Mapping Synthesis for Hyperproperties

Tzu-Han Hsu\textsuperscript{1}, Borzoo Bonakdarpour\textsuperscript{1}, Eunsuk Kang\textsuperscript{2}, and Stavros Tripakis\textsuperscript{3}

\textsuperscript{1}Department of Computer Science and Engineering, Michigan State University, USA\{tzuhan,borzoo\}@msu.edu
\textsuperscript{2}School of Computer Science, Carnegie Mellon University, USA eskang@cmu.edu
\textsuperscript{3}Khoury College of Computer Sciences, Northeastern University, USA stavros@northeastern.edu

Abstract—In system design, high-level system models typically need to be mapped to an execution platform (e.g., hardware, environment, compiler, etc.). The platform may naturally strengthen some constraints or weaken some others, but it is expected that the low-level implementation on the platform should preserve all the functional and extra-functional properties of the model, including the ones for information-flow security. It is, however, well known that simple notions of refinement do not preserve information-flow security properties.

In this paper, we propose a novel automated mapping synthesis approach that preserves hyperproperties expressed in the temporal logic HyperLTL. The significance of our technique is that it can handle formulas with quantifier alternations, which is typically the source of difficulty in refinement for information-flow security policies. We reduce the mapping synthesis problem to HyperLTL model checking and leverage recent efforts in bounded model checking for hyperproperties. We demonstrate how mapping synthesis can be used in various applications, including enforcing non-interference and automating secrecy-preserving refinement mapping. We also evaluate our approach using the battleship game and password validation use cases.

Keywords—Information-flow security, Hyperproperties, Synthesis, Refinement

I. INTRODUCTION

System development often involves relating program artifacts at different levels of abstraction. For instance, one may begin by designing a high-level model of the system and later refine it into a more detailed implementation. This step may involve imposing additional constraints or concretizing abstract constructs (e.g., actions) of the system into more detailed counterparts. For example, a simple assignment, \( \texttt{a := b + c} \), in a high-level programming language is typically mapped to two steps during translation to an intermediate language: (1) \( \texttt{reg := b + c} \), and (2) \( \texttt{a := reg} \), where \( \texttt{reg} \) is a register in the target hardware. It is understood that conducting such a step has to result in an artifact that preserves desired properties of the original model. Typically, this preservation achieved by establishing a notion of refinement; that is, any behavior of the implementation is also allowed by the abstract model. However, it is well known that in the context of information-flow security, simple notions of refinement of an abstract model that satisfies a security property may result in an implementation that violates the property, also known as the refinement paradox \cite{refinement_paradox, refinement_paradox_2}.

In a prior work \cite{mapping_synthesis_1}, a notion of mappings was introduced as a mechanism for relating the elements (e.g., actions) of a pair of independently developed system models (e.g., a high-level design and a target implementation platform). For example, when building an application, an abstract procedure to “check a user input against a password” may be implemented by using a string matching algorithm that is available as part of a library; a mapping would then relate this abstract procedure to a specific instantiation of the matching algorithm. But a platform may exhibit its own complex behavior, including subtle interactions with the environment that may be difficult to anticipate and reason about (e.g., the string matching algorithm may leak the number of characters in the password). This makes the task of weaving the behaviors of the two models non-trivial; to alleviate this process, \cite{mapping_synthesis_1} also proposed a technique for automatically synthesizing a mapping that establishes a trace-based property \cite{trace_based_properties} in the resulting weaving of the models. However, trace properties are not expressive enough to encode a wide range of security properties \cite{trace_properties}, including non-interference, and thus the approach of \cite{mapping_synthesis_1} falls short of being able to synthesize mappings to satisfy such properties.

In this paper, we study the problem of mapping synthesis in the context of hyperproperties \cite{hyperproperties} expressed in the temporal logic HyperLTL. Let us first explain the problem by using the following high-level toy example. Consider the transition system (Kripke structure) \( K_A \) shown in Fig. 1a and the following HyperLTL formula describing a requirement for \( K_A \):

\[
\varphi = \forall \pi. \exists \pi'. (p \pi \rightarrow \neg p_{\pi'})
\]

This formula stipulates that for any trace \( \pi \), there exists a trace \( \pi' \), such that it is always the case that if proposition \( p \) holds in some position of \( \pi \), then \( p \) should not hold in the same position of trace \( \pi' \). Observe that \( K_A \models \varphi \), because for any arbitrary trace \( \pi \) in \( K_A \) (e.g., trace 1 = \( \{\{p\}\}\)\(\omega \) produced by path \( s_0, s_1, s_2, s_3 \)), there exists another trace \( \pi' \), such that when \( p \) is true in \( \pi \), \( p \) is false in \( \pi' \) (e.g., trace 2 = \( \{\{q\}\}\)\(\omega \) produced by path \( s_0, s_3, s_2, s_3 \)).

Now, suppose that \( K_A \) models a high-level design, and our goal is to implement \( K_A \) on platform \( K_B \) shown in Fig. 1b. In our approach, this task can be formulated as synthesizing a mapping function \( m \), such that \( (K_A \parallel m K_B) \models \varphi \), where \( \parallel m \) denotes the parallel composition operator under mapping \( m \),

This work was funded in part by the NSF SaTC Award 2100989, Title: SaTC: CORE: Small: Techniques for Software Model Checking of Hyperproperties and NSF SaTC award CNS-1801546 Title: SaTC: CORE: Medium: Collaborative: Bridging the Gap between Protocol Design and Implementation through Automated Mapping.
also called mapping composition [38]. This means that when we compose $K_A$ with $K_B$, we have to somehow map the propositions of $K_A$ to the propositions of $K_B$, in such a way that their mapping composition satisfies $\varphi$. An intuitive meaning behind mapping composition is that whenever proposition $a$ holds in $K_A$, $m(a)$ must also hold in $K_B$ in order for the two machines to move synchronously in their composition. Otherwise, the composed state does not exist in $K_A \parallel m \cdot K_B$. For instance, if we apply the following mapping function:

$$m(p) = r, \quad m(q) = s$$

we obtain the composed Kripke structure $K_A \parallel m \cdot K_B$ shown in Fig. 1c. We can see that traces $t_1$ and $t_2$ in $K_A$ mentioned above are now manifested as traces $t_1' = \{ \{p, r\}, \{q, s\}\} \omega$ and $t_2' = \{ \{q, s\}, \{q, s\}\} \omega$ in $K_A \parallel m \cdot K_B$, which allows $\varphi$ to be preserved under mapping $m$. Thus, $m$ is considered to be a valid mapping. Now, consider another mapping function $m'$, where:

$$m'(p) = s, \quad m'(q) = r$$

Mapping $m'$ results in the composed Kripke structure $K_A \parallel m' \cdot K_B$ shown in Fig. 1d. Here, trace $t_1$ in $K_A$ is manifested in the composed structure as $t_1'' = \{ \{p, r\}, \{p, s\}\} \omega$, for which there exists no trace that satisfies formula $\varphi$. Thus, we identify mapping $m'$ as an invalid mapping that breaks the satisfaction of the desired HyperLTL formula. The mapping synthesis problem is to find a valid mapping, if one exists.

In this paper, we solve the mapping synthesis problem for hyperproperties through a reduction to the bounded model checking (BMC) problem for HyperLTL [35]. Roughly speaking, given a model $K$ and a HyperLTL formula $\varphi$, the goal of BMC is to search for a set of traces bounded by some length $k \geq 0$ in $K \models \varphi$. Our reduction takes a pair of models, $K_A$ and $K_B$, and HyperLTL formula $\varphi$ as input, and generates a positive answer if and only if the answer to the original mapping synthesis problem has a solution. The reduced BMC problem deals with a HyperLTL formula of the form $\exists \pi_M. \Xi$, where identifying a witness to $\pi_M$ is analogous to synthesizing a function $m$ in the original mapping synthesis problem. Here, $\Xi$ is formula that encodes the original formula $\varphi$ and its relation to the existence of a mapping function as well as the models. A prominent characteristic of our technique is that it can handle formulas with quantifier alternations, which is typically the source of difficulty in verifying information-flow security policies.

Although we have so far motivated this work with synthesizing a mapping that encodes platform implementation decisions, the notion of mappings here is more general, and our mapping synthesis framework can be used to solve a number of other tasks that are relevant to secure program development.

In general, the pair of models $K_A$ and $K_B$ need not necessarily represent abstract/high-level and concrete/platform machines, respectively. Moreover, the direction of mapping from abstract to concrete may be reversed, and instead be from concrete to abstract. For instance, $K_A$ may represent a given program and $K_B$ a specification of desired behaviors, and $m$ could be synthesized as a refinement mapping [1] from the concrete model $K_A$ to the more abstract model $K_B$, to show that every behavior of $K_A$ is also one of $K_B$ (see Section V). In addition, our notion of mappings can also be used to enforce a property instead of preserving it: this is done in cases where a given program ($K_A$) violates a desired property (e.g., non-interference), which is then enforced by being deployed in a target environment ($K_B$) with additional behavioral constraints (see example introduced in Section II-C). To summarize, our synthesis approach can be applied to many scenarios including synthesizing implementation decisions (see example in Fig. 1, and [38]), synthesizing refinement mapping (see Section V), and property enforcement (see Section II-C).

We have built a prototype implementation of our technique. As the main demonstration of our tool, we show how our approach can be used to achieve preservation of secrecy under refinement. In [4], the authors prove that their notion of refinement preserves what they call secrecy properties, but only hint at the possibility of automated techniques for verifying that a concrete program is a secrecy-preserving refinement of an abstract program. We take one step further and show how our approach can be used to synthesize such a refinement mapping.

