information flow properties presented in our work can be described by formulas of the alternation-free fragment of Timed HyperLTL.

As shown by Bouyer et al. [6], the model checking problem for MITL (a relaxed fragment of MTL without punctuality), which is partially used in the verifying language of Uppaal, is EXPSPACE-complete. We investigate in the following, whether this complexity translates to our model checking approach. After applying the self-composition to a timed system, we have a polynomial blow up of the state space of the system. The self-composition always computes the cross-product of the state sets. The polynomial degree depends on the numbers of trace variables of the formula to be checked.

The model checking problem for Timed HyperLTL is at least as expensive as the model checking problem for MITL.

Theorem 4.4 (Model Checking Complexity). *The model checking problem for the alternation free fragment of Timed HyperLTL is EXPSPACE-complete in the size of the formula.*

Proof. We have to prove two things: the model checking problem for Timed HyperLTL is EXPSPACE-hard and it is in the class EXPSPACE.

EXPSPACE: By applying the self-composition on A, the size of A is enlarged polynomially in the number of trace quantifiers. The state set of the self-composed system increases quadratically for two trace quantifier in the formula, because the new state set is the cross-product of the old state set with itself. The transition set grows at most quadratically because of the mapping of old transitions to new transitions.
The check whether a non-satisfying trace is part of the timed automaton can be reduced to the emptiness check for non-satisfying traces in the timed automaton. As shown by Alur et al. [3], the emptiness check for timed automata is PSPACE-complete. The model checking of MITL is performed by building a timed automaton $B_{\neg\phi}$ for the formula ϕ and checking, whether the disjunction of the languages of A and $B_{\neg\phi}$ is empty, i.e.:

$$L(A) \overset{?}{\subseteq} L(B_{\neg\phi})$$

$B_{\neg\phi}$ is built by a construction described by Alur et al. [3]. The authors distinguish between six types of MITL-subformulas of ϕ. These are grouped by the occurrence of the temporal operators $\Box$ and $\Diamond$ and the structure of the interval on the operators. If the intervals are of the form $[0, b]$ then the construction of a timed automata for subformula ϕ' is straightforward and the automaton only needs one clock for checking validity. If the intervals do not start at zero, the construction is more complex as it needs pairs of clocks for checking the subformula in the interval. Overall, the construction is exponential in the number N of temporal operators in ϕ and in K as the largest occurring constant of a clock constraint in ϕ. More detailed, the complexity is in $\mathcal{O}(2^{N \cdot K \cdot log(N \cdot K)})$. Hence, checking emptiness for a non-satisfying trace in the self-composed system is in EXPSPACE. Thus, the self-composition does not enlarge the complexity of the model checking problem. In sum, model checking of Timed HyperLTL formulas is in EXPSPACE.

EXPSPACE-hard: As we saw before, MITL and HyperLTL are both subsumed by Timed HyperLTL. The model checking problem of MITL is EXPSPACE-complete [6], whereas the model checking problem of HyperLTL (for the alternation free fragment) is PSPACE-complete [12]. To prove EXPSPACE hardness, we give a reduction of model checking MITL to the problem of model checking Timed HyperLTL. Given an arbitrary MITL formula ϕ. Then ϕ is satisfied in a timed automaton A, i.e. $A \vDash \phi$, iff all timed state sequences, that are accepted by a given timed automaton A, satisfy ϕ:

$$\forall \tau \in L(A) : \ \tau \vDash \phi,$$

where τ is a timed state sequence of the form $(\overline{s}, \overline{I})$. We have to built a Timed HyperLTL ϕ' with

$$A \vDash \phi' \leftrightarrow A \vDash \phi$$

This is done by the following construction. Take all timed state sequences of A, that satisfy ϕ. We add the universal quantifier, a trace variable and update the formula to index all atomic propositions with the trace variable:

$$\phi' \equiv \forall \pi. \ \phi[a \mapsto a_\pi]$$

ϕ' is a Timed HyperLTL formula for which we can apply our model checking algorithm. Hence, the model checking problem of MITL can be reduced to the one of Timed HyperLTL, which proves, that model checking Timed HyperLTL is EXPSPACE-hard.

■

5 Experimental Results

Chapter 5 outlines some case studies of the properties described before and the experimental results of running our model checker for these formulas on suitable systems. We will present several models for password checking systems. Additionally, we depict a model of the SSL handshake protocol and give some safety properties a system should satisfy to prevent a timing attack on the OpenSSL implementation, introduced by Boneh and Brumley [7]. With these experiments, we want to show some benchmarks of our prototype model checker and hence, the possible applications of model checking timed hyperproperties.

5.1 Password checker

Simple Password Checker

We model a simple password checking system that should not reveal any secure information to an eavesdropper if the entered password is incorrect in a time interval $[0, t]$. The model is depicted in Figure 5.1.1. It simply reads the complete input given by the user and then compares it to the saved password. The check whether input and password coincide is finished only after all bits have been compared. This ensures the checking routine to only depend on the length of the input and not on the quality of its correctness. For simplicity we set the time constraint to 2. This depends on the real implementation of the system and therefore, may be adapted.

