Mapping Synthesis for Hyperproperties

Tzu-Han Hsu[1], Borzoo Bonakdarpour[1], Eunsuk Kang[2], and Stavros Tripakis[3]

[1]Department of Computer Science and Engineering, Michigan State University, USA
{tzuhan,borzoo}@msu.edu
[2]School of Computer Science, Carnegie Mellon University, USA
eskang@cmu.edu
[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, $\mathtt{a} := \mathtt{b} + \mathtt{c}$ in a high-level programming language is typically mapped to two steps during translation to an intermediate language: (1) $\mathtt{reg} := \mathtt{b} + \mathtt{c}$, and (2) $\mathtt{a} := \mathtt{reg}$, where 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* [37], [39].

In a prior work [38], 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, [38] also proposed a technique for automatically synthesizing a mapping that establishes a *trace-based property* [3] in the resulting weaving of the models. However, trace properties are not expressive enough to encode a wide range of security properties [16], including non-interference, and thus the approach of [38] falls short of being able to synthesize mappings to satisfy such properties.

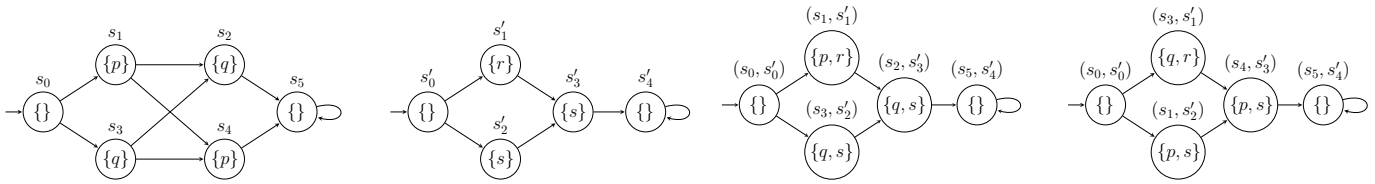
In this paper, we study the problem of mapping synthesis in the context of *hyperproperties* [16] expressed in the temporal logic HyperLTL. Let us first explain the problem by using the following high-level toy example. Consider the transition system (*Kripke structure*) K_A shown in Fig. 1a and the following HyperLTL formula describing a requirement for K_A:

$$\varphi = \forall\pi.\ \exists\pi'.\ \square(p_\pi \rightarrow \neg p_{\pi'})$$

This formula stipulates that for any trace π, there exists a trace π', such that it is always the case that if proposition p holds in some position of π, then p should not hold in the same position of trace π'. Observe that $K_A \models \varphi$, because for any arbitrary trace π in K_A (e.g., trace $t_1 = \{\}\{p\}\{p\}\{\}^\omega$ produced by path $s_0 s_1 s_4 s_5^\omega$), there exists another trace π', such that when p is true in π, p is false in π' (e.g., trace $t_2 = \{\}\{q\}\{q\}\{\}^\omega$ produced by path $s_0 s_3 s_2 s_5^\omega$).

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.



(a) Kripke structure K_A. (b) Kripke structure K_B. (c) Valid mapping. (d) Invalid mapping.

Fig. 1: Illustration of the mapping synthesis problem for HyperLTL formula $\varphi = \forall\pi.\exists\pi'.\square(p_\pi \rightarrow \neg p_{\pi'})$.

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 φ. 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 $K_A \parallel_m K_B$. For instance, if we apply the following mapping function:

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

we obtain the composed Kripke structure $K_A \parallel_m K_B$ 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 $K_A \parallel_m K_B$, which allows φ to be preserved under mapping m. Thus, m is considered to be a *valid* mapping. Now, consider another mapping function m', where:

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

Mapping m' results in the composed Kripke structure $K_A \parallel_{m'} K_B$ shown in Fig. 1d. Here, trace t_1 in K_A is manifested in the composed structure as $t''_1 = \{\}\{p,s\}\{p,s\}\{\}^\omega$, for which there exists no trace that satisfies formula φ. 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 φ, 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 φ 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 π_M is analogous to synthesizing a function m in the original mapping synthesis problem. Here, Ξ is formula that encodes the original formula φ 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.[1] 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.

[1] Our 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 $\Sigma = 2^{\mathsf{AP}}$ be the *alphabet*, where 2^X denotes the powerset (set of all subsets) of a set X. A *letter* is an element of Σ. A *trace* $t \in \Sigma^\omega$ over alphabet Σ 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 $K = \langle S, S^0, \delta, \mathsf{AP}, L\rangle$, 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
- $L : S \to 2^{\mathsf{AP}}$ 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 $0 \le i < n$, 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 $i \ge 0$. 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 $i \ge 0$. We denote by $\mathsf{Traces}(K, s)$ the set of all traces of K with paths that start in state $s \in S$, and use $\mathsf{Traces}(K)$ as a shorthand for $\bigcup_{s\in S_0} \mathsf{Traces}(K, s)$.

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:

$$\varphi ::= \exists \pi.\varphi \mid \forall \pi.\varphi \mid \phi$$
$$\phi ::= p_\pi \mid \neg\phi \mid \phi \vee \phi \mid \bigcirc \phi \mid \phi \,\mathcal{U}\, \phi$$

where $p \in \mathsf{AP}$ is an atomic proposition and π is a *trace variable* from an infinite supply of variables $\mathcal{V}$. The Boolean connectives $\neg$ and $\vee$ have the usual meaning, $\mathcal{U}$ is the temporal *until* operator, and $\bigcirc$ is the temporal *next* operator. We also consider syntactic sugar $\mathsf{true} \equiv p_\pi \vee \neg p_\pi$, $\mathsf{false} \equiv \neg\mathsf{true}$, $\varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2)$, and $\varphi_1 \to \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$. Also, the derived temporal operators include *eventually* $\Diamond\varphi \equiv \mathsf{true}\,\mathcal{U}\,\varphi$ and *globally* $\Box\varphi \equiv \neg\Diamond\neg\varphi$. The quantified formulas $\exists\pi$ and $\forall\pi$ are read as "along some trace π" and "along all traces π", 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 $\mathit{Vars}(\varphi)$ for the set of trace variables used in formula φ.