In particular, our evaluation consists of two case studies of secrecy-preserving refinement, where the goal is to find a valid mapping from a concrete program to an abstract specification. The first case study is the well-known battleship game, a strategic guessing game for two players, where the locations of each player’s fleet of ships are marked on a grid and concealed from each other. We show how our technique can be used to synthesize a refinement mapping from a concrete implementation of the game to an abstract specification that does not leak additional information about the fleet locations. Our second case study is a password checking program that compares a user input with a server-side password. A program implementing this procedure should not leak potentially critical information about the password, such as the size of the stored data. Here, we show how our approach can be used to automatically check whether a given implementation preserves the secrecy of the sensitive data by synthesizing a mapping from the concrete implementation to the abstract specification.

---

1Our code and case studies are available at https://bit.ly/3aJwKbf.
Organization: The rest of the paper is organized as follows. In Section II, we present the preliminary concepts. The formal statement of mapping synthesis is given in Section III. Section IV describes the reduction of the mapping synthesis problem to HyperLTL model checking. The application of mapping synthesis to secrecy-preserving refinement mapping is explained in Section V, while the case studies are presented in Section VI. We discuss related work in Section VII and conclude in Section VIII.

II. Preliminaries

A. Kripke Structures

Let AP be a finite set of atomic propositions and $\Sigma = 2^{AP}$ be the alphabet, where $2^X$ denotes the powerset (set of all subsets) of a set $X$. A letter is an element of $\Sigma$. A trace $t \in \Sigma^\omega$ over alphabet $\Sigma$ is an infinite sequence of letters: $t = t(0)t(1)t(2)\cdots$. We model systems as finite-state Kripke structures.

Definition 1: A Kripke structure is a tuple $K = (S, S^0, \delta, AP, L)$, where

- $S$ is a finite set of states,
- $S^0 \subseteq S$ is the set of initial states,
- $\delta \subseteq S \times S$ is a transition relation,
- AP is the set of atomic propositions, and
- $L: S \rightarrow 2^{AP}$ is a labeling function.

We require that for each $s \in S$, there exists $s' \in S$, such that $(s, s') \in \delta$. That is, every state must have at least one successor (no deadlocks).

The size of the Kripke structure is the number of its states. A loop in $K$ is a finite sequence $s(0)s(1)\cdots s(n)$, such that $(s(i), s(i+1)) \in \delta$, for all $0 \leq i < n$, and $(s(n), s(0)) \in \delta$. We call a Kripke structure acyclic, if the only loops are self-loops on otherwise terminal states, i.e., on states that have no other outgoing transition. Since Definition 1 does not allow terminal states, we only consider acyclic Kripke structures that have such added self-loops.

A path of a Kripke structure is an infinite sequence of states $s(0)s(1)\cdots \in S^\omega$, such that $s(0) \in S_0$, and $(s(i), s(i+1)) \in \delta$, for all $i \geq 0$. A trace of a Kripke structure is a sequence $t(0)t(1)t(2)\cdots \in \Sigma^\omega$, such that there exists a path $s(0)s(1)\cdots \in S^\omega$ with $t(i) = L(s(i))$, for all $i \geq 0$. We denote by $\text{Traces}(K,s)$ the set of all traces of $K$ with paths that start in state $s \in S$, and use $\text{Traces}(K,s)$ as a shorthand for $\bigcup_{s \in S_0} \text{Traces}(K,s)$.

B. The Temporal Logic HyperLTL

Linear Temporal Logic (LTL) is a standard logic used in formal specification and verification of reactive systems [5]. HyperLTL [15] is an extension of LTL for hyperproperties.

1) Syntax: The syntax of HyperLTL formulas is defined inductively by the following grammar:

- $\phi ::= \exists \pi. \phi \mid \forall \pi. \phi \mid \phi$
- $\phi ::= p_\pi \mid \neg \phi \mid \phi \land \phi \mid \bigotimes \phi \mid \phi \bigcirc \phi$

where $p \in AP$ is an atomic proposition and $\pi$ is a trace variable from an infinite supply of variables $V$. The Boolean connectives $\land$ and $\lor$ have the usual meaning. $U$ is the temporal until operator, and $\bigcirc$ is the temporal next operator. We also consider syntactic sugar true $\equiv p_\pi \lor \neg p_\pi$, false $\equiv \neg true$, $\phi_1 \land \neg \phi_2 \equiv \neg (\phi_1 \lor \neg \phi_2)$, and $\phi_1 \lor \phi_2 \equiv \neg \phi_1 \lor \neg \phi_2$. Also, the derived temporal operators include eventually $\bigotimes \phi \equiv true$ and globally $\bigcirc \phi \equiv \neg \bigotimes \neg \phi$. The quantified formulas $\exists \pi$ and $\forall \pi$ are read as “along some trace $\pi$” and “along all traces $\pi$”, respectively.

A formula is closed (i.e., a sentence) if all trace variables used in the formula are quantified. We assume, without loss of generality, that all formulas we discussed in this paper are closed, and no variable is quantified twice. We use $\text{Vars}(\phi)$ for the set of trace variables used in formula $\phi$.

2) Semantics: An interpretation $\mathcal{T} = (T_\pi)_{\pi \in \text{Vars}(\phi)}$ of a formula $\phi$ consists of a tuple of sets of traces, one set $T_\pi$ per trace variable $\pi$ in $\text{Vars}(\phi)$. We use $T_\pi$ for the set of traces assigned to $\pi$. The idea here is to allow trace quantifiers to range over different systems. We use this feature to solve mapping synthesis problem as explained in Section IV. With this feature, each set of traces comes from its own Kripke structure. Thus, the set of traces that $\pi$ can range over, $T_\pi$, comes from a specific $K_\pi$, i.e., $T_\pi = \text{Traces}(K_\pi)$. We use $K = (K_\pi)_{\pi \in \text{Vars}(\phi)}$ to denote a family of Kripke structures. To simplify, we write $\mathcal{T} = \text{Traces}(K)$ to denote the set of sets of traces derived from the family of Kripke structures. The multi-model nature of our interpretation allows us to synthesize the mappings between any arbitrary two models $K_A$ and $K_B$ represented by two different Kripke structures.

The semantics of HyperLTL is defined with respect to a trace assignment, which is a partial map $\Pi: \text{Vars}(\phi) \rightarrow \Sigma^\omega$. The assignment with an empty domain is denoted by $\Pi_\emptyset$. Given a trace assignment $\Pi$, a trace variable $\pi$, and a trace $t \in \Sigma^\omega$, we denote by $\Pi[\pi \rightarrow t]$ the assignment that coincides with $\Pi$ everywhere except at $\pi$, which in $\Pi[\pi \rightarrow t]$ is mapped to trace $t$. The satisfaction of a HyperLTL formula $\phi$ is a binary relation $\models$ that associates a formula to the models $(T_\Pi,i)$, where $i \in \mathbb{Z}_{\geq 0}$ is a pointer that indicates the current evaluating position. The semantics of HyperLTL is defined in Fig. 2.

We say that an interpretation $\mathcal{T}$ satisfies a HyperLTL formula $\phi$, denoted by $\mathcal{T} \models \phi$, if $(T_\Pi,i_0) \models \phi$. We say that a family of Kripke structures $K$ satisfies a specification $\phi$, denoted by $K \models \phi$, if it holds that $\text{Traces}(K) \models \phi$.

It is often the case that the number of Kripke structures in the family $K$ is not equal to the number of quantifiers in the HyperLTL formula $\phi$ that $K$ is checked against. In such cases, we need to specify explicitly which trace variable in $\phi$ corresponds to which Kripke structure in $K$. We do this by appropriately subscriptsing trace variables in $\phi$. For example, if $K = (K_A, K_B, K_C)$ and $\phi = \forall \pi_A.\exists \pi_B.\forall \pi_C.\exists \pi_A.\phi$, then $\pi_A$ and $\pi_A'$ correspond to traces of $K_A$, $\pi_B$ to traces of $K_B$, and $\pi_C$ to those of $K_C$. In other words, $K \models \phi$ means that $(\text{Traces}(K_A), \text{Traces}(K_C), \text{Traces}(K_B), \text{Traces}(K_A')) \models \phi$. A special case is when $K$ contains only one single Kripke structure $K$. In such a case, we do not subscript the trace variables of $\phi$ as they all implicitly correspond to traces of $K$. 


C. Running Example: Enforcing Non-Interference

We use the following example to clarify the preliminary concepts in this section as well as the notion of mapping synthesis which will be formalized in Section III.

Consider the Kripke structure $K_A$ shown in Fig 3 representing an abstract model of communication between three parties Alice, Bob, and Eve. In addition to the states $S = \{s_0, s_1, \ldots, s_6\}$, this system also has a Boolean state variable sec representing the secret (not modeled as a state for reasons of space). Thus, the complete state of $K_A$ is the pair $(sec, s)$, where $s \in S$. Hence, $K_A$ has a total of $2 \times 7 = 14$ states (not all these states are reachable). The state variable sec is initialized non-deterministically to either 0 or 1. Hence, $K_A$ has two possible initial states: $(0, s_0)$ and $(1, s_0)$. After initialization, the value of sec remains constant. The conditions $sec = 0$ and $sec = 1$ on the transitions $(s_4, s_5)$ and $(s_4, s_6)$ mean that the corresponding transition exists only if the value of sec satisfies the condition. For instance, there is a transition from $(0, s_4)$ to $(0, s_5)$, and also from $(1, s_4)$ to $(1, s_6)$, but there is no transition from $(1, s_4)$ to $(1, s_5)$. The terminating states are those with self-loops, i.e., states $s_2$, $s_3$, $s_5$, and $s_6$.