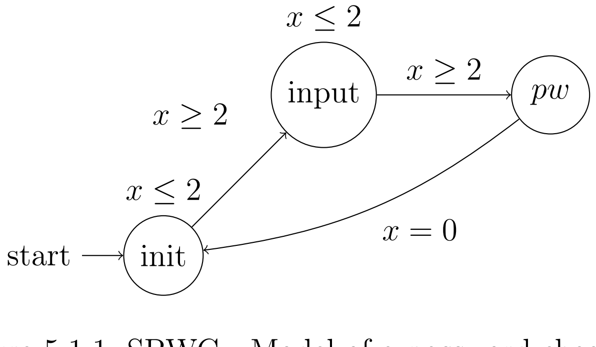

Figure 5.1.1: SPWC - Model of a password checker

The system shown in Figure 5.1.1 is checked against the formula with our prototype model checker. The property below states, that the system has to show the same behaviour for the validation of the input if the input is the same on two traces. The property is an adapted variant of the timed observational determinism property shown in Property 3.6.

$$\forall \pi_1, \pi_2. \Box^{[0,2]}(in_{\pi_1} \leftrightarrow in_{\pi_2}) \rightarrow (pw_{\pi_1} \leftrightarrow pw_{\pi_2})$$

As contrast to the secure system modelled in Figure 5.1.1, we additionally modelled an insecure password checker, that also reads in the complete input, but has a different runtime when checking correct or incorrect input. The formulas used in the check are shown in Table 5.1.1.

SPC1	$\forall \pi_1, \pi_2 \; \Box^{[0,t]} \, pw_{\pi_1} \leftrightarrow pw_{\pi_2}$
SPC2	$\exists \pi_1, \pi_2. \; \Diamond^{[0,t]} \, \neg(pw_{\pi_1} \leftrightarrow pw_{\pi_2})$
SPC3	$\forall \pi, \pi'. \Diamond^{[0,4]} \, \neg(pw_{\pi} \leftrightarrow pw_{\pi'})$
SPC4	$\forall \pi_1, \pi_2. \Box^{[0,2]}(in_{\pi_1} \leftrightarrow in_{\pi_2}) \rightarrow (pw_{\pi_1} \leftrightarrow pw_{\pi_2})$

Table 5.1.1: Results for model checking the password checker example

With SPC1, we postulate that the system must always behave deterministically in the validation of the password. We state Formula SPC2 as a possible vulnerability of a password checker. It is satisfied by a system with at least one trace pair where the passwords are once not validated in the same time interval. If a system A satisfies SPC3, then it automatically satisfies SPC2, as SPC3 is satisfied, if all runs of a system do not coincide on the validation of the password. One could think

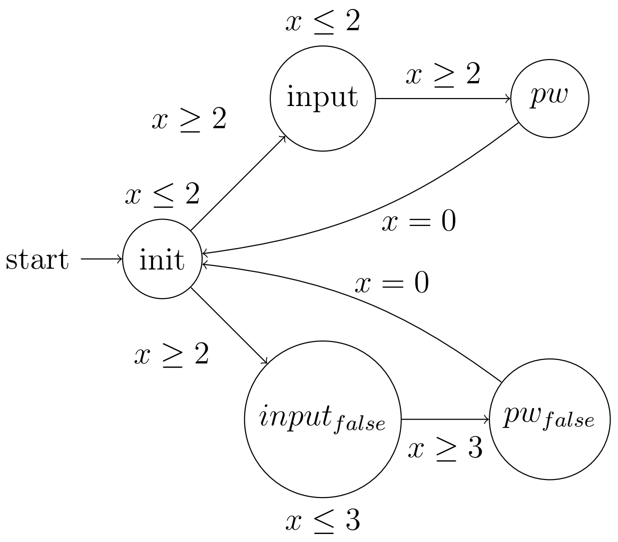

Figure 5.1.2: SPWC$_{nc}$ Model of a vulnerable system

of the system A_{SPC3} to be more unsafe than A_{SPC2}, because all runs of the system are bad, whereas A_{SPC2} could have less possibilities for unsafe executions.

Bitwise Password Checker

Based on System 5.1.1 we modelled a bitwise password checker that compares the input bit for bit with the password. When the first position is reached, where input bits and secret bits differ, the system goes into the *incorrect* state. This variant differs from the previous system. It is possible to extract information from the output of the system if the implementation is insecure with respect to the output behaviour. An eavesdropper can easily extract the wrong and correct bits by the computation time and the dijkstra-distance of the different inputs. This is possible, as the system can stop the input parsing at any time to go into the *false* state, whereas it can only reach the *correct* state after t time steps. Thus, by relating the input to the time needed for validation, the attacker can compute which prefix of the input was correct and which position of the input is the first with another bit compared to the secret.