2) Semantics: An *interpretation* $\mathcal{T} = \langle T_\pi\rangle_{\pi\in \mathit{Vars}(\varphi)}$ of a formula φ consists of a tuple of sets of traces, one set T_π per trace variable π in $\mathit{Vars}(\varphi)$. We use T_π for the set of traces assigned to π. 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 π can range over, T_π, comes from a specific K_π, i.e., $T_\pi = \mathsf{Traces}(K_\pi)$. We use $\mathcal{K} = \langle K_\pi\rangle_{\pi\in \mathit{Vars}(\varphi)}$ to denote a *family* of Kripke structures. To simplify, we write $\mathcal{T} = \mathsf{Traces}(\mathcal{K})$ 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 $\Pi\colon \mathit{Vars}(\varphi) \rightharpoonup \Sigma^\omega$. The assignment with an empty domain is denoted by $\Pi_\emptyset$. Given a trace assignment Π, a trace variable π, and a trace $t \in \Sigma^\omega$, we denote by $\Pi[\pi \to t]$ the assignment that coincides with Π everywhere except at π, which in $\Pi[\pi \to t]$ is mapped to trace t. The satisfaction of a HyperLTL formula φ is a binary relation $\models$ that associates a formula to the models $(\mathcal{T}, \Pi, i)$ where $i \in \mathbb{Z}_{\ge 0}$ 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 φ, denoted by $\mathcal{T} \models \varphi$, if $(\mathcal{T}, \Pi_\emptyset, 0) \models \varphi$. We say that a family of Kripke structures $\mathcal{K}$ satisfies a specification φ, denoted by $\mathcal{K} \models \varphi$, if it holds that $\mathsf{Traces}(\mathcal{K}) \models \varphi$.

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

$$
\begin{array}{lll}
(\mathcal{T},\Pi,0) \models \exists\pi.\ \psi & \text{iff} & \text{there is a } t \in T_\pi, \text{ such that } (\mathcal{T},\Pi[\pi \to t],0) \models \psi,\\
(\mathcal{T},\Pi,0) \models \forall\pi.\ \psi & \text{iff} & \text{for all } t \in T_\pi, \text{ such that } (\mathcal{T},\Pi[\pi \to t],0) \models \psi,\\
(\mathcal{T},\Pi,i) \models a_\pi & \text{iff} & a \in \Pi(\pi)(i),\\
(\mathcal{T},\Pi,i) \models \neg\psi & \text{iff} & (\mathcal{T},\Pi,i) \not\models \psi,\\
(\mathcal{T},\Pi,i) \models \psi_1 \vee \psi_2 & \text{iff} & (\mathcal{T},\Pi,i) \models \psi_1 \text{ or } (\mathcal{T},\Pi,i) \models \psi_2,\\
(\mathcal{T},\Pi,i) \models \bigcirc\psi & \text{iff} & (\mathcal{T},\Pi,i+1) \models \psi,\\
(\mathcal{T},\Pi,i) \models \psi_1\,\mathcal{U}\,\psi_2 & \text{iff} & \text{there is a } j \geq i \text{ for which } (\mathcal{T},\Pi,j) \models \psi_2 \text{ and}\\
 & & \quad \text{for all } k \in [i,j), (\mathcal{T},\Pi,k) \models \psi_1,
\end{array}
$$

Fig. 2: Semantics of HyperLTL.

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 from.to.*content*. For example, Alice.Bob.sec indicates that Alice sends the secret to Bob, and Bob.Eve.pub means that 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:

$$
\begin{aligned}
\varphi_{\mathsf{NI}} = &\forall\pi_1.\exists\pi_2.(sec_{\pi_1} \not\leftrightarrow sec_{\pi_2}) \wedge\\
&\square\,(\mathsf{Eve\_}secNonEmpty_{\pi_1} \leftrightarrow \mathsf{Eve\_}secNonEmpty_{\pi_2})
\end{aligned}
$$

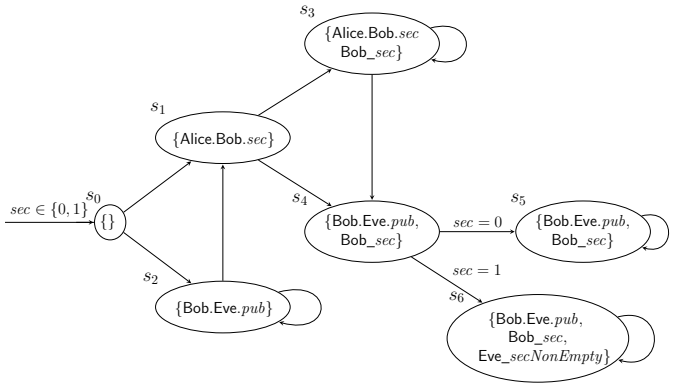


Fig. 3: Kripke structure K_A

In our example, K_A violates the non-interference property, that is, $K_A \not\models \varphi_{\mathsf{NI}}$. 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 φ_{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 $K_A \not\models \varphi_{\mathsf{NI}}$.

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, it also results in a simpler definition of mapping composition as compared to [38].