The set depicted inside each state is the label of that state. The intuition behind the atomic propositions is as follows. Each message communication is of the form to.from.content. For example, Alice.Bob.sec indicates that Alice sends the secret to Bob, and Bob.Eve.pub means Bob sends Eve some arbitrary public message. The information possessed by a party is represented by party_info. For example, Bob.sec means that Bob knows the value of the secret, and Eve_secNonEmpty means that Eve knows that the value of the secret message is not empty, but Eve does not know the actual value of the message.

The security property of non-interference [30] requires that low-security variables should be independent from high-security variables, thus, preserving secure information flow. Let us illustrate non-interference on our example. Assume that in $K_A$, the state variable sec is a high-security variable and the information that Eve knows (i.e., Eve_secNonEmpty) is a low-security variable. Non-interference can be expressed by the following HyperLTL formula:

$$\varphi_{NI} = \forall \pi_1, \exists \pi_2. (sec_{\pi_1} \neq sec_{\pi_2}) \land \Box (Eve_{secNonEmpty}_{\pi_1} \leftrightarrow Eve_{secNonEmpty}_{\pi_2})$$

In our example, $K_A$ violates the non-interference property, that is, $K_A \not\models \varphi_{NI}$. Consider trace $t_1$ corresponding to the path $s_0s_1s_4s_6$, where initially $sec = 1$. Observe that when $sec = 1$, transition $(s_4, s_6)$ is taken from state $s_4$ and $Eve_{secNonEmpty}$ holds in $s_6$. Now, according to $\varphi_{NI}$, for any such trace $t_1$, we must be able to find another trace $t_2$ of $K_A$, such that $(1, t_2)$ has a different value of $sec$ than $t_1$ and (2) at each step $Eve_{secNonEmpty}$ holds at $t_2$ if and only if it holds at $t_1$. The first constraint implies that $sec$ must be 0 in the initial state of $t_2$. The second constraint implies that at the fourth step, $Eve_{secNonEmpty}$ must hold in $t_2$ (because it holds at the same step in $t_1$). But this means that $t_2$ must also follow the path $s_0s_1s_4s_6$, which is impossible, since $sec = 0$ in $t_2$, which violates the condition from $s_4$ to $s_6$. Thus, we cannot find such a trace $t_2$, which means that $K_A \not\models \varphi_{NI}$.

In the sequel we will use this running example to show how mapping synthesis can be used to enforce non-interference.

III. Problem Statement

The mapping synthesis problem was introduced in [38]. Mapping synthesis relies on the notion of mapping composition, a generalization of parallel composition, where processes are allowed to synchronize over actions that are not common to their alphabets. In this paper, we use a different variant of the mapping composition operator than the one defined in [38]. Specifically, mapping composition in [38] is defined as an asynchronous parallel composition of two labeled transition systems, where mapping controls the synchronization of certain transitions by mapping the labels annotating the
transitions. In our setting, mapping composition is defined as a synchronous parallel composition of two Kripke structures, where mapping controls the set of atomic propositions annotating the product states. This technical change allows us to handle more easily specification languages like HyperLTL that refer to atomic propositions on system states. As it turns out, HyperLTL handle more easily specification languages like

\[ \text{transitions.} \]

In our setting, mapping composition is defined as a synchronous parallel composition of two Kripke structures, where mapping controls the set of atomic propositions annotating the product states. This technical change allows us to handle more easily specification languages like HyperLTL that refer to atomic propositions on system states. As it turns out, HyperLTL handle more easily specification languages like

\[ \text{transitions.} \]

In our setting, mapping composition is defined as a synchronous parallel composition of two Kripke structures, where mapping controls the set of atomic propositions annotating the product states. This technical change allows us to handle more easily specification languages like HyperLTL that refer to atomic propositions on system states. As it turns out, HyperLTL handle more easily specification languages like

\[ \text{transitions.} \]

In our setting, mapping composition is defined as a synchronous parallel composition of two Kripke structures, where mapping controls the set of atomic propositions annotating the product states. This technical change allows us to handle more easily specification languages like HyperLTL that refer to atomic propositions on system states. As it turns out, HyperLTL handle more easily specification languages like

\[ \text{transitions.} \]

In our setting, mapping composition is defined as a synchronous parallel composition of two Kripke structures, where mapping controls the set of atomic propositions annotating the product states. This technical change allows us to handle more easily specification languages like HyperLTL that refer to atomic propositions on system states. As it turns out, HyperLTL handle more easily specification languages like

\[ \text{transitions.} \]

In our setting, mapping composition is defined as a synchronous parallel composition of two Kripke structures, where mapping controls the set of atomic propositions annotating the product states. This technical change allows us to handle more easily specification languages like HyperLTL that refer to atomic propositions on system states. As it turns out, HyperLTL handle more easily specification languages like

\[ \text{transitions.} \]

In our setting, mapping composition is defined as a synchronous parallel composition of two Kripke structures, where mapping controls the set of atomic propositions annotating the product states. This technical change allows us to handle more easily specification languages like HyperLTL that refer to atomic propositions on system states. As it turns out, HyperLTL handle more easily specification languages like

\[ \text{transitions.} \]
IV. REDUCING HYPERLTL MAPPING SYNTHESIS TO HYPERLTL MODEL CHECKING

In this section, we provide a method to reduce the mapping synthesis problem to HyperLTL verification and we prove the correctness of the method. We also demonstrate this method on our running example.

A. The Reduction

In a nutshell, our reduction works as follows. In an instance of the mapping synthesis problem, we are given two Kripke structures $K_A$ and $K_B$, and a HyperLTL formula $\varphi$. The goal is to find (if there exists) a mapping $m$ such that $(K_A \parallel m K_B)$ satisfies $\varphi$. In our reduction, we will construct a new Kripke structure $K_M$ which represents the set of all candidate mappings. We will also construct a new HyperLTL formula $\Phi_{ABM} = \exists \pi_M. \Xi$, which encodes the existence of a good mapping (the quantified trace variable $\pi_M$ is instantiated by traces of $K_M$). The idea is that verifying $\exists \pi_M. \Xi$ is analogous to synthesizing a mapping for $K_A$ and $K_B$. Furthermore, $\Xi$ encodes the original formula $\varphi$ adapted to the verification problem. Then, finding a valid mapping $m$ is reduced to checking whether the family $(K_M, K_A, K_B)$ satisfies $\Phi_{ABM}$. We now present this method in detail.

1) The Kripke structure $K_M$: Consider two Kripke structures $K_A = (S_A, S_0^A, \delta_A, AP_A, L_A)$ and $K_B = (S_B, S_0^B, \delta_B, AP_B, L_B)$. We denote by $\text{Mappings}(AP_A, AP_B)$ the set of all possible mappings, that is, the set of all possible partial functions from $AP_A$ to $AP_B$. In principle, each proposition of $K_A$ can be mapped to any one of the propositions of $K_B$, or to none, since mappings are partial functions. This gives a total of $|\text{Mappings}(AP_A, AP_B)| = (|AP_B| + 1)|AP_A|$ possible mappings.

We construct Kripke structure $K_M$ as follows:

- $S_M = \{ s_m \mid m \in \text{Mappings}(AP_A, AP_B) \}$
- $S_0^M = S_M$
- $\delta_M = \{(s, s) \mid s \in S_M\}$
- $AP_M = AP_A \times AP_B$
- $L_M : S_M \rightarrow 2^{AP_M}$ s.t. for any $s_m \in S_M$, we have $L(s_m) = \{(a, b) \in AP_M \mid m(a) = b\}$.

The set of states $S_M$ encodes the set of all possible mappings. Note that $AP_A$ and $AP_B$ are finite sets, therefore the number of possible mappings is also finite. Thus, $K_M$ is a finite-state Kripke structure. Choosing a mapping means selecting one of the initial states in $K_M$. Once chosen, this choice remains fixed. This is ensured by the set of self-loops of each state of $K_M$. Observe that each proposition in $AP_M$ is a pair $(a, b)$, where $a$ is a proposition of $K_A$ and $b$ is a proposition of $K_B$. The idea is that proposition $(a, b)$ holds at a given state $s$ of $K_M$ iff the mapping represented by $s$ maps $a$ to $b$. This is ensured by the labeling function $L_M$.

Example: Consider the two Kripke structures $K_A$ and $K_B$ in Figs. 3 and 4, respectively. Observe that there are four atomic propositions in each of them: $|AP_A| = |AP_B| = 4$. Then, the Kripke structure $K_M$ has $(4 + 1)^4 = 625$ states (see Figure 5). $M_0, M_1, M_2, \ldots, M_{624}$ correspond to the 625 possible mappings between $K_A$ and $K_B$. Each mapping is represented by a subset of $AP_M$ (including the empty subset which represents the empty mapping). In this example, we have the following possible mappings:

- $M_0 = \{\}$
- $M_1 = \{(\text{Alice}.\text{Bob}.\text{sec}, \text{S}.\text{R}.\text{pub})\}$
- $M_2 = \{(\text{Bob}.\text{Eve}.\text{pub}, \text{S}.\text{R}.\text{pub})\}$
- $M_3 = \{(\text{Alice}.\text{Bob}.\text{sec}, \text{S}.\text{R}.\text{pub})$
- $(\text{Bob}.\text{pub}, \text{S}.\text{R}.\text{sec})$
- $(\text{Eve}.\text{sec}.\text{NonEmpty}, \text{R}.\text{sec})\}$