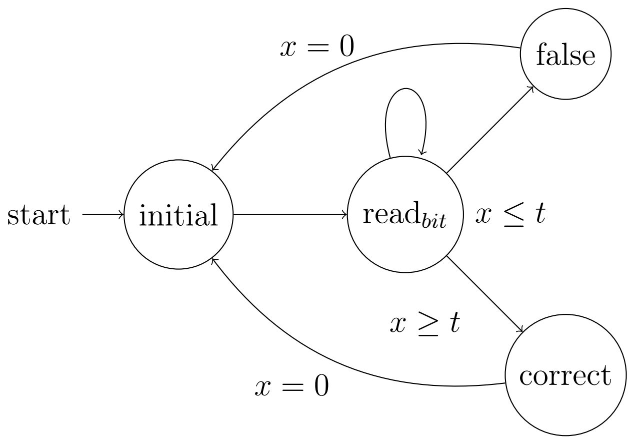

Figure 5.1.3: Bitwise password checker for t bit input length

The properties we checked build upon the formula template of Property 5.1. The main difference is the scaling in the input length. The larger the size of the password, the larger is the runtime of the password checker before outputting the result.

$$\forall \pi_1, \pi_2 . (\Box^{[0,t]}\, in_{\pi_1} \leftrightarrow in_{\pi_2}) \rightarrow (\Box^{[t,\infty]}\, correct\pi_1 \leftrightarrow correct_{\pi_2})$$

Property 5.1: BWt - Hyperproperty for the bitwise password checker model

To prevent a timing attack on the bitwise password checker, the system's runtime needs to be independent of the correctness quality of the input. That's why we force the model to satisfy the *correct* state on all traces for the time steps $t, t{+}1, \ldots$ if the input is equal beforehand.

The following property distinguishes the safe password checker of Figure 5.1.1 and the unsafe bitwise password checker of Figure 5.1.3.

$$\forall \pi_1, \pi_2.\ \Diamond(in_{\pi_1} \wedge \neg in_{\pi_2}) \rightarrow \Diamond^{[0,t]}(pw_{\pi_1} \wedge \neg pw_{\pi_2})$$

The above formula is satisfied by Figure 5.1.1, because the validation of the password solely depends on the input. If the input on trace π_2 differs at one position to the input of π_1, then the traces cannot match on all positions on the password checking. The property is not satisfied by the system of Figure 5.1.3. There are two traces π_1 and π_2 where the inputs have different positions for the first difference in bits to the password. Hence, the system of Figure 5.1.3 is insecure and vulnerable against a timing attack.

Experimental Results

We checked different scales of the input bit lengths and give an overview of the results in Table 5.1.2. It contains the number of states and transitions of the system after the self-composition has been applied, as well as the time needed for model checking the property. It is worth to note that almost all runs of the model checker were finished in less then 1 second on this low-size models. We will later on present similar experiments on a system with larger number of states and transitions. In this context, we will discuss the enlarging of time needed by our prototype model checker for validating properties on the respective systems.

The runtime for model checking depends on whether the property is satisfied by the system. Intuitively, this makes sense with respect to the model checking algorithm of Uppaal. It is implemented as a search for a counterexample trace for the property. So the algorithm searches through the system to find a violation state sequence. If the property is not satisfied, the algorithm will find one before having checked all sequences. If the property is satisfied, the algorithm won't find one trace and thus, has to check all state sequences.

The experiments were carried out on a medium-class machine running Windows 10 on an Intel Core i5-4200U CPU with 2.3 GHZ and 8 GB RAM.

We provide other models to demonstrate the correlation between the structure of the models, the constants in the clock guards and the verification runtime. Both again describe a bitwise password checker, however with differences to the model of Figure 5.1.3. The first model is a system with a branching structure for every bit check, i.e. the system takes other transitions if the bit at position i of the input differs from the password. However, the model does not fail directly. Instead it checks the next position and branches again if the bit at position $i + 1$ differs. This model is a secure variant of a bitwise password checker. We set the input

	Model	# States	# Transitions	Runtime in ms	
SPC1	SPWC	9	15	307	✓
SPC2				254	✗
SPC3				230	✗
SPC4				273	✓
SPC1	SPWC$_{nc}$	25	48	176	✗
SPC2				256	✓
SPC3				296	✓
SPC4				291	✗
BW6	BPWC	16	20	391	✓
BW8				493	✓
SPC1	BWPC	16	20	491	✓
SPC2				254	✗
SPC3				230	✗
SPC4				273	✓

Table 5.1.2: Overview of the examples

bit length to 8 bits for this example. With 8 bits, we can handle 255 different passwords, if the domain is limited to integers. The basic structure of the model is depicted in Figure 5.1.4. The index of the *read* propositions represent, which positions of the input are different to the bit positions of the password.