Definition 2: Let $K_A = \langle S_A, S_A^0, \delta_A, \mathsf{AP}_A, L_A\rangle$ and $K_B = \langle S_B, S_B^0, \delta_B, \mathsf{AP}_B, L_B\rangle$ be two Kripke structures such that $\mathsf{AP}_A \cap \mathsf{AP}_B = \emptyset$. A *mapping* is a partial function $m : \mathsf{AP}_A \rightharpoonup \mathsf{AP}_B$. A pair of states $(s_A, s_B) \in S_A \times S_B$ satisfies the *mapping condition*, denoted $\mathsf{MapCond}(s_A, s_B)$, iff the following holds:

$$\forall a \in L(s_A).\forall b \in \mathsf{AP}_B.\big(m(a) = b\big) \;\rightarrow\; \big(b \in L(s_B)\big).$$

■

That is, if a is an atomic proposition that holds in state s_A, and a is mapped by m to atomic proposition b, then atomic proposition b must hold in state s_B.

Definition 3: The *mapping composition* of two Kripke structures $K_A = \langle S_A, S_A^0, \delta_A, \mathsf{AP}_A, L_A\rangle$ and $K_B = \langle S_B, S_B^0, \delta_B, \mathsf{AP}_B, L_B\rangle$ w.r.t. mapping m is defined as a Kripke structure $(K_A \parallel_m K_B) = \langle S, S^0, \delta, \mathsf{AP}, L\rangle$, where

- $S = \{(s_A, s_B) \in S_A \times S_B \mid \mathsf{MapCond}(s_A, s_B)\}$
- $S^0 = S \cap (S_A^0 \times S_B^0)$
- $\delta = \{\big((s_A, s_B), (s'_A, s'_B)\big) \in S \times S \mid (s_A, s'_A) \in \delta_A \wedge (s_B, s'_B) \in \delta_B\}$
- $\mathsf{AP} = \mathsf{AP}_A \cup \mathsf{AP}_B$
- $L : S \to 2^{\mathsf{AP}}$, where $L((s_A, s_B)) = L(s_A) \cup L(s_B)$

■

That is, a state in the composite system $(K_A \parallel_m K_B)$ is a product state (s_A, s_B) satisfying the mapping condition. An initial state of $(K_A \parallel_m K_B)$ is a pair of initial states of K_A and K_B such that the pair again satisfies the mapping condition. A transition of $(K_A \parallel_m K_B)$ consists in both K_A and K_B moving synchronously. The set of atomic propositions of $(K_A \parallel_m K_B)$ contains all atomic propositions of K_A plus those of K_B. And the set of atomic propositions holding at a product state (s_A, s_B) contains all propositions holding at s_A plus all those holding at s_B. Note that the mapping condition guarantees that if a holds in state s_A and a maps to b, then b holds at s_B, which implies that both a and b will hold at the product state (s_A, s_B). This is important because if a is mapped to b and satisfaction of a property in K_A depends on a, this satisfaction should carry to the mapped proposition, which is b.

We can now define the mapping synthesis problem for HyperLTL:

The HyperLTL mapping synthesis problem. Given two Kripke structures K_A and K_B, and a HyperLTL formula φ, find, if there exists, a *valid* mapping, i.e., a mapping m such that $(K_A \parallel_m K_B) \models \varphi$.

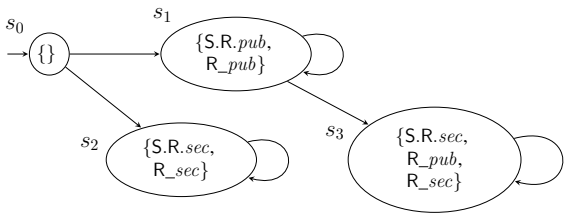


Fig. 4: A Kripke structure K_B

In the above definition of mapping synthesis, the search space of possible mappings m is left unconstrained. This is done for reasons of simplicity, so that the problem definition does not become overly complicated. In our approach and implementation, we allow the user to specify additional constraints. For example, the user may require the mapping function to be injective, surjective, or bijective. The user may also disallow the empty mapping (i.e., the function defined nowhere), which is a-priori a possible candidate since mappings are partial functions. Finally, the user may also restrict the possible mappings of certain atomic propositions. For instance, the user may specify that only a subset of AP_A is to be mapped, or that a certain proposition in AP_A can only be mapped to some of the propositions in AP_B.

Also, notice that the statement of the problem does not require that $K_A \models \varphi$, i.e., K_A does not have to necessarily satisfy hyperproperty φ. This is desirable in order to be able to model scenarios like our running example, where the goal is to enforce φ during mapping. Note that other properties than φ that K_A may satisfy are not necessarily preserved by mapping. We, thus, emphasize that any other property, say φ', that the designer requires the system to satisfy after mapping has to be explicitly included as a part of φ, e.g., by taking the conjunction $\varphi \wedge \varphi'$.

Example: Let Kripke structure K_B in Fig. 4 represents a communication platform between a sender S and a receiver R for the abstract model K_A in Fig. 3 and the non-interference policy φ_{NI} discussed in Section II-C. The platform only allows S to either (1) send a secret message to R (i.e. $s_0 {s_2}^\omega$), or (2) send R a public message first then followed by a secret messages (i.e. $s_0 s_1 s_1 {s_3}^\omega$). Our goal is to find a mapping m, such that $(K_A \parallel_m K_B) \models \varphi_{\mathsf{NI}}$. In the following section, we provide a solution to this problem and we illustrate on this example some valid and invalid mappings.

Synthesis of invalid mappings: In addition to synthesizing valid mappings, our technique can also be used to synthesize invalid mappings, i.e., mappings that satisfy $\neg\varphi$, i.e., mappings that violate φ. We can achieve this simply by negating φ in the definition of the mapping synthesis problem. Why would we ever want to synthesize invalid mappings? This could be useful, for example, for debugging purposes. If no invalid mapping exists, this might indicate a trivially satisfied specification φ. There are also other reasons why we might want to synthesize invalid mappings, e.g., in order to exhibit some bad (insecure) implementation choices, that we might want to be certain to avoid. Overall, our method is versatile enough and can be used both for φ and $\neg\varphi$.