2) The Global HyperLTL Formula $\Phi_{ABM}$: Let the original HyperLTL formula $\varphi$ be of the form: $Q_1 \pi_1. Q_2 \pi_2. \ldots. Q_n \pi_n. \psi$ where $\psi$ is quantifier-free and each $Q_i \in \{\forall, \exists\}$, for $1 \leq i \leq n$. Then, $\Phi_{ABM}$ is constructed as follows:

\[
\Phi_{ABM} \text{ def } = \exists \pi_M. \Xi
\]

\[
\Xi \text{ def } = Q_1 \pi_{A_1}. Q_1 \pi_{B_1}. \ldots. Q_n \pi_{A_n}. Q_n \pi_{B_n}. \Psi
\]

\[
\Psi \text{ def } = \varphi_{map_1} \circ_1 \varphi_{map_2} \circ_2 \ldots. \varphi_{map_n} \circ_n \psi_{\text{new}}
\]

where $\pi_M$ ranges over $K_M$, trace variables $\pi_{A_i}$ range over $K_A$, while $\pi_{B_i}$ range over $K_B$, for $1 \leq i \leq n$. Also, $\pi_i = \wedge$ if $Q_i = \exists$ and $\pi_i = \rightarrow$ if $Q_i = \forall$, where $\pi_i$ has precedence over all the preceding such operators. The subformulas $\varphi_{map_i}$ and $\psi_{\text{new}}$ are defined below. Intuitively, existence of a trace of $K_M$ such that $\Xi$ is satisfied encodes existence of a valid mapping. In $\varphi$, each trace variable $\pi_i$ ranges over the traces of $K_A \parallel m K_B$. In $\Phi_{ABM}$, we replace $\pi_i$ by two trace variables $\pi_{A_i}$ and $\pi_{B_i}$ ranging over $K_A$ and $K_B$, respectively. These variables together with $\varphi_{map_i}$ encoding the mapping composition constraints describe the requirements of composite traces of $K_A \parallel m K_B$. 
We now proceed to define the subformulas $\varphi_{\text{map}_i}$ and $\psi_{\text{new}}$. First, we define, for $1 \leq i \leq n$:

$$\varphi_{\text{map}_i} \overset{\text{def}}{=} \bigwedge_{(a,b) \in \text{AP}_M} \Box ((a,b)_{\pi M} \rightarrow (a_{\pi A_i} \rightarrow b_{\pi B_i}))$$

Intuitively, $\varphi_{\text{map}_i}$ states that if $(a, b)$ holds in a trace of $K_M$ (i.e., if $a$ is mapped to $b$ in the selected mapping), then whenever $a$ holds at a certain step in the trace of $K_A$, $b$ must hold at the same step in the trace of $K_B$. This ensures that the pair of traces from $K_A$ and $K_B$ fulfill the mapping composition requirements.

We next construct $\psi_{\text{new}}$ to be the subformula obtained by replacing each $a_{\pi_i} \in \text{AP}_A$ in $\psi$ with $a_{\pi A_i}$, for $1 \leq i \leq n$.

**B. Illustration of the Reduction on our Running Example**

Recall the Kripke structures $K_A$ and $K_B$ from Figs. 3 and 4, and the HyperLTL formula $\varphi_{\text{NI}}$. Our goal is to find a mapping $m$, such that $(K_A \parallel \parallel_{m} K_B) \models \varphi_{\text{NI}}$, by reducing it to a HyperLTL verification problem of checking whether the family $(K_M, K_A, K_B)$ satisfies the constructed global formula w.r.t. $\varphi_{\text{NI}}$, called $\Phi_{\text{NI-ABM}}$. We now use this running example to demonstrate the construction of $\Phi_{\text{NI-ABM}}$ and how the satisfiability of $\Phi_{\text{NI-ABM}}$ gives us a valid mapping that enforces $K_A$ to satisfy the property $\varphi_{\text{NI}}$.

Recall the non-interference property introduced in Section II. The HyperLTL formula $\varphi_{\text{NI}}$ can be rewritten as follows:

$$\varphi_{\text{NI}} = \forall \pi_1, \exists \pi_2, \psi_{\text{NI}}$$

$$\psi_{\text{NI}} = (\exists_{\pi_1, \exists_{\pi_2} \forall_{\pi_2}} \land \Box (\text{Eve}_{\text{secNonEmpty}}_{\pi_1} \leftrightarrow \text{Eve}_{\text{secNonEmpty}}_{\pi_2}))$$

where $\psi_{\text{NI}}$ is the quantifier-free part of $\varphi_{\text{NI}}$. According to our definition, the formula $\Phi_{\text{NI-ABM}}$ is thus:

$$\Phi_{\text{NI-ABM}} = \exists_{\pi M, \forall_{\pi A_1}, \forall_{\pi B_1}, \exists_{\pi A_2}, \exists_{\pi B_2}} (\varphi_{\text{map}_1} \rightarrow (\varphi_{\text{map}_2} \land \psi_{\text{NI-new}}))$$

We next present in detail how the elements $\varphi_{\text{map}_1}, \varphi_{\text{map}_2}$, and $\psi_{\text{NI-new}}$ are constructed.

First, the mapping subformulas $\varphi_{\text{map}_1}$ and $\varphi_{\text{map}_2}$ are as follows:

$$\varphi_{\text{map}_1} = \bigwedge_{(a,b) \in \text{AP}_M} \Box ((a,b)_{\pi M} \rightarrow (a_{\pi A_1} \rightarrow b_{\pi B_1}))$$

$$\varphi_{\text{map}_2} = \bigwedge_{(a,b) \in \text{AP}_M} \Box ((a,b)_{\pi M} \rightarrow (a_{\pi A_2} \rightarrow b_{\pi B_2}))$$

The purpose of $\varphi_{\text{map}_1}$ and $\varphi_{\text{map}_2}$ is to make sure that the labels of traces fulfill the mapping constraints. That is, for all $a \in \text{AP}_A$ and $b \in \text{AP}_B$, if $m(a) = b$, whenever $a$ holds at a certain step in the trace of $K_A$, $b$ must hold at the same step in the trace of $K_B$.

For instance, if a mapping chose to map $\text{Alice}.\text{Bob.}sec_{\pi A_1}$ to $\text{S.R.}sec_{\pi B_1}$, then $\varphi_{\text{map}_1}$ and $\varphi_{\text{map}_2}$ will impose $(\text{Alice}.\text{Bob.}sec_{\pi A_1} \rightarrow \text{S.R.}sec_{\pi B_1})$ and $(\text{Alice}.\text{Bob.}sec_{\pi A_2} \rightarrow \text{S.R.}sec_{\pi B_2})$, respectively.

Next, we construct $\psi_{\text{NI-new}}$ by replacing each $a_{\pi_i} \in \text{AP}_A$ in $\psi_{\text{NI}}$ with $a_{\pi A_i}$ for $i \in \{1, 2\}$ as follows,

$$\psi_{\text{NI-new}} = (\sec_{\pi A_1} \land \neg \sec_{\pi A_2}) \land \Box (\text{Eve}_{\text{secNonEmpty}}_{\pi A_1} \leftrightarrow \text{Eve}_{\text{secNonEmpty}}_{\pi A_2})$$

Finally, by using the bounded model checking approach proposed in [35], we check whether:

$$(K_M, K_A, K_B) \models \Phi_{\text{NI-ABM}}$$

is satisfiable (SAT) or not. The solver returns SAT, which implies that the outer-most existentially quantified variable $\pi M$ has been instantiated by a witness trace that represents a valid mapping. The witness trace represents the following mapping:

$$M_{\text{valid}} = \{(\text{Alice}.\text{Bob.}sec, \text{S.R.}sec), (\text{Bob}.\text{Eve.}pub, \text{S.R.}pub)\}$$

Let us evaluate $M_{\text{valid}}$ together with $K_A$ and $K_B$ in Figs. 3 and 4, respectively. As we discussed in Section II, the path $s_0s_1s_4s_6$ in $K_A$ corresponds to Alice sending Bob a non-empty secret, followed by Bob sending Eve a public message but leaking the information about “secret is not empty” to Eve, thus violating $\varphi_{\text{NI}}$. However, if we map $\text{Alice}.\text{Bob.}sec$ to $\text{S.R.}sec$, and $\text{Bob}.\text{Eve.}pub$ to $\text{S.R.}pub$, as prescribed by $M_{\text{valid}}$, the path $s_0s_1s_4s_6$ is eliminated after mapping composition. This is because the corresponding trace on $K_B$, i.e. $\text{S.R.}sec$ followed by $\text{S.R.}pub$, does not exist. Hence, the path that violates $\varphi_{\text{NI}}$ is now eliminated, resulting in $M_{\text{valid}}$ begin indeed a mapping that enforces non-interference.

As mentioned at the end of Section III, our method is also capable of finding invalid mappings, i.e., mappings that violate the given HyperLTL formula. Let us illustrate this on our running example. First, we negate $\varphi_{\text{NI}}$ to obtain:

$$\neg \varphi_{\text{NI}} = \exists \pi_1, \forall \pi_2, \neg \psi_{\text{NI}}$$

Next we construct the global HyperLTL formula $\Phi_{\text{not-NI-ABM}}$ w.r.t. $\neg \varphi_{\text{NI}}$, as follows:

$$\Phi_{\text{not-NI-ABM}} = \exists_{\pi M, \forall_{\pi A_1}, \forall_{\pi B_1}, \forall_{\pi A_2}, \forall_{\pi B_2}} (\varphi_{\text{map}_1} \land (\varphi_{\text{map}_2} \rightarrow \neg \psi_{\text{NI-new}}))$$

Once again, the solver returns SAT meaning that it now found a mapping violating non-interference. In particular, the following mapping is invalid:

$$M_{\text{invalid}} = \{(\text{Alice}.\text{Bob.}sec, \text{S.R.}pub), (\text{Bob}.\text{Eve.}pub, \text{S.R.}sec)\}$$

This mapping allows the path $s_0s_1s_4s_6$ on $K_A$ to happen, with the corresponding path $s_0s_1s_3s_3$ on $K_B$. Thus, under $M_{\text{invalid}}$ the mapping composition of $K_A$ and $K_B$ violates non-interference.