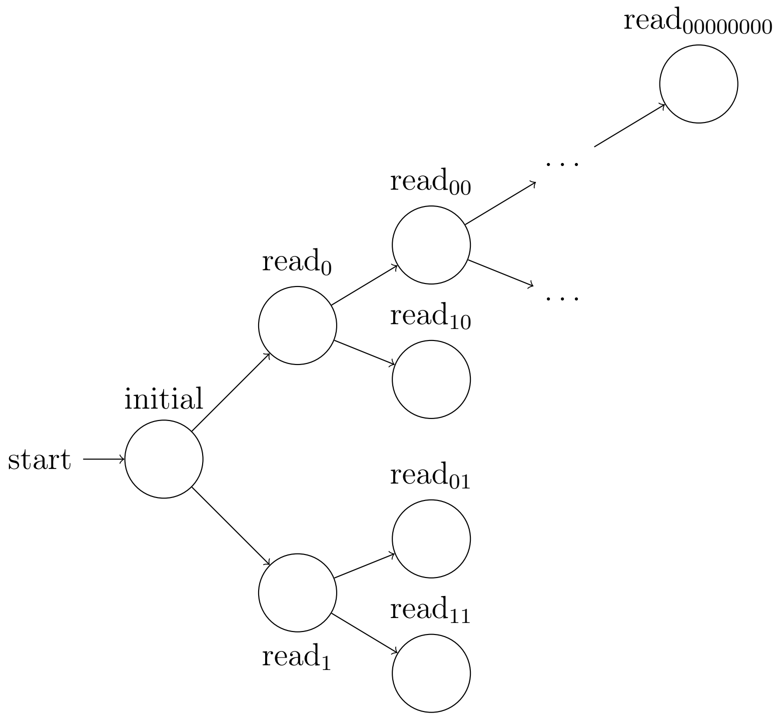

Figure 5.1.4: Bitwise password checker – Branching

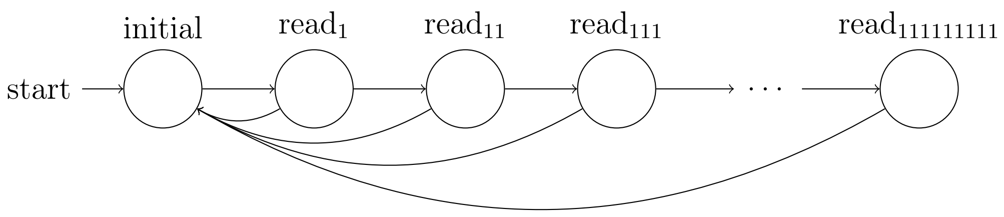

Figure 5.1.5: Bitwise password checker – Linear

The second model, depicted in Figure 5.1.5, is another variant of bitwise password checking. It is built linear where the input validation can fail at every position of the model, in contrast to the branching checker. The validation shows that the runtime of the second model depends on the first position where input and password differ.

We check the same properties on the two models as for the simple password checker, namely SPC1 to SPC4. The results presented in Table 5.1.3 show the large

	Model	# States	# Transitions	Runtime in ms	
SPC1	Branching	3969	3968	18145	✓
SPC2				15354	✗
SPC3				16157	✗
SPC4				19396	✓
SPC1	Linear	81	256	408	✓
SPC2				387	✗
SPC3				273	✗
SPC4				198	✓

Table 5.1.3: Overview of the examples

increase of the validation time for the branching bitwise password checker. The initial system already has 63 states and 62 transitions, which leads by quadratic blow-up of the self-composition to more than 3.900 states and transitions. In contrast to this, the linear bitwise password checker only has 9 states and still 16 transitions, as every state can directly go back to the initial state on a failure.

The results underline the dependency of the validation time on the structure of the model. We have seen two possibilities for modelling a bitwise password checker, where both are realizable. The two presented models serve as a synthetic example for the scalability of our prototype model checker and the efficiency of the developed model checking algorithm.

5.2 SSL

We present a model of the SSL handshake protocol in this section as developed in [10] and [25]. We will use this to simulate the timing attack of Brumley and Boneh on SSL [7]. First, we define the RSA factorization problem to understand, how the gap of OpenSSL is used by the authors to break the system. Then, we run our model checker with properties defining safety against the timing attack on two systems, one modelling a secure system, one modelling an insecure system.

RSA

RSA (Rivest-Shamir-Adleman) is a public-key cryptosystem for secure data broadcasting and digital signatures [24]. It was invented by Ron Rivest, Adi Shamir and Leonard Adleman. In the RSA cryptosystem, Alice first computes a public key that is built from two large prime numbers, p and q. The prime numbers themselves may not be revealed as they are the secrets. The public key is the product of the prime numbers $n = p \cdot q$ and the exponent e computed by $e \cdot d \equiv 1 \mod \phi(n)$ with $\phi(.)$ as Euler's totient function. (N, e) is Alice's public key and (N, d) is Alice's private key. (N, e) may be revealed and is used to *encrypt* a message m, while (N, d) is used to *decrypt* a cipher text c: $c = m^e \mod N$ and $m' = c^d \mod N$.