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 φ. The goal is to find (if there exists) a mapping m such that $(K_A \parallel_m K_B)$ satisfies φ. In our reduction, we will construct a new Kripke structure K_M which represents the set of all candidate mappings.[2] We will also construct a new HyperLTL formula $\Phi_{\mathsf{ABM}} = \exists \pi_M . \Xi$, which encodes the existence of a good mapping (the quantified trace variable π_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, Ξ encodes the original formula φ adapted to the verification problem. Then, finding a valid mapping m is reduced to checking whether the family $\langle K_M, K_A, K_B \rangle$ satisfies Φ_{ABM}. We now present this method in detail.

1) The Kripke structure K_M: Consider two Kripke structures $K_A = \langle S_A, S_A^0, \delta_A, \mathsf{AP}_A, L_A \rangle$ and $K_B = \langle S_B, S_B^0, \delta_B, \mathsf{AP}_B, L_B \rangle$. We denote by $\mathsf{Mappings}(\mathsf{AP}_A, \mathsf{AP}_B)$ 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 $|\mathsf{Mappings}(\mathsf{AP}_A, \mathsf{AP}_B)| = (|\mathsf{AP}_B| + 1)^{|\mathsf{AP}_A|}$ possible mappings.

We construct Kripke structure K_M as follows:

- $S_M = \{s_m \mid m \in \mathsf{Mappings}(\mathsf{AP}_A, \mathsf{AP}_B)\}$
- $S_M^0 = S_M$
- $\delta_M = \{(s, s) \mid s \in S_M\}$
- $\mathsf{AP}_M = \mathsf{AP}_A \times \mathsf{AP}_B$
- $L_M : S_M \to 2^{\mathsf{AP}_M}$ s.t. for any $s_m \in S_M$, we have $L(s_m) = \{(a, b) \in \mathsf{AP}_M \mid m(a) = b\}$.

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.

[2]This set is typically very large, and therefore the state space of K_M is also large. However, fortunately, we can represent K_M symbolically and not explicitly, which allows us to combat state explosion.

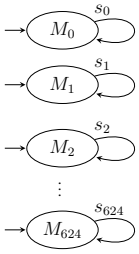


Fig. 5: The Kripke structure K_M derived from K_A and K_B of Figs. 3 and 4.

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: $|\mathsf{AP}_A| = |\mathsf{AP}_B| = 4$. 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:

$$
\begin{aligned}
M_0 &= \{\} \\
M_1 &= \{(\mathsf{Alice.Bob}.sec, \mathsf{S.R}.pub)\} \\
M_2 &= \{(\mathsf{Bob.Eve}.pub, \mathsf{S.R}.pub)\} \\
M_3 &= \{(\mathsf{Alice.Bob}.sec, \mathsf{S.R}.pub), \\
&\qquad (\mathsf{Bob.Eve}.pub, \mathsf{S.R}.pub)\}
\end{aligned}
$$

and so on, up to

$$
\begin{aligned}
M_{624} = \{&(\mathsf{Alice.Bob}.sec, \mathsf{S.R}.pub), \\
&(\mathsf{Bob.Eve}.pub, \mathsf{S.R}.sec), \\
&(\mathsf{Bob}\_sec, \mathsf{R}\_sec), \\
&(\mathsf{Eve}\_secNonEmpty, \mathsf{R}\_sec)\}.
\end{aligned}
$$

2) The Global HyperLTL Formula Φ_{ABM}: Let the original HyperLTL formula φ be of the form: $\mathbb{Q}_1 \pi_1 . \mathbb{Q}_2 \pi_2 \ldots \mathbb{Q}_n \pi_n . \psi$ where ψ is quantifier-free and each $\mathbb{Q}_i \in \{\forall, \exists\}$, for $1 \leq i \leq n$. Then, Φ_{ABM} is constructed as follows:

$$
\begin{aligned}
\Phi_{\mathsf{ABM}} &\stackrel{\text{def}}{=} \exists \pi_M \, . \, \Xi \\
\Xi &\stackrel{\text{def}}{=} \mathbb{Q}_1 \pi_{A_1} . \mathbb{Q}_1 \pi_{B_1} . \ldots . \mathbb{Q}_n \pi_{A_n} . \mathbb{Q}_n \pi_{B_n} . \Psi \\
\Psi &\stackrel{\text{def}}{=} \varphi_{map_1} \circ_1 \varphi_{map_2} \circ_2 \ldots \varphi_{map_n} \circ_n \psi_{new}
\end{aligned}
$$

where π_M ranges over K_M, trace variables π_{A_i} range over K_A, while π_{B_i} range over K_B, for $1 \leq i \leq n$. Also, $\circ_i = \wedge$ if $\mathbb{Q}_i = \exists$ and $\circ_i = \rightarrow$ if $\mathbb{Q}_i = \forall$, where $\circ_i$ has precedence over all the preceeding such operators. The subformulas φ_{map_i} and ψ_{new} are defined below. Intuitively, existence of a trace of K_M such that Ξ is satisfied encodes existence of a valid mapping. In φ, each trace variabe π_i ranges over the traces of $K_A \parallel_m K_B$. In Φ_{ABM}, we replace π_i by two trace variables π_{A_i} and π_{B_i} ranging over K_A and K_B, respectively. These variables together with φ_{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 φ_{map_i} and ψ_{new}. First, we define, for $1 \leq i \leq n$:

$$\varphi_{map_i} \stackrel{\text{def}}{=} \bigwedge_{(a,b)\in \mathsf{AP}_M} \square\Big((a,b)_{\pi_M} \;\rightarrow\; (a_{\pi_{A_i}} \rightarrow b_{\pi_{B_i}})\Big)$$

Intuitively, φ_{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 ψ_{new} to be the subformula obtained by replacing each $a_{\pi_i} \in \mathsf{AP}_A$ in ψ 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 φ_{NI}. Our goal is to find a mapping m, such that $(K_A \parallel_m K_B) \models \varphi_{\mathsf{NI}}$, by reducing it to a HyperLTL verification problem of checking whether the family $\langle K_M, K_A, K_B\rangle$ satisfies the constructed global formula w.r.t. φ_{NI}, called $\Phi_{\mathsf{NI-ABM}}$. We now use this running example to demonstrate the construction of $\Phi_{\mathsf{NI-ABM}}$ and how the satisfiability of $\Phi_{\mathsf{NI-ABM}}$ gives us a valid mapping that enforces K_A to satisfy the property φ_{NI}.