**C. Proof of Correctness**

We now present our main theoretical result, namely, the correctness of our reduction method. That is, we show that the answer to the mapping synthesis problem is affirmative if and only if the answer to the corresponding HyperLTL verification problem is positive.
Theorem 1: Given a closed HyperLTL formula $\varphi$ of the form $Q_1\pi_1.Q_2\pi_2.A_1\psi_1$ and Kripke structures $K_A$ and $K_B$, there exists a mapping $m$ such that $(K_A \mid m,K_B) \models \varphi$ if and only if $(K_M,K_A,K_B) \models \Phi_{ABM}$, according to the above construction.

Proof:

$(\Rightarrow)$ Suppose that there exists a mapping $m$ such that $(K_A \mid m, K_B) \models \varphi$. We need to show that $(K_M,K_A,K_B) \models \Phi_{ABM}$. We proceed by induction on the number of quantifiers in $\varphi$.

For the base case where $n = 1$, we have $\varphi = Q_1\pi_1.\psi$. Then, $\Phi_{ABM}$ will be of the form $\exists \gamma_1.M_1\pi_1.A_1\gamma_1.\psi_1 \land \psi_{new}$, where $Q_1 \in \{\forall,\exists\}$. We first instantiate the outermost quantifier $\exists \gamma_1$ of $\Phi_{ABM}$. There exists a unique state $s \in K_M$ such that the label of $s$ corresponds to mapping $m$, that is, for all $(a,b) \in L(s)$ we have $m(a) = b$. In $K_M$, by construction, we instantiate $\pi_m$ with $t_M = s^\omega$. We now distinguish two cases. If $Q_1 = \exists$, then there exists a trace $t_1 \in (K_A \mid m, K_B)$, such that

$$\left(\text{Traces}(K_A \mid m, K_B), \Pi[\pi_1 \rightarrow t_1], 0 \right) \models \psi.$$ 

For existential quantifier, $\forall_1$ is $\land$, meaning that $t_M, t_{A_1}$, and $t_{B_1}$ must satisfy $\varphi_{map_1} \land \psi_{new}$. The projection of $t_1$ on $K_A$ and $K_B$ derives two traces $t_{A_1}$ and $t_{B_1}$, respectively. Since $t_1 \in \text{Traces}(K_A \mid m, K_B)$, it implies that traces $t_{A_1}$ and $t_{B_1}$ are obtained according to mapping function $m$. This in turn ensures that $(t_M, t_{A_1}, t_{B_1}) \models \varphi_{map_1}$. Because for each $a \in A_P$ and $b \in A_B$, if $m(a) = b$, then $\varphi_{map_1}$ is satisfiable only if $(a_{\pi_1} \rightarrow b_{\pi_1})$ always holds. That is, the instantiated $\pi_m$ and $\pi_{new}$ fulfills the mapping function $m$. From here, $t_{A_1}$ and $t_{B_1}$ are sufficient to instantiate $\pi_m$ and $\pi_{new}$ to satisfy $\varphi_{map_1}$. Since $t_1$ satisfies $\psi$, so $t_{A_1}$ also satisfies $\psi_{new}$. That is,

$$\left(\text{Traces}(K_M,K_A,K_B), \Pi[\pi_1 \rightarrow t_M],
\pi_{A_1} \rightarrow t_{A_1}, \pi_{B_1} \rightarrow t_{B_1}, 0 \right) \models \varphi_{map_1} \land \psi_{new}$$

If $Q_1 = \forall$, we have $\forall_1$ is $\rightarrow$, meaning that $t_M, t_{A_1}$, and $t_{B_1}$ must satisfy $\varphi_{map_1} \rightarrow \psi_{new}$. The segment $Q_1\pi_1, Q_1\pi_{B_1}$ represents all paths on the composed model of $K_A$ and $K_B$. Take an arbitrary pair of traces $t_{A_1} \in \text{Traces}(K_A)$ and $t_{B_1} \in \text{Traces}(K_B)$. If $(t_M, t_{A_1}, t_{B_1}) \not\models \varphi_{map_1}$, then $\varphi_{map_1} \rightarrow \psi_{new}$ is vacuously true and, hence, the HyperLTL verification problem is consequently true as well. Now, consider the case where $(t_M, t_{A_1}, t_{B_1}) \models \varphi_{map_1}$. By the assumption of the existence of a valid mapping function $m$, by composing $t_{A_1}$ and $t_{B_1}$ into one trace $t_1$ according to $m$, we have:

$$\left(\text{Traces}(K_A \mid m, K_B), \Pi[\pi_1 \rightarrow t_1], 0 \right) \models \psi,$$

because $\pi_1$ is universally quantified and can be instantiated by any arbitrary trace from the mapping composition. Again, since we are able to obtain $t_{A_1}$ and $t_{B_1}$ by projecting $t_1$ on $K_A$ and $K_B$, $t_{A_1}$ and $t_{B_1}$ will be a pair of traces that fulfills $\varphi_{map_1}$ and $\psi_{new}$. Hence, the following always holds

$$\left(\text{Traces}(K_M,K_A,K_B), \Pi[\pi_1 \rightarrow t_M],
\pi_{A_1} \rightarrow t_{A_1}, \pi_{B_1} \rightarrow t_{B_1}, 0 \right) \models \varphi_{map_1} \rightarrow \psi_{new}.$$ 

For inductive step, the inductive hypothesis is as follows. Suppose there is a mapping function $m : AP_A \rightarrow AP_B$ that fulfills $(K_A \mid m,K_B) \models \varphi$, where $\varphi = Q_1\pi_1.Q_2\pi_2.A_1\psi_1$, it is true that $(K_M,K_A,K_B) \models \Phi_{ABM}$ for any $k \geq 1$. We now want to show $k+1$ holds.

From inductive hypothesis, we know for any $k$, each assignment $[\pi_i \rightarrow t_i]$ is valid, where $1 \leq i \leq k$ regardless of whether $Q_i = \forall$ or $Q_i = \exists$. As a result, the following:

$$\left(\text{Traces}(K_M,K_A,K_B), \Pi[\pi_1 \rightarrow t_1],
\pi_2 \rightarrow t_2,
\vdots
\pi_k \rightarrow t_k, 0 \right) \models \psi,$$

must holds. It implies that we are able to construct the HyperLTL formula for any $1 \leq i \leq k$ by projecting $t_i$ to $K_A$ and $K_B$ and obtain $\pi_{A_i}$ and $\pi_{B_i}$, respectively, as follows:

$$\left(\text{Traces}(K_M,K_A,K_B), \Pi[\pi_i \rightarrow t_M],
\pi_{A_1} \rightarrow t_{A_1}, \pi_{B_1} \rightarrow t_{B_1}, \pi_{A_2} \rightarrow t_{A_2}, \pi_{B_2} \rightarrow t_{B_2},
\vdots
\pi_{A_k} \rightarrow t_{A_k}, \pi_{B_k} \rightarrow t_{B_k}, 0 \right) \models \left(\varphi_{map_1} \circ_1 \varphi_{map_2} \circ_2 \cdots \varphi_{map_k} \circ_k \psi_{new}\right).$$

From here, for $Q_{k+1}$, the instantiation of $\pi_{A_{k+1}}$ and $\pi_{B_{k+1}}$ can be done similarly with the same approaches as in the base case, depending on the corresponding $t_{k+1}$ and $Q_{k+1}$, where $Q_{k+1} \in \{\exists,\forall\}$. Hence, after instantiating each trace variable in the HyperLTL formula, the following must hold:

$$\left(\text{Traces}(K_M,K_A,K_B), \Pi[\pi_i \rightarrow t_M],
\pi_{A_1} \rightarrow t_{A_1}, \pi_{B_1} \rightarrow t_{B_1}, \pi_{A_2} \rightarrow t_{A_2}, \pi_{B_2} \rightarrow t_{B_2},
\vdots
\pi_{A_k} \rightarrow t_{A_k}, \pi_{B_k} \rightarrow t_{B_k}, \pi_{A_{k+1}} \rightarrow t_{A_{k+1}}, \pi_{B_{k+1}} \rightarrow t_{B_{k+1}}, 0 \right) \models \left(\varphi_{map_1} \circ_1 \varphi_{map_2} \circ_2 \cdots \varphi_{map_k} \circ_k \varphi_{map_{k+1}} \circ_{k+1} \psi_{new}\right).$$

This is indeed the case because when $(t_1,t_2,\ldots,t_{k+1}) \models \psi$ in the composed Kripke structure $(K_A \mid m, K_B)$, then by construction, the mapping and the projections of traces $(t_1,t_2,\ldots,t_{k+1})$ on $K_M, K_A$ and $K_B$ satisfy $\psi_{new}$, that is, $(t_M,t_{A_1},t_{B_1},t_{A_2},t_{B_2},\ldots,t_{A_{k+1}},t_{B_{k+1}}) \models \psi_{new}$. 


We now prove the reverse direction. Suppose $(K_M, K_A, K_B) = \Phi_{ABM}$. Then $\pi_M$ can be instantiated by some $t_M$ and each trace variable can be instantiated w.r.t. its quantifier. We now proceed as follows.

- **Mapping Function.** The trace $t_M$ is of the form $t_M = s^r$ where $s \in S_M$ is a unique state. The mapping function $m$ is defined based on the label of $s$. That is, for all $(a, b) \in L(s)$, $m(a) = b$.

- **Mapping Composition.** For a quantifier $Q_i = \forall$, each trace variable $\pi_{A_i}$ and $\pi_{B_i}$ is instantiated by all possible pairs of traces $t_{A_i} \in \text{Traces}(K_A)$ and $t_{B_i} \in \text{Traces}(K_B)$, respectively. The formula $\varphi_{\text{map}}$ checks if each pair $\langle t_{A_i}, t_{B_i} \rangle$ fulfills the mapping. By having $\varphi_{\text{map}}$ be followed by an implication, we only enforce the pairs which fulfill the mapping $m$, to satisfy the rest of the formula. Hence, each pair that satisfies $\varphi_{\text{map}}$ corresponds to the projection of each $t_i \in \text{Traces}(K_A \mid_m K_B)$.