Timing attack

The timing attack presented by Boneh and Brumley is based on the specific implementation of the RSA modulus factorization in OpenSSL. OpenSSL uses the Chinese remainder Theorem (CRT) to compute the exponent d in $m = c^d$:

$$\begin{aligned} m_1 &= c^{d_1} \mod p \text{ and} \\ m_2 &= c^{d_2} \mod q \\ \implies m &= (q^{-1}(m_1 - m_2) \mod p) \cdot q + m_2 \end{aligned}$$

An attacker client starts a client key exchange with a guessed value for one RSA factor, q. The guess will be iteratively improved by approximating the decryption time of the guess g and g_{hi}, where bit i of g is set to 1. Two cases for g_{hi} have to be distinguished: $g < g_{hi} < q$ and $g < q < g_{hi}$. Case 1 implies that bit i of q is indeed 1, otherwise it is 0. With this, they compute $u_g R = gR^{-1} modN$ and $u_{g_{hi}} = g_{hi}R^{-1}$. These two values are send to the server, such that the decryption of both is started. $t_1 = DecryptTime(u_g)$ and $t_2 = DecryptTime(u_{g_{hi}})$. If $\Delta = |t_1 - t_2|$ is big, then $g < q < g_{hi}$ has to hold and $q_i = 0$, otherwise if Δ is small, then $g < g_{hi} < q$ and $q_i = 1$. The attacking client iteratively repeats the process of sending the guesses and measuring the decryption time.

The proposed attack is working for a software-only implementation of SSL, since hardware-based solutions have additional defences against timing attacks. They did some experiments to explore the bounds on the number of queries to extract the factor N in the RSA setting. As a result, they obtained 1.433.600 queries to recover a single bit of the factor.

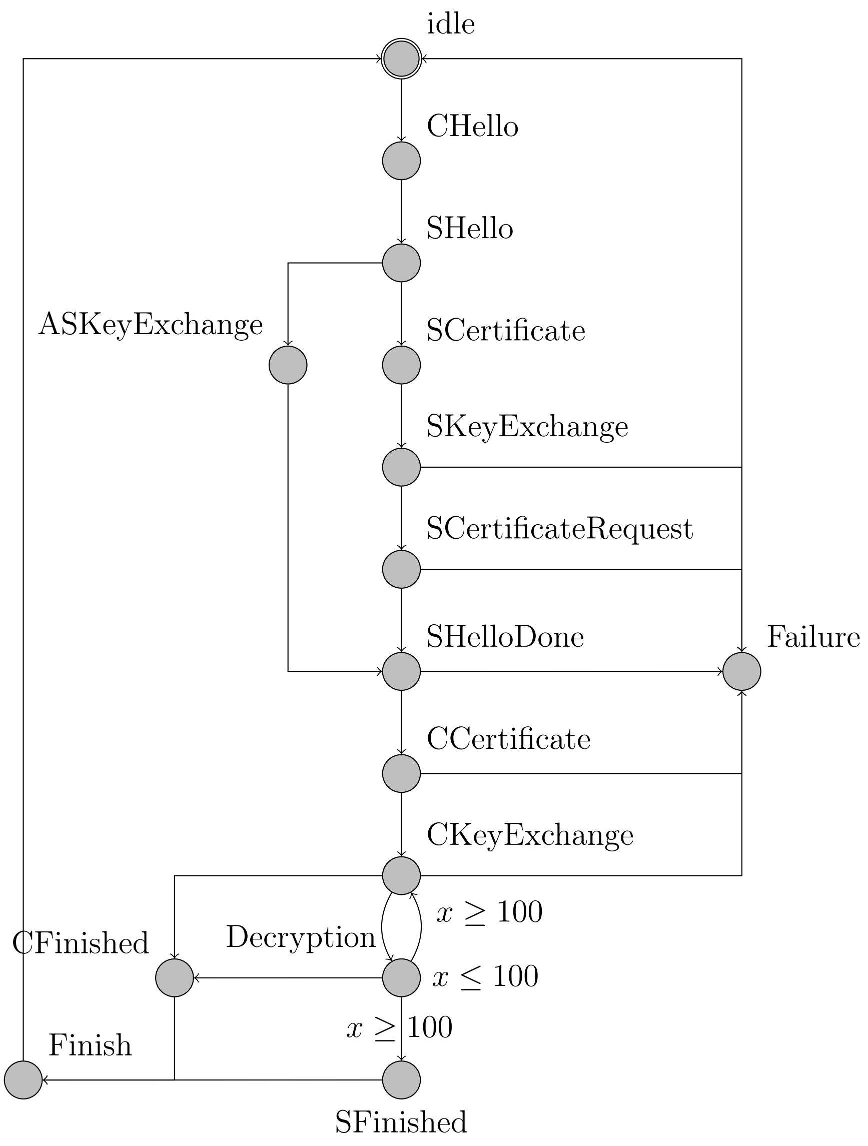

Figure 5.2.1: SSL$_{sec}$ – UPPAAL model of the SSL Handshake protocol [10], secure against the timing attack of [7]

We provide a timed system to model the SSL handshake between server (S) and client (C), that is secure against the timing attack described above. Then we validate a Timed HyperLTL property on this system that checks the vulnerability of against the timing attack. Additionally, we give an insecure system, that is vulnerable against the attack and check the Timed HyperLTL properties on this system, too. The secure model to be checked can be seen in Figure 5.2.1.