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

$$\begin{aligned}\varphi_{\mathsf{NI}} =& \forall\pi_1.\exists\pi_2.\ \psi_{\mathsf{NI}}\\ \psi_{\mathsf{NI}} =& (sec_{\pi_1} \not\leftrightarrow sec_{\pi_2}) \wedge\\ & \square\ (\mathsf{Eve\_}secNonEmpty_{\pi_1} \leftrightarrow \mathsf{Eve\_}secNonEmpty_{\pi_2}),\end{aligned}$$

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

$$\Phi_{\mathsf{NI-ABM}} = \exists\pi_M.\forall\pi_{A_1}.\forall\pi_{B_1}.\exists\pi_{A_2}.\exists\pi_{B_2}.\ (\varphi_{map_1} \rightarrow (\varphi_{map_2} \wedge \psi_{\mathsf{NI}-new}))$$

We next present in detail how the elements φ_{map_1}, φ_{map_2}, and $\psi_{\mathsf{NI}-new}$ are constructed.

First, the mapping subformulas φ_{map_1} and φ_{map_2} are as follows:

$$\varphi_{map_1} = \bigwedge_{(a,b)\in \mathsf{AP}_M} \square\Big((a,b)_{\pi_M} \;\rightarrow\; (a_{\pi_{A_1}} \rightarrow b_{\pi_{B_1}})\Big)$$

$$\varphi_{map_2} = \bigwedge_{(a,b)\in \mathsf{AP}_M} \square\Big((a,b)_{\pi_M} \;\rightarrow\; (a_{\pi_{A_2}} \rightarrow b_{\pi_{B_2}})\Big)$$

The purpose of φ_{map_1} and φ_{map_2} is to make sure that the labels of traces fulfill the mapping constraints. That is, for all $a \in \mathsf{AP}_A$ and $b \in \mathsf{AP}_B$, 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 $\mathsf{Alice.Bob}.sec_{\pi_{A_i}}$ to $\mathsf{S.R}.sec_{\pi_{B_i}}$, then φ_{map_1} and φ_{map_2} will impose $(\mathsf{Alice.Bob}.sec_{\pi_{A_1}} \rightarrow \mathsf{S.R}.sec_{\pi_{B_1}})$ and $(\mathsf{Alice.Bob}.sec_{\pi_{A_2}} \rightarrow \mathsf{S.R}.sec_{\pi_{B_2}})$, respectively.

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

$$\begin{aligned}\psi_{\mathsf{NI}-new} =& (sec_{\pi_{A_1}} \not\leftrightarrow sec_{\pi_{A_2}}) \wedge\\ & \square\ (\mathsf{Eve\_}secNonEmpty_{\pi_{A_1}} \leftrightarrow \mathsf{Eve\_}secNonEmpty_{\pi_{A_2}})\end{aligned}$$

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

$$\langle K_M, K_A, K_B\rangle \models \Phi_{\mathsf{NI-ABM}},$$

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

$$\begin{aligned}M_{valid} = \{&(\mathsf{Alice.Bob}.sec,\ \mathsf{S.R}.sec),\\ &(\mathsf{Bob.Eve}.pub,\ \mathsf{S.R}.pub)\}.\end{aligned}$$

Let us evaluate M_{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 φ_{NI}. However, if we map $\mathsf{Alice.Bob}.sec$ to $\mathsf{S.R}.sec$, and $\mathsf{Bob.Eve}.pub$ to $\mathsf{S.R}.pub$, as prescribed by M_{valid}, then the path $s_0s_1s_4s_6$ is eliminated after mapping composition. This is because the corresponding trace on K_B, i.e. $\mathsf{S.R}.sec$ followed by $\mathsf{S.R}.pub$, does not exist. Hence, the path that violates φ_{NI} is now eliminated, resulting in M_{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 φ_{NI} to obtain:

$$\neg\varphi_{\mathsf{NI}} = \exists\pi_1.\forall\pi_2.\ \neg\psi_{\mathsf{NI}}$$

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

$$\Phi_{\text{not}-\mathsf{NI}-\mathsf{ABM}} = \exists\pi_M.\exists\pi_{A_1}.\exists\pi_{B_1}.\forall\pi_{A_2}.\forall\pi_{B_2}.\ (\varphi_{map_1} \wedge (\varphi_{map_2} \rightarrow \neg\psi_{\mathsf{NI}-new}))$$

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

$$\begin{aligned}M_{invalid} = \{&(\mathsf{Alice.Bob}.sec,\ \mathsf{S.R}.pub),\\ &(\mathsf{Bob.Eve}.pub,\ \mathsf{S.R}.sec)\}\end{aligned}$$

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 $M_{invalid}$ 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 φ of the form $\mathbb{Q}_1\pi_1.\mathbb{Q}_2\pi_2 \ldots \mathbb{Q}_n\pi_n.\psi$ and Kripke structures K_A and K_B, there exists a mapping m such that $(K_A \parallel_m K_B) \models \varphi$ iff $\langle K_M, K_A, K_B\rangle \models \Phi_{\mathsf{ABM}}$, according to the above construction.