- **Hyperproperty.** Since the $(K_M, K_A, K_B) = \Phi_{ABM}$, we have, $\langle t_{A_1}, t_{B_1}, \ldots, t_{A_n}, t_{B_n} \rangle = \psi_{\text{new}}$. By composing each pair of traces $\langle t_{A_i}, t_{B_i} \rangle$ as a composed trace $t_i$ in the traces of $(K_A \mid_m K_B)$, we have $\langle t_1, t_2, \ldots, t_n \rangle \models \psi$.

V. **APPLICATION: SECURE REFINEMENT**

It is well-known that traditional notions of refinement (e.g., ones based on trace inclusion [34]) do not preserve certain types of security properties, such as non-interference [40]. Alternative definitions of refinement have been proposed to ensure that desired security properties of an abstract system are preserved by its implementations [4], [37], [39]. As an application, we describe how our mapping synthesis technique can be used to synthesize refinement mappings that preserve security properties that are specified as hyper-properties.

### A. Secrecy Property

We adopt the notion of secrecy-preserving refinement proposed by Alur et al. [4], as it is general enough to capture a wide range of security properties, including noninterference. In [4] the authors prove that their notion of refinement preserves what they call secrecy properties, but only hint at the possibility of automated techniques for verifying that a concrete program is a secrecy-preserving refinement of an abstract program. We take one step further and show how our approach can be used to synthesize such a refinement mapping.

Consider Kripke structure $K = \langle S, S^0, \delta, \text{AP}, L \rangle$ and trace property $P \subseteq \Sigma^\omega$. Some subset of the atomic propositions are designated to be observable: $\text{AP}^{\text{obs}} \subseteq \text{AP}$. Conceptually, these propositions represent parts of the state that are visible to the environment (e.g., an attacker), while the rest remain hidden (e.g., internal actions). In addition, given a set $T \subseteq \Sigma^\omega$ of traces, let $\approx \subseteq T \times T$ be an equivalence relation on pairs of traces with respect to the propositions in $\text{AP}^{\text{obs}}$.

**Definition 4**: Given a trace property $P$ and a set of traces $T$, the secrecy property, $\text{Sec}(T, P, \approx)$, is defined as follows:

$$\text{Sec}(T, P, \approx) \equiv \forall t \in T. \exists t' \in T. (t \in P) \rightarrow (t \approx t' \land t' \not\in P)$$

Then, we say that $K$ satisfies the secrecy of $P$ for given $\approx$ if and only if $K$ satisfies $\text{Sec}(K, P, \approx)$. Intuitively, $P$ states that sensitive information is to be protected: Given an execution of the system represented by trace $t$, we wish to prevent the attacker from knowing whether or not $t \in P$. One way to protect the secret, as stated in Definition 4 is to introduce uncertainty into the attacker’s knowledge by allowing an additional trace $t'$ that is observationally equivalent to $t$ but does not satisfy $P$. Note that secrecy as defined above is a hyperproperty.

Hyperproperty $\text{Sec}(T, P, \approx)$ can be instantiated with different equivalence relations ($\approx$). Let us define a notion of strong equivalence relation $\approx_w$ between a pair of traces, $t, t' \in T$ as follows. Let $E : \Sigma \rightarrow \Sigma$ be an erasing function that hides all non-observable propositions from an element of a trace; i.e., given $e \in \Sigma$, $E(e) = e'$, where $e'$ is derived as $e \cap \text{AP}^{\text{obs}}$. If the latter results in an empty set ($e \cap \text{AP}^{\text{obs}} = \emptyset$), special symbol $\tau$ is assigned as $e'$; conceptually, $\tau$ represents internal states that are hidden from an observer. With abuse of notation, we define $E$ over traces as well, where $E(t)$ returns the trace that results from applying $E$ to every element in trace $t$. Then, we say that a pair of traces $t$ and $t'$ are strongly equivalent (i.e., $t \approx_w t'$) if and only if $E(t) = E(t')$; i.e., the traces match each other in their observable parts.

Let us define another notion of equivalence, called weak equivalence relation $\approx_w$. Let $E_w$ be similar to the erasing function $E$ introduced above, except it removes all $\tau$ from given trace $t$; e.g., $E_w(e) = \varepsilon$ if $L(e) \cap \text{AP}^{\text{obs}} = \emptyset$ (we assume that $\tau$ is treated as an non-existent element and removed from traces; i.e., $t = e_0\varepsilon e_1 e_2 = e_0\varepsilon_1 e_2$). Then, a pair of traces $t$ and $t'$ are weakly equivalent (i.e., $t \approx_w t'$) if and only if $E_w(t) = E_w(t')$. This relation is also called a time-insensitive equivalence relation, as it considers a pair of traces to be equivalent even if they do not agree on the number of internal computational steps.

As we will illustrate in Section VI, an implementation that preserves secrecy under one notion of observational equivalence (e.g., $\approx_w$) might not do so under a different notion (e.g., $\approx_s$). By allowing $\approx$ to be provided as a parameter, our technique can be used to synthesize a refinement mapping (if it exists) under varying notions of observational equivalence.

### B. Secrecy-Preserving Refinement Mapping

Consider a pair of Kripke structures $K_{sp} = \langle S_{sp}, S^0_{sp}, \delta_{sp}, \text{AP}_{sp}, L_{sp} \rangle$ and $K_{imp} = \langle S_{imp}, S^0_{imp}, \delta_{imp}, \text{AP}_{imp}, L_{imp} \rangle$, representing an abstract specification and its concrete implementation, respectively. A refinement mapping $m_r : S_{imp} \rightarrow S_{sp}$ is a function that relates the states of the implementation to those of the abstract specification [1]. With abuse of notation, we define $m_r$ to be applicable over traces as well, where $m_r(t_{imp})$ returns the result of applying $m_r$ to every state in
t_{imp} \in T_{imp} \text{ (i.e., } m_r(t_{imp}) \in T_{sp}). \text{ We say that } m_r \text{ is a valid refinement mapping if and only if it satisfies the following two conditions:}

- **Simulation condition:** The abstract specification simulates the implementation; i.e., every behavior of } K_{imp} \text{ is also a behavior of } K_{sp}:
  \forall c, c' \in S_{imp}, \forall a \in S_{sp}, a = m_r(c) \land (c, c') \in \delta_{imp} \rightarrow \\
  \exists a' \in S_{sp}, (a, a') \in \delta_{sp} \land a' = m_r(c')

- **Secrecy-preserving:** Given } K_{sp} \models S_{Sec}(T_{sp}, P, \approx), \text{ the implementation does not leak the secrecy of } P:
  \forall t \in T_{imp}, m_r(t) \in P \rightarrow \\
  \exists t' \in T_{imp}, t \approx t' \land m_r(t') \notin P

For the problem of refinement mapping synthesis that we consider in Sections V and VI, we assume that the labeling functions in the Kripke structures (i.e., } L_{imp} \text{ and } L_{sp}) \text{ are injective. We believe that this is a reasonable assumption to make, as program states can be defined uniquely by assignments to state variables (which are represented by propositions).

C. Formulation as Mapping Synthesis with HyperLTL

We describe how the task of finding a secrecy-preserving refinement mapping can be formulated as an instance of the mapping synthesis problem. One complication is that our notion of mapping is between propositions, whereas a refinement mapping is defined over states. As a workaround, a given Kripke structure is first flattened into another structure where each state is assigned exactly one proposition that encodes the set of all propositions in the original state.

**Flattened Kripke Structure.** Given } K = \langle S, S^0, \delta, \text{AP}, L \rangle, \text{ let } \text{flat}(K) = \langle \hat{S}, \hat{S}_0, \delta, \text{AP}, L \rangle \text{ where:}

- \hat{S} = \{ \hat{s} \mid \exists s \in S. \tilde{s} = f(s) \}
- \hat{S}_0 = \{ \hat{s} \mid \exists s \in S_0. \tilde{s} = f(s) \}
- \hat{\delta} = \{ (\hat{s}_a, \hat{s}_b) \mid (f^{-1}(\hat{s}_a), f^{-1}(\hat{s}_b)) \in \delta \}
- \text{AP} = ^0\text{AP}
- \hat{L}(\hat{s}) = \{ L(f^{-1}(\hat{s})) \} \text{ for each } \hat{s} \in \hat{S}

where } f : S \to \hat{S} \text{ is a bijection between the states of } K \text{ and its flattening. For example, for state } s \in S \text{ with } L(s) = \{ a, b \}, \text{ its corresponding state } \tilde{s} = f(s) \text{ in the flattened structure is assigned label } p_s \text{ where } \hat{L}(\hat{s}) = \{ p_s \} \text{ and } p_s = \{ a, b \}.

**Refinement Mappings.** Given } K_{sp} \text{ and } K_{imp}, \text{ consider a pair of Kripke structures } K_A = \langle S_A, S^A_0, \delta_A, \text{AP}_A, L_A \rangle \text{ and } K_B = \langle S_B, S^B_0, \delta_B, \text{AP}_B, L_B \rangle \text{ such that } K_A = \text{flat}(K_{imp}) \text{ and } K_B = \text{flat}(K_{sp}). \text{ Then, a mapping between } K_A \text{ and } K_B, \text{ } m : \text{AP}_A \to \text{AP}_B, \text{ can be defined in terms of } m_r \text{ as follows:}

\[ m = \{ (p_{s_{imp}}, p_{s_{sp}}) \mid (L_{imp}^{-1}(p_{s_{imp}}), L_{sp}^{-1}(p_{s_{sp}})) \in m_r \} \]

For convenience, we also define function } R_m, \text{ which maps } m \text{ to every element in given path } \pi = s_0s_1s_2 \ldots s_n \text{ to obtain another path } \pi' = s_0s_1s'_2 \ldots s'_n; \text{ i.e.,}

\[ R_m(\pi) = \pi' \leftrightarrow \text{for each } s \in \pi, m(L(s)) = L(s') \land s' \in \pi' \]

Note that here the label of each state refers to the one proposition (in the flattened structure) that encodes the original set of atomic propositions.