Property 5.2 states that there may not be a delay in the client key exchange state on two distinct traces and the handshake protocol may not fail after a handshake. This is a very restrictive property to prevent the SSL timing attack [7], as the attack relays on the different runtime of the decryption algorithm and the following failure message. Through the restriction of part (2) of the formula, the system is

$$\begin{aligned}\forall \pi_1, \pi_2.\, \square^{[0,\infty]} &(CKeyExchange_{\pi_1} \leftrightarrow CKeyExchange_{\pi_2}) \\ &\wedge (CKeyExchange_{\pi_1} \rightarrow \underbrace{(\square^{[0,0]} \neg failure_{\pi_1} \wedge ServerFinish_{\pi_1}))}_{(2)} \\ &\wedge (CKeyExchange_{\pi_1} \rightarrow \underbrace{(\square^{[0,\infty]} decryption_{\pi_1} \leftrightarrow decryption_{\pi_2}))}_{(3)}\end{aligned}$$

Property 5.2: Safety property to prevent SSL timing attack

forced to have no delay to go to the finish state after the *ClientKeyExchange* is done. If the system would have different runtime in the decryption method, the property would not be satisfied and the system may be vulnerable.

We additionally stipulate (3), such that the decryption algorithm runs in a synchronous way on all traces after the *ClientKeyExchange* has been done, which prohibits different runtime of the decryption method for different inputs. The different possible inputs are the guesses computed iteratively in the timing attack. To prevent the dependency of runtime from inputs we use implicit timed noninterference for the decryption algorithm, which is expressed by the above property fragment.

The model for the insecure model is depicted in Figure 5.2.2.

The second timed hyperproperty shown in Property 5.3 describes the behaviour of the system to have a delay of at least t time steps between the first decryption query to be finished and the second one. This is a kind of *bad* security property as it expresses the vulnerability of the system for the OpenSSL timing attack. If a *Decrypt* query may be finished on one run of the system t time steps later than another run, then the system's decryption method may violate the noninterference hyperproperty.

$$\forall \pi_1, \pi_2.\, CKeyExchange_{\pi_1} \wedge (Decrypt_{\pi_1} \rightarrow \Diamond^{[0,t]} \neg Decrypt_{\pi_2}) \wedge \Diamond^{[t,\infty]} Decrypt_{\pi_2}$$

Property 5.3: Property of a system being vulnerable for SSL timing attack

If we investigate the second property and the last part of property 5.2, we see that

$$\begin{aligned}\forall \pi_1, \pi_2.\, (&CKeyExchange_{\pi_1} \wedge (Decrypt_{\pi_1} \rightarrow \Diamond^{[0,t]} \neg Decrypt_{\pi_2}) \wedge \Diamond^{[t,\infty]} Decrypt_{\pi_2} \\ &\wedge (CKeyExchange_{\pi_1} \rightarrow (\square^{[0,\infty]} Decrypt_{\pi_1} \leftrightarrow Decrypt_{\pi_2}))\end{aligned}$$

cannot be satisfied by the same system if $t > 0$, since

$$\forall \pi_1, \pi_2.\, (Decrypt_{\pi_1} \rightarrow \Diamond^{[0,t]} \neg Decrypt_{\pi_2}) \wedge \square^{[0,\infty]}(Decrypt_{\pi_1} \leftrightarrow Decrypt_{\pi_2}) \equiv \bot$$