Proof:

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

For the base case where $n = 1$, we have $\varphi = \mathbb{Q}_1\pi_1.\psi$. Then, Φ_{ABM} will be of the form $\exists\pi_M.\mathbb{Q}_1\pi_{A_1}.\mathbb{Q}_1\pi_{B_1}.\ \varphi_{map_1} \circ_1 \psi_{new}$, where $\mathbb{Q}_1 \in \{\forall, \exists\}$. We first instantiate the outermost quantifier $\exists\pi_M$ of Φ_{ABM}. There exists a unique state $\mathfrak{s} \in K_M$ such that the label of $\mathfrak{s}$ corresponds to mapping m, that is, for all $(a, b) \in L(\mathfrak{s})$ we have $m(a) = b$. In K_M, by construction, we instantiate π_M with $t_M = \mathfrak{s}^\omega$. We now distinguish two cases. If $\mathbb{Q}_1$ = $\exists$, then there exists a trace $t_1 \in (K_A||_m K_B)$, such that

$$\Big(\mathsf{Traces}(K_A \parallel_m K_B), \Pi[\pi_1 \to t_1],\ 0\Big) \models \psi.$$

For existential quantifier, $\circ_1$ is $\wedge$, meaning that t_M, t_{A_1}, and t_{B_1} must satisfy $\varphi_{map_1} \wedge \psi_{new}$. The projection of t_1 on K_A and K_B derives two traces t_{A_1} and t_{B_1}, respectively. Since $t_1 \in \mathsf{Traces}(K_A \parallel_m K_B)$, it implies that traces t_{A_1} and t_{B_1} are obtained according to mapping function m. This in turn ensures that $\langle t_M, t_{A_1}, t_{B_1}\rangle \models \varphi_{map_1}$. Because for each $a \in \mathsf{AP}_A$ and $b \in \mathsf{AP}_B$, if $m(a) = b$, then φ_{map_1} is satisfiable only if $(a_{\pi_{A_1}} \to b_{\pi_{B_1}})$ always holds. That is, the instantiated π_{A_1} and π_{B_1} fulfills the mapping function m. From here, t_{A_1} and t_{B_1} are sufficient to instantiate π_{A_1} and π_{B_1} to satisfy φ_{map_1}. Since t_1 satisfies ψ, so t_{A_1} also satisfies ψ_{new}. That is,

$$\Big(\mathsf{Traces}(K_M, K_A, K_B), \Pi[\pi_M \to t_M, \pi_{A_1} \to t_{A_1}, \pi_{B_1} \to t_{B_1}], 0\Big) \models \varphi_{map_1} \wedge \psi_{new}$$

If $\mathbb{Q}_1$ = $\forall$, we have $\circ_1$ is $\to$, meaning that t_M, t_{A_1}, and t_{B_1} must satisfy $\varphi_{map_1} \to \psi_{new}$. The segment $\mathbb{Q}_1\pi_{A_1}.\mathbb{Q}_1\pi_{B_1}$ represents all paths on the composed model of K_A and K_B. Take an arbitrary pair of traces $t_{A_1} \in \mathsf{Traces}(K_A)$ and $t_{B_1} \in \mathsf{Traces}(K_B)$. If $\langle t_M, t_{A_1}, t_{B_1}\rangle \not\models \varphi_{map_1}$, then $\varphi_{map_1} \to \psi_{new}$ is vacuously true and, hence, the HyperLTL verification problem is consequently true as well. Now, consider the case where $\langle t_M, t_{A_1}, t_{B_1}\rangle \models \varphi_{map_1}$. 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:

$$\Big(\mathsf{Traces}(K_A \parallel_m K_B), \Pi[\pi_1 \to t_1],\ 0\Big) \models \psi,$$

because π_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 φ_{map_1} and ψ_{new}. Hence, the following always holds

$$\Big(\mathsf{Traces}(K_M, K_A, K_B), \Pi[\pi_M \to t_M, \pi_{A_1} \to t_{A_1}, \pi_{B_1} \to t_{B_1}], 0\Big) \models \varphi_{map_1} \to \psi_{new}$$

For inductive step, the inductive hypothesis is as follows. Suppose there is a mapping function $m : \mathsf{AP}_A \rightharpoonup \mathsf{AP}_B$ that fulfills $(K_A \parallel_m K_B) \models \varphi$, where $\varphi = \mathbb{Q}_1\pi_1.\mathbb{Q}_2\pi_2 \ldots \mathbb{Q}_k\pi_k.\psi$, it is true that $\langle K_M, K_A, K_B\rangle \models \Phi_{\mathsf{ABM}}$ for any $k \geq 1$. We now want to show $k + 1$ holds.

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

$$\Big(\mathsf{Traces}(K_A||_m K_B), \Pi[\pi_1 \to t_1, \pi_2 \to t_2, \vdots\ , \pi_k \to t_k], 0\Big) \models \psi,$$

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 π_{A_i} and π_{B_i}, respectively, as follows:

$$\Big(\mathsf{Traces}(K_M, K_A, K_B), \Pi[\pi_M \to t_M, \pi_{A_1} \to t_{A_1}, \pi_{B_1} \to t_{B_1}, \pi_{A_2} \to t_{A_2}, \pi_{B_2} \to t_{B_2}, \vdots\ , \pi_{A_k} \to t_{A_k}, \pi_{B_k} \to t_{B_k}], 0\Big) \models (\varphi_{map_1} \circ_1 \varphi_{map_2} \circ_2 \ldots \varphi_{map_k} \circ_k \psi_{new})$$

From here, for $\mathbb{Q}_{k+1}$, 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 $\mathbb{Q}_{k+1}$, where $\mathbb{Q}_{k+1} \in \{\exists, \forall\}$. Hence, after instantiating each trace variable in the HyperLTL formula, the following must hold:

$$\Big(\mathsf{Traces}(K_M, K_A, K_B), \Pi[\pi_M \to t_M, \pi_{A_1} \to t_{A_1}, \pi_{B_1} \to t_{B_1}, \pi_{A_2} \to t_{A_2}, \pi_{B_2} \to t_{B_2}, \vdots\ , \pi_{A_k} \to t_{A_k}, \pi_{B_k} \to t_{B_k}, \pi_{A_{k+1}} \to t_{A_{k+1}}, \pi_{B_{k+1}} \to t_{B_{k+1}}], 0\Big) \models (\varphi_{map_1} \circ_1 \varphi_{map_2} \circ_2 \ldots \varphi_{map_k} \circ_k \varphi_{map_{k+1}} \circ_{k+1} \psi_{new})$$