**Simulation Condition.** We impose simulation conditions as part of } \varphi_{imp} \text{ as we introduce in Section III: Given two flattened structures } K_A \text{ and } K_B, \text{ if } (L_A^{-1}(p_{s_{imp}}), L_B^{-1}(p_{s_{sp}})) \in \delta_A \text{ and } (p_{s_{imp}}, p_{s_{sp}}) \in m, \text{ then } (p_{s_{imp}}, p_{s_{sp}}) \in m \text{ if and only if}

\[ (L_B^{-1}(p_{s_{sp}}), L_B^{-1}(p_{s_{sp}}')) \in \delta_B. \]

**Observational Equivalence.** The above two notions of observational equivalence can be encoded in HyperLTL:

- For weak equivalence } (\approx_w), \text{ we define formula } eq_w:
  \[ eq_w(\pi, \pi') \equiv \bigwedge_{p \in \text{AP}^{obs}} \square (p_{\pi} \leftrightarrow p_{\pi'}) \]
  That is, the observable variables always match in their values in a given pair of traces.

- For strong equivalence } (\approx_s), \text{ in addition to the observable variables, we also consider the internal steps } (\tau) \text{ that do not contain any observable variables. This can be implicitly done by only comparing the observable variables, because } p_{\pi} \leftrightarrow p_{\pi'} \text{ is vacuously true when a state does not contain any observable variables. In addition, we abuse the notation here and use } PC \text{ to represent the encoding of the program counter for a path variable } \pi. \text{ The value of } PC \text{ advances on each state transition of a trace and the domain of } PC \text{ is finite. We add } PC \text{ to each path variable } \pi \text{ and } \pi' \text{ for strong equivalence to ensure that the given traces advance in a lockstep with each other:}
  \[ eq_s(\pi, \pi') \equiv \bigwedge_{p \in \text{AP}^{obs}} \square (p_{\pi} \leftrightarrow p_{\pi'}) \land (PC_{\pi} = PC_{\pi'}) \]

**Secrecy Property.** Given trace property } P, \text{ we write } P(\pi) \text{ to denote that some concrete trace } t \text{ assigned to } \pi \text{ satisfies } P. \text{ Then, the secrecy property from [4] in Definition 4 can be encoded in HyperLTL } \varphi_{sec} \text{ as follows:}

\[ \varphi_{sec} = \forall \pi, \exists \pi'. P(\pi) \to (eq(\pi, \pi') \land \neg P(\pi')) \]

where } eq \text{ can be } eq_w \text{ or } eq_s.

**Secrecy Preservation.** Given } K_A = \text{flat}(K_{imp}) \text{ and } K_B = \text{flat}(K_{sp}), \text{ let us assume that } K_B \models \varphi_{sec}. \text{ Given mapping } m, \text{ Kripke structure } K_A \text{ is said to preserve the secrecy for } P \text{ if and only if the following HyperLTL formula holds over } K_A \parallel_m K_B:

\[ \varphi_{sp} = \forall \pi_A, \exists \pi'_A. P(R_m(\pi_A)) \rightarrow (eq(\pi_A, \pi'_A) \land \neg P(\pi'_A)) \]

Finally, finding a valid refinement mapping that preserves the secret } P \text{ between } K_{sp} \text{ and } K_{imp} \text{ amounts to solving the following mapping synthesis problem:}

**Secrecy-Preserving Refinement Mapping Synthesis.** Given Kripke structures } K_{sp} \text{ and } K_{imp}, \text{ trace property } P, \text{ and equivalence relation } \approx \text{ such that } K_B \models S_{Sec}(T_{sp}, P, \approx), \text{ let } K_A \text{ and } K_B \text{ be the flattening of } K_{imp} \text{ and } K_{sp}, \text{ respectively. Find a mapping } m : \text{AP}_A \to \text{AP}_B \text{ (if it exists) such that } (K_A \parallel_m K_B) \models \varphi_{sp}. \]
Once we have a solution mapping \( m : AP_A \rightarrow AP_B \), it is straightforward to convert it into its state-based equivalence \( m_r : S_{imp} \rightarrow S_{sp} \):
\[
m_r = \{(s_{imp}, s_{sp}) \mid (L_{imp}(s_{imp}), L_{sp}(s_{sp})) \in m \}
\]
If there exists at least one refinement mapping, it demonstrates that \( K_{imp} \) may be used as an implementation that preserves the secrecy in the abstract machine \( K_{sp} \). On the other hand, if no such mapping exists, \( K_{imp} \) cannot be used as a secure implementation for \( K_{sp} \).

Proposition 1: Given Kripke structures \( K_{sp} \) and \( K_{imp} \), trace property \( P \), equivalence relation \( \approx \), let \( m \) be a solution to the above mapping synthesis problem. Then, its state-based equivalence \( m_r \), is a valid secrecy-preserving refinement mapping from \( K_{imp} \) to \( K_{sp} \).

Remarks on the direction of mapping. It is worth noting that the direction of refinement mapping \( m_r \) is from concrete machine \( K_{imp} \) to abstract machine \( K_{sp} \), in contrast to the examples introduced in Sections I to III. In these earlier examples, a mapping is used to represent a concretization function, which decides how abstract propositions in \( K_A \) are realized as their concrete counterparts in \( K_B \). Under a valid mapping \( m \), the resulting machine, \( K_A \parallel m \mid m_B \), preserves a desired property but may not necessarily be a behavioral refinement of \( K_A \). On the other hand, a refinement mapping in this section is used to encode an abstraction function and is imposed additional constraints (i.e., simulation conditions) to ensure that \( K_{imp} \) is a behavioral refinement of \( K_{sp} \).

VI. Case Studies and Evaluation

In this section, we provide an experimental evaluation of our approach to synthesize secrecy-preserving refinement mappings (as described in Section V) on two different case studies: the battleship game and a password checker. The experimental results from the two case studies show that our technique is able to synthesize secrecy-preserving refinement mappings under different notions of observational equivalence, and also scales to reasonably sized programs.

Our prototype implementation is built on top of the HyperLTL bounded model checker (BMC) HyperQube [35]. The model checker performs two tasks: (1) generation of a QBF formula that describes the model and the hyperproperty of interest (called genQBF), and (2) invocation of the QBF-solver QuAbs [46] to solve the satisfiability problem for the QBF query. All experiments in this Section were run on a MacBook with Intel i7 CPU @2.8 GHz and 16 GB of RAM.

A. Case Study 1: Battleship Game

Given an \( n \times n \) grid, two players can place an equal number of ships onto the grid by marking the cells that correspond to their \((i, j)\) coordinates. In the beginning of the game, the positions of each player’s fleet of ships are concealed from each other. The players will then take turns guessing the position of one of the opponent’s ships; if the guess is correct, the corresponding ship is removed from the game. The game ends when one of the players has their entire fleet removed from the game.

The secret that we wish to protect in this game is the occupancy of a specific row (i.e., whether or not a row is empty) to maintain fairness between the players. For example, suppose that one of the players guesses position \((3, 3)\) and receives a miss. If one is able to infer that Row 3 is empty, then in the next round, the player can avoid guessing any position on row 3 knowing that this will result in a miss. Hence, any implementation of the battleship game must protect the secrecy of this information.

Algorithm 1 describes the specification \( K_{sp} \) that uses a 2-dimensional array to represent the positions of the ships. Given \( i \) and \( j \) as input, the program returns in one atomic step a hit if the \((i, j)\) position is marked, and a miss otherwise.

The program \( K_{imp} \), shown in Algorithm 2, is a concrete implementation of \( K_{sp} \) that uses a set of lists to store the information about ship positions. Given input coordinates \( i \) and \( j \), \( K_{imp} \) first obtains the list for the \( i \)-th row, and then uses the helper function isEmpty() and returns a miss immediately when this row contains no ships. Otherwise, the program continues on and checks whether the \( j \)-th position in this row is marked, returning a hit if so.

```
Algorithm 1: Battleship Game Spec (Ksp)

Input: (i,j)
Output: \{hit,miss\}
1 boolean Board(n x n);
2 if Board[i][j] then
3 r ← hit;
4 else
5 r ← miss;
6 end
7 output ← r;
8 return output;
```

```
Algorithm 2: Battleship Game Impl (Kimp)

Input: (i,j)
Output: \{hit,miss\}
1 row = Board.getRow(i);
2 if row.isEmpty() then
3 r ← miss;
4 output ← r;
5 return output;
6 else
7 if row.get(j) then
8 r ← hit;
9 else
10 r ← miss;
11 end
12 end
13 output ← r;
14 return output;
```

In our setting, regardless of its size, every board is guaranteed to have some empty rows, i.e., rows with no ship. Each program has a special Boolean value \( rowEmpty \) that will be set to true when the input \( i \) is pointing to an empty row, and to false otherwise. We then define trace property \( P_{row} \) (which represents the secret being protected):
\[
P_{row} \equiv \Diamond (rowEmpty)
\]
i.e., eventually the row guessed by the opponent is empty.