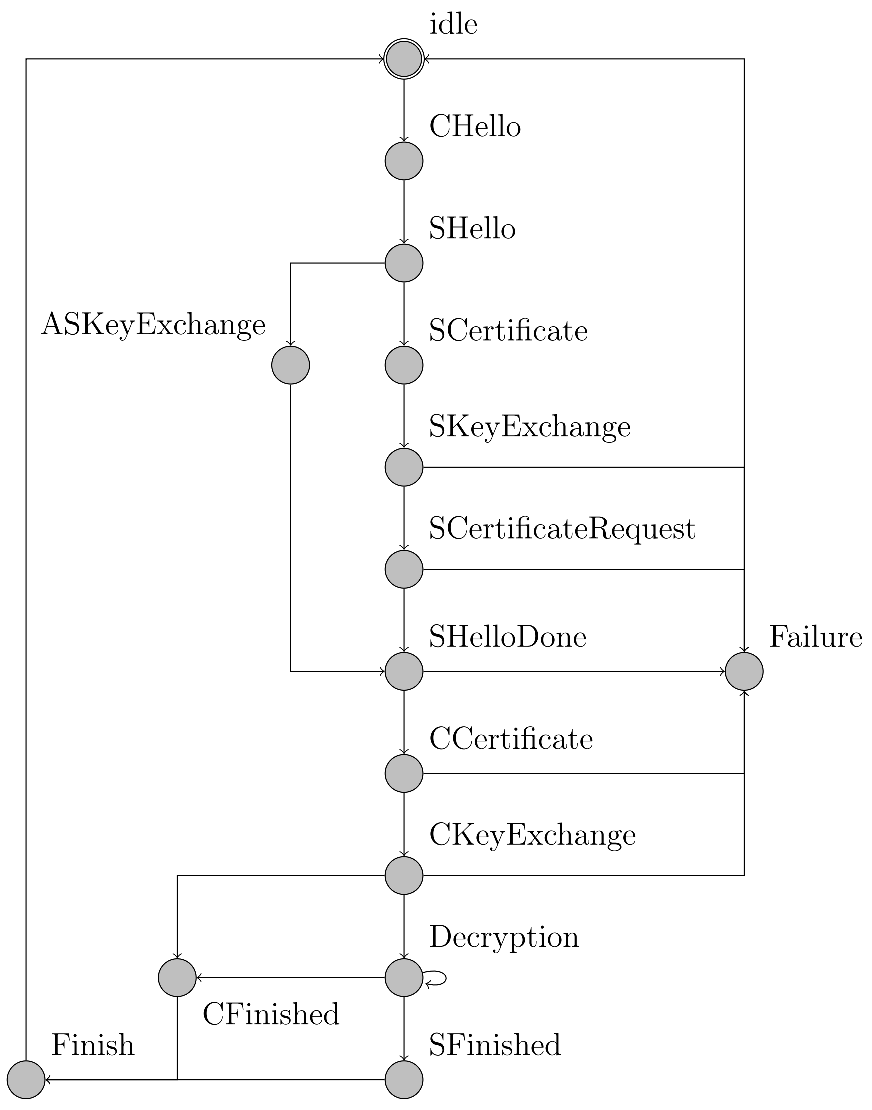

Figure 5.2.2: SSL$_{insec}$ – Uppaal model of the SSL Handshake protocol [10], vulnerable for the timing attack of [7]

This holds because of the definitions of $\Diamond^{[0,t]}$ and $\Box^{[0,\infty]}$. Note that the two temporal operators are in different scopes and thus the time intervals are independent.

The first system, which is a model with an implemented defence, ensures that all decryption queries take at least as long as the longest decryption run. This is forced by the guard $x \geq 100$ on the transition and the state inference $x \leq 100$ where we assume the longest query to take 100ms. Thus all decryption queries take exactly 100ms. Although this slows down the complete system, we have a higher security in the decryption section. The optimization of this trade-off between efficiency and security is non-trivial, as it always has to take specific conditions of the single systems into account.

Better defences than artificially enlarging the decryption runtime have been examined and published. One possible fix is so called RSA blinding, where a random

Nr	Model	# States	# Transitions	Runtime	Space	Satisfied?
Prop1	SSL$_{sec}$	225	243	1080ms	20 MB	✓
Prop2				734ms	20 MB	✗
Prop1	SSL$_{insec}$	225	235	449ms	16 MB	✗
Prop2				861 ms	21 MB	✓

Table 5.2.4: Results of model checking the SSL systems against the properties

number is exponentiated r^e, instead of computing g^e. With this, it is harder to compute the secret part g as r is random for each decryption query. The artificial change of the decryption runtime is bad for the overall runtime of the OpenSSL implementation.

Table 5.2.4 shows the results of checking the two properties against the secure and insecure models of the SSL Handshake Protocol. It contains the runtime and space needed for the verification as well as the numbers of states and transitions of the self-composed systems.

The experiments show that our approach is able to verify timed hyperproperties for medium and large scaled models in efficient time. The runtime for validation increases significantly by the number of states and transitions of the self-composed systems compared to the models of the password checkers. However, models with more than thousands states after self composition could still be checked in reasonable time. The required space has been kept within limits of up to 100 MB. There is still potential for optimization, for example by handling the returned diagnostic trace of Uppaal in case the property is not satisfied on the system. It has to be reworked, such that it matches the traces of the original system.

6 Conclusion & Future work

We introduced a new temporal logic, Timed HyperLTL, for the verification of temporal hyperproperties. These properties are used to establish requirements of a system to be secure against timing attacks. For the construction of Timed HyperLTL we combined the two logics HyperLTL and MITL to obtain a logic, which is able to express timing restrictions on multiple execution traces of a system.

After that, we extended several classical hyperproperties to timed hyperproperties. Additionally, we modelled well-known timing attacks on encryption schemes and cryptographic systems. We proposed some possible defences by stating requirements on the systems as Timed HyperLTL formulas. We outlined the model checking algorithm of Timed HyperLTL as a combination of the respective algorithms of HyperLTL and MITL. The resulting model checking problem has been proven to be EXPSPACE-complete.Then, we developed a prototype model checker based on the verifying module of Uppaal. Using the prototype model checker, we evaluated some timed hyperproperties on several models, e.g. models of password checkers, and checked properties against timing attacks on a model of the SSL handshake protocol. The experiments show that the usage of Uppaal as underlying verification tool for timed systems is efficient in time.