To find a refinement mapping \( m_r : S_{imp} \rightarrow S_{sp} \) such that simulation condition and secrecy preservation are both

\footnote{Our code and case studies are available at https://bit.ly/3aJwKbf.}
fully, we proceed as follows. The simulation condition can be imposed as part of the mapping constrains as mentioned in Section V, i.e., allow only mappings such that each trace in $K_{imp}$ has an abstract counterpart in $K_{sp}$. A HyperLTL property encoding the preservation of secrecy $P_{row}$ is defined as:

$$\varphi_{sp} \equiv \forall \pi. \exists \pi'. P_{row}(R_m(\pi)) \rightarrow (eq(\pi, \pi') \wedge \neg P_{row}(\pi'))$$

In addition, we assume that the attacker can only observe the final output of the programs (i.e., $AP_{obs} = \{Output\}$).

**Synthesized refinement mapping:** Given $K_{imp}$, $K_{sp}$, $\varphi_{sp}$, and time-insensitive (weak) equivalence (i.e., $eq \equiv eq_w$ in $\varphi_{sp}$), our tool was able to synthesize a valid refinement mapping. The synthesized mapping guarantees both functional equivalence (i.e., given the same input, both machines should produce the same output) and preserves the secrecy of $P_{row}$. Informally, $m$ maps the state of $K_{imp}$ that corresponds to an early return of miss (line 5 in Algorithm 2) to the last state in $K_{sp}$ that ends with Output = miss (line 8 in Algorithm 1). Since the attacker is insensitive to the number of computation steps, it is unable to infer any additional information in $K_{imp}$.

However, under time-sensitive (strong) equivalence (i.e., $eq \equiv eq_s$ in $\varphi_{sp}$), the tool is unable to synthesize a valid refinement mapping. Intuitively, this is because for each trace that ends on line 5 in Algorithm 2, it is impossible to find another observationally equivalent trace that takes the same amount of time but differs on the satisfaction of $P_{row}$. Thus, the concrete program $K_{imp}$ cannot be considered a secure implementation if the attacker is capable of observing the length of program executions.

**Analysis of results:** Table I shows the results from synthesis runs for varying sizes of the game grid. While genQBF can generate the QBF formulas quickly, QuAbS takes longer to return an answer. This difference is due to the nondeterministic nature of the program: Since the opponent can guess any possible position on the board in each round, the input $(i, j)$ is nondeterministic. Also, $K_{sp}$ and $K_{imp}$ are loop-free programs constructed using only if-else statements, implying that the depth of the model is constant for any grid size. In our implementation, the coordinates $i, j$ are in bitwise representation (i.e., for a grid of size $= 40^2$, we need at least 6 bits + 6 bits for both $i$ and $j$). Thus, even when we increase the grid size drastically, the size of AP in the model do not increase in the same magnitude, and the time it takes to generate and solve the QBF queries also remains relatively small. However, we need to check through all the possible values of the input to ensure secrecy property, when the grid size increases, more enumeration needs to be done by the QBF solver QuAbS, affecting the solving time.

**B. Case Study 2: Password Checker**

Our second case study is a **password checker** program that compares an input from a client against a password stored on a web server. Modern web applications rely on the strength of passwords to ensure security; for example, a user may be required to create a password that contains a given object. Then, $P_{size}$ states that the size of user input is

| grid size | $|S|$ | genQBF [s] | QuAbS [s] | Total [s] |
|-----------|------|------------|-----------|-----------|
| $3^2$     | $13^9$ | 1.07       | 0.23      | 1.30      |
| $10^2$    | $13^100$ | 1.03 | 0.58 | 1.61 |
| $20^2$    | $13^400$ | 1.25 | 2.25 | 3.50 |
| $40^2$    | $13^1600$ | 1.54 | 12.73 | 24.62 |
| $60^2$    | $13^3600$ | 1.79 | 16.81 | 28.82 |
| $80^2$    | $13^6400$ | 3.36 | 116.25 | 130.51 |
| $100^2$   | $13^10000$ | 4.51 | 165.82 | 177.57 |

**Table I:** Synthesis times for the battleship game problem. BMC with bound $k = 7$ terminates for all grid sizes.

The basic functionality of a password checker is to return “yes” when the user’s input matches the secret password and “no” otherwise. We investigate two programs, the specification $K_{sp}$ and and implementation $K_{imp}$. Consider $K_{sp}$ in Algorithm 3. It initializes a Boolean value matches as true and iterates through every character of the user input. If there is any mismatch between the input and the password, matches is set to false. Algorithm 4 shows a candidate implementation, $K_{imp}$. In the beginning of the program (lines 1-2), it first compares the lengths of the input and the password, and returns false immediately if the lengths do not match. This appears to be a reasonable implementation, as if the input differs from the password in length, the two cannot possibly match.

We define a trace property $P_{size}$, which represents the secret to be preserved (i.e., the size of password). We first introduce a helper function $size()$, which returns the length of a given object. Then, $P_{size}$ states that the size of user input is
TABLE II: Synthesis times for the password checking problem.

<table>
<thead>
<tr>
<th>length of secret</th>
<th>genQBF [s]</th>
<th>QuAbS [s]</th>
<th>Total [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>49 × 3²</td>
<td>8.06</td>
<td>11.58</td>
</tr>
<tr>
<td>5</td>
<td>49 × 5²</td>
<td>13.78</td>
<td>4.61</td>
</tr>
<tr>
<td>7</td>
<td>49 × 7²</td>
<td>19.45</td>
<td>5.36</td>
</tr>
<tr>
<td>10</td>
<td>49 × 10²</td>
<td>33.91</td>
<td>9.82</td>
</tr>
<tr>
<td>13</td>
<td>49 × 13²</td>
<td>49.23</td>
<td>8.41</td>
</tr>
<tr>
<td>15</td>
<td>49 × 15²</td>
<td>60.19</td>
<td>10.19</td>
</tr>
<tr>
<td>17</td>
<td>49 × 17²</td>
<td>79.83</td>
<td>12.51</td>
</tr>
<tr>
<td>20</td>
<td>49 × 20²</td>
<td>111.35</td>
<td>15.86</td>
</tr>
</tbody>
</table>

TABLE VII. RELATED WORK

The closest work to the study in this paper is the work in [38], where the authors study the problem of mapping synthesis for trace properties. We generalize [38] to hyper-properties that capture complex information-flow requirements, and we also provide a reduction to HyperLTL model checking. Also, the work in [38] focuses on asynchronous event-based systems, while in this paper we present a synchronous state-based version of mapping synthesis which we believe is more elegant.

Another related line of work is efforts on program synthesis for hyperproperties. The *program repair* problem for HyperLTL has been studied in [10]. The repair problem is to find a subset of traces of a Kripke structure that satisfies a given HyperLTL formula. The authors in [10] study the complexity of program for different fragments of HyperLTL. The repair problem is similar to the *controller synthesis* problem studied in [11]. In both problems, the goal is to prune the set of transitions of the given plant or model. However, in program repair, all transitions are controllable, whereas in controller synthesis the pruning cannot be applied to uncontrollable transitions. The complexity of the controller synthesis problem for HyperLTL was studied in [11]. Directly related to the controller synthesis problem studied in this paper is the *satisfiability*. The satisfiability problem for HyperLTL was shown to be decidable for the \(3E\) fragment and for any fragment that includes a \(\exists\) quantifier alternation [19]. The hierarchy of hyperlogics beyond HyperLTL has been studied in [17]. The general *synthesis* problem differs from controller synthesis in that the solutions are not limited to the state graph of the plant. For HyperLTL, synthesis was shown to be undecidable in general, and decidable for the \(3E\) fragment [21]. While the synthesis problem becomes, in general, undecidable as soon as there are two universal quantifiers, there is a special class of universal specifications, called the linear \(\forall^*\) fragment, which is still decidable. The linear \(\forall^*\)-fragment corresponds to the decidable *distributed synthesis* problems [28]. The *bounded synthesis* problem [21], [18] considers only systems up to a given bound on the number of states. Bounded synthesis has been successfully applied to various benchmarks including the dining cryptographers [14].

There has been a lot of recent progress in automatically verifying [27], [26], [25], [18] and monitoring [2], [24], [13], [12], [23], [45], [32] HyperLTL specifications. HyperLTL is also supported by a growing set of tools, including the model checker MCHyper [27], [18], the bounded model checker HyperQube [35], the satisfiability checkers EAHyper [22] and MGHyper [20], and the runtime monitoring tool RVHyper [23]. The problem of *model checking* hyperproperties for tree-shaped and acyclic graphs was studied in [9].

Refinement for information-flow properties such as non-interference has been investigated extensively in the literature [4], [7], [6], [33], [36], [39], [29], [41], [42], [43]. These works typically prescribe manual or semi-automated methods (based on proof assistants such as Coq [8]) for verifying refinement. In comparison, by formulating refinement as a synthesis problem, our ultimate goal is to provide automated support for security refinement. Our approach is also intended to be general: Although our case studies focused on the notion of secrecy from [4], our technique can be used to synthesize a mapping for any property that is expressible in HyperLTL.
VIII. Conclusion and Future Work

In this paper, we have proposed an approach for automatically synthesizing a mapping between two models that satisfies hyperproperties, and demonstrated its utility on a variety of problems, including mapping high-level designs to low-level execution platforms, enforcing properties such as non-interference, and secrecy-preserving refinement mappings. We showed how the mapping synthesis problem can be solved by reducing it to bounded model checking for the logic HyperLTL. We also reported on a prototype implementation and demonstrated two case studies.

Currently, our technique is used to synthesize a refinement mapping to one particular given implementation. A promising future direction is to build a platform model that encodes a set of possible implementations (e.g., by adopting the notion of holes in program sketching [44]) and leverage our technique to synthesize an implementation that preserves a desired security property. We also plan to extend our synthesis technique for properties that are beyond the expressiveness of HyperLTL, such as hyperproperties that involve stuttering equivalence [31].

References