Future work contains the solution to the asynchronous setting of stuttering traces. While the synchronous case considered in this thesis deals with only one clock for all traces, the asynchronous case can be defined by introducing clocks for each individual trace. Therefore, two traces τ, τ' starting at the same time point, can have different delay transitions. When one of both traces stutters, we allow $\exists z \in \mathbb{R} : \Pi \models_{\tau+z,\tau'} \psi$. Then, we cannot match the respective positions of the traces as defined before.

Also, the complexity of the satisfiability problem in general, the model checking problem for the $\exists^*\forall^*$ fragment and the complete language are points of interest. This involves the question for a bounded satisfiability and model checking problem. In context of hyperproperties it would be interesting whether the synthesizing approach of reactive systems from hyperproperties can intuitively be extended to properties formulated in Timed HyperLTL.

References

[1] Bowen Alpern and Fred B. Schneider. “Defining Liveness”. In: *Inf. Process. Lett.* 21 (1985), pp. 181–185.

[2] Rajeev Alur, Pavol Černý, and Steve Zdancewic. “Preserving secrecy under refinement”. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2006, pp. 107–118.

[3] Rajeev Alur, Tomás Feder, and Thomas A Henzinger. “The benefits of relaxing punctuality”. In: *Journal of the ACM (JACM)* 43.1 (1996), pp. 116–146.

[4] Johan Bengtsson et al. “UPPAAL - a tool suite for automatic verification of real-time systems”. In: *International Hybrid Systems Workshop*. Springer. 1995, pp. 232–243.

[5] Daniel J. Bernstein. *Cache-timing attacks on AES*. 2005.

[6] Patricia Bouyer et al. *Model Checking Real-Time Systems*. 2016.

[7] David Brumley and Dan Boneh. “Remote Timing Attacks Are Practical”. In: *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*. SSYM’03. Washington, DC: USENIX Association, 2003, pp. 1–1.

[8] Michael R Clarkson and Fred B Schneider. “Hyperproperties”. In: *Journal of Computer Security* 18.6 (2010), pp. 1157–1210.

[9] Michael R. Clarkson et al. “Temporal Logics for Hyperproperties”. In: *POST*. 2014, pp. 265–284.

[10] Gregorio Dıaz et al. *Automatic Verification of the TLS HandShake Protocol.* 2004.

[11] Bernd Finkbeiner. “Temporal Hyperproperties”. In: *Bulletin of EATCS* 3.123 (2017).

[12] Bernd Finkbeiner and Christopher Hahn. “Deciding Hyperproperties”. In: *CONCUR*. 2016.

[13] Bernd Finkbeiner, Markus N Rabe, and Cesar Sanchez. “Algorithms for model checking HyperLTL and HyperCTL*”. In: *International Conference on Computer Aided Verification*. Springer. 2015, pp. 30–48.

[14] Riccardo Focardi, Roberto Gorrieri, and Fabio Martinelli. “Real-time information flow analysis”. In: *IEEE Journal on Selected Areas in Communications* 21.1 (2003), pp. 20–35.

[15] Paul C Kocher. “Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems”. In: *Annual International Cryptology Conference*. Springer. 1996, pp. 104–113.

[16] Boris Köpf and David Basin. "Timing-sensitive information flow analysis for synchronous systems". In: *European Symposium on Research in Computer Security*. Springer. 2006, pp. 243–262.

[17] Saul Kripke. "Semantical Considerations Of The Modal Logic". In: *Studia Philosophica* 1 (2007).

[18] Kim G Larsen, Paul Pettersson, and Wang Yi. "Diagnostic model-checking for real-time systems". In: *International Hybrid Systems Workshop*. Springer. 1995, pp. 575–586.

[19] David E Muller, Ahmed Saoudi, and Paul E Schupp. "Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time". In: *Logic in Computer Science, 1988. LICS'88., Proceedings of the Third Annual Symposium on*. IEEE. 1988, pp. 422–427.

[20] Luan Viet Nguyen et al. "Hyperproperties of real-valued signals". In: *MEMOCODE*. 2017.

[21] Flemming Nielson, Hanne Riis Nielson, and Panagiotis Vasilikos. "Information flow for timed automata". In: *Models, Algorithms, Logics and Tools*. Springer, 2017, pp. 3–21.

[22] Joël Ouaknine and James Worrell. "On metric temporal logic and faulty Turing machines". In: *International Conference on Foundations of Software Science and Computation Structures*. Springer. 2006, pp. 217–230.

[23] A. Pnueli. "The temporal logic of programs". In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 1977, pp. 46–57.

[24] R. L. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-key Cryptosystems". In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126.

[25] Werner Schindler. "A timing attack against RSA with the chinese remainder theorem". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2000, pp. 109–124.

[26] Moshe Y Vardi. "Alternating automata and program verification". In: *Computer Science Today*. Springer, 1995, pp. 471–485.

[27] Sergio Yovine. "Model checking timed automata". In: *School organized by the European Educational Forum*. Springer. 1996, pp. 114–152.