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

($\Leftarrow$) We now prove the reverse direction. Suppose $(K_M, K_A, K_B) \models \Phi_{\mathsf{ABM}}$. Then π_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 a trace of the form $t_M = \mathfrak{s}^\omega$ where $\mathfrak{s} \in S_M$ is a unique state. The mapping function m is defined based on the label of $\mathfrak{s}$. That is, for all $(a, b) \in L(\mathfrak{s})$, $m(a) = b$.
- **Mapping Composition.** For a quantifier $\mathbb{Q}_i = \forall$, each trace variable π_{A_i} and π_{B_i} is instantiated by all possible pair of traces $t_{A_i} \in \mathsf{Traces}(K_A)$ and $t_{B_i} \in \mathsf{Traces}(K_B)$, respectively. The formula φ_{map_i} checks if each pair $\langle t_{A_i}, t_{B_i} \rangle$ fulfills the mapping. By having φ_{map_i} be followed by an implication, we only enforce the pairs which fulfill the mapping m, to satisfies the rest of the formula. Hence, each pair that satisfies φ_{map_i} corresponds to the projection of each $t_i \in \mathsf{Traces}(K_A \parallel_m K_B)$.
  For a quantifier $\mathbb{Q}_i = \exists$, π_{A_i} and π_{B_i} are instantiated by some t_{A_i} and t_{B_i}. Since $\circ_i = \wedge$, then t_{A_i} and t_{B_i} must also be a pair that satisfies φ_{map_i}. Thus, t_{A_i} and t_{B_i} are sufficient to compose t_i as a witness of the existential quantifier.
- **Hyperproperty.** Since the $\langle K_M, K_A, K_B \rangle \models \Phi_{\mathsf{ABM}}$, we have, $\langle t_{A_1}, t_{B_1}, \ldots, t_{A_n}, t_{B_n} \rangle \models \psi_{new}$. By composing each pair of traces $\langle t_{A_i}, t_{B_i} \rangle$ as a composed trace t_i in the traces of $(K_A \parallel_m K_B)$, we have $\langle t_1, t_2, \ldots, t_n \rangle$, such that $\langle t_1, t_2, \ldots, t_n \rangle \models \psi$.

■

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 $K = \langle S, S^0, \delta, \mathsf{AP}, L \rangle$ and trace property $P \subseteq \Sigma^\omega$. Some subset of the atomic propositions are designated to be *observable*: $\mathsf{AP}^{\mathsf{obs}} \subseteq \mathsf{AP}$. 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 $\mathsf{AP}^{\mathsf{obs}}$.

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

$$Sec(T, P, \approx) \equiv \forall t \in T. \exists t' \in T. (t \in P) \rightarrow (t \approx t' \wedge t' \notin P)$$

■

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

Let us define another notion of equivalence, called *weak equivalence relation* $\approx_{\mathbf{w}}$. Let $\mathcal{E}_{\mathbf{w}}$ be similar to the erasing function $\mathcal{E}$ introduced above, except it removes all τ from given trace t; e.g., $\mathcal{E}_{\mathbf{w}}(e) = \epsilon$ if $L(e) \cap \mathsf{AP}^{\mathsf{obs}} = \emptyset$ (we assume that ϵ is treated as an non-existent element and removed from traces; i.e., $t = e_0 \epsilon e_1 \epsilon e_2 = e_0 e_1 e_2$). Then, a pair of traces t and t' are *weakly equivalent* (i.e., $t \approx_{\mathbf{w}} t'$) if and only if $\mathcal{E}_{\mathbf{w}}(t) = \mathcal{E}_{\mathbf{w}}(t')$. 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., $\approx_{\mathbf{w}}$) might not do so under a different notion (e.g., $\approx_{\mathbf{s}}$). 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 $K_{sp} = \langle S_{sp}, S^0_{sp}, \delta_{sp}, \mathsf{AP}_{sp}, L_{sp} \rangle$ and $K_{imp} = \langle S_{imp}, S^0_{imp}, \delta_{imp}, \mathsf{AP}_{imp}, L_{imp} \rangle$, 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

$t_{imp} \in T_{imp}$ (i.e., $m_r(t_{imp}) \in T_{sp}$). We say that m_r 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} is also a behavior of K_{sp}:

$$\forall c, c' \in S_{imp}.\forall a \in S_{sp}.a = m_r(c) \wedge (c, c') \in \delta_{imp} \rightarrow \exists a' \in S_{sp}.(a, a') \in \delta_{sp} \wedge a' = m_r(c')$$

- **Secrecy-preserving**: Given $K_{sp} \models Sec(T_{sp}, P, \approx)$, the implementation does not leak the secrecy of P:

$$\forall t \in T_{imp}.m_r(t) \in P \rightarrow \exists t' \in T_{imp}.t \approx t' \wedge m_r(t') \notin P$$

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} and L_{sp}) 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* $K = \langle S, S^0, \delta, \mathsf{AP}, L\rangle$, *let* $\mathsf{flat}(K) = \langle \hat{S}, \hat{S}_0, \hat{\delta}, \hat{\mathsf{AP}}, \hat{L}\rangle$ where:

- $\hat{S} = \{\hat{s} \mid \exists s \in S.\hat{s} = f(s)\}$
- $\hat{S}_0 = \{\hat{s} \mid \exists s \in S_0.\hat{s} = f(s)\}$
- $\hat{\delta} = \{(\hat{s}_a, \hat{s}_b) \mid (f^{-1}(\hat{s}_a), f^{-1}(\hat{s}_b)) \in \delta\}$
- $\hat{\mathsf{AP}} = 2^{\mathsf{AP}}$
- $\hat{L}(\hat{s}) = \{L(f^{-1}(\hat{s}))\}$ for each $\hat{s} \in \hat{S}$

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

Refinement Mappings. Given K_{sp} and K_{imp}, consider a pair of Kripke structures $K_A = \langle S_A, S^0_A, \delta_A, \mathsf{AP}_A, L_A\rangle$ and $K_B = \langle S_B, S^0_B, \delta_B, \mathsf{AP}_B, L_B\rangle$ such that $K_A = \mathsf{flat}(K_{imp})$ and $K_B = \mathsf{flat}(K_{sp})$. Then, a mapping between K_A and K_B, $m : \mathsf{AP}_A \rightharpoonup \mathsf{AP}_B$, can be defined in terms of m_r as follows:

$$m = \{(p_{s_A}, p_{s_B}) \mid (L^{-1}_{imp}(p_{s_A}), L^{-1}_{sp}(p_{s_B})) \in m_r\}$$

For convenience, we also define function R_m, which applies m to every element in given path $\pi = s_0 s_1 s_2 \ldots s_n$ to obtain another path $\pi' = s'_0 s'_1 s'_2 \ldots s'_n$; i.e.,

$$R_m(\pi) = \pi' \leftrightarrow \text{for each } s \text{ in } \pi, m(L(s)) = L(s') \wedge s' \in \pi'$$

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 φ_{map} as we introduce in Section III: Given two flattened structures K_A and K_B, if $(L^{-1}_A(p_{s_A}), L^{-1}_A(p'_{s_A})) \in \delta_A$ and $(p_{s_A}, p_{s_B}) \in m$, then $(p'_{s_A}, p'_{s_B}) \in m$ if and only if $(L^{-1}_B(p_{s_B}), L^{-1}_B(p'_{s_B})) \in \delta_B$.

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

- For weak equivalence ($\approx_{\mathbf{w}}$), we define formula $eq_{\mathbf{w}}$:

$$eq_{\mathbf{w}}(\pi, \pi') \equiv \bigwedge_{p \in \mathsf{AP}^{\mathsf{obs}}} \square\,(p_\pi \leftrightarrow p_{\pi'})$$

  That is, the observable variables always match in their values in a given pair of traces

- For strong equivalence ($\approx_{\mathbf{s}}$), in addition to the observable variables, we also consider the internal steps (τ) that do not contain any observable variables. This can be implicitly done by only comparing the observable variables, because $p_\pi \leftrightarrow p_{\pi'}$ is vacuously true when a state does not contain any observable variables. In addition, we abuse the notation here and use PC_π to represent the encoding of the program counter for a path variable π. The value of PC advances on each state transition of a trace and the domain of PC is finite. We add PC to each path variable π and π' for strong equivalence to ensure that the given traces advance in a lockstep with each other:

$$eq_{\mathbf{s}}(\pi, \pi') \equiv \bigwedge_{p \in \mathsf{AP}^{\mathsf{obs}}} \square\,(p_\pi \leftrightarrow p_{\pi'}) \wedge \square\,(PC_\pi = PC_{\pi'})$$

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

$$\varphi_{sec} = \forall\pi.\exists\pi'.\ P(\pi) \ \rightarrow\ \Big(eq(\pi, \pi') \wedge \neg P(\pi')\Big)$$

where eq can be $eq_{\mathbf{w}}$ or $eq_{\mathbf{s}}$.

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

$$\varphi_{\mathsf{sp}} = \forall\pi_A.\exists\pi'_A.\ P(R_m(\pi_A)) \ \rightarrow\ \Big(eq(\pi_A, \pi'_A) \wedge \neg P(\pi'_A)\Big)$$

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

Secrecy-Preserving Refinement Mapping Synthesis. Given Kripke structures K_{sp} and K_{imp}, trace property P, and equivalence relation $\approx$ such that $K_{sp} \models Sec(T_{sp}, P, \approx)$, let K_A and K_B be the flattening of K_{imp} and K_{sp}, respectively. Find a mapping $m : \mathsf{AP}_A \rightharpoonup \mathsf{AP}_B$ (if it exists) such that $(K_A \parallel_m K_B) \models \varphi_{sp}$.

Once we have a solution mapping $m : \mathsf{AP}_A \rightharpoonup \mathsf{AP}_B$, it is straightforward to convert it into its state-based equivalence $m_r : S_{imp} \to S_{sp}$:

$$m_r = \{(s_{imp}, s_{sp}) \mid (L_{imp}(s_{imp}), L_{sp}(s_{sp})) \in m\}$$

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, $K_A \parallel_m K_B$, 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].[3] 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.

[3]Our code and case studies are available at https://bit.ly/3aJwKbf.

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 (K_{sp})

```
   Input: (i, j)
   Output: {hit, miss}
1  boolean Board(n × 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 (K_{imp})

```
   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):

$$P_{row} \equiv \Diamond(rowEmpty_\pi)$$

i.e., eventually the row guessed by the opponent is empty.

To find a refinement mapping $m_r : S_{imp} \to S_{sp}$ such that *simulation condition* and *secrecy preservation* are both

fulfilled, 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:

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

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

Synthesized refinement mapping: Given K_{imp}, K_{sp}, φ_{sp}, and time-insensitive (weak) equivalence (i.e., $eq \equiv eq_{\mathbf{w}}$ in φ_{sp}), our tool was able to synthesis 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 in Algorithm 2) to the last state in K_{sp} that ends with $\mathsf{Output} = \mathsf{miss}$ (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., $eq \equiv eq_{\mathbf{s}}$ in φ_{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 bit-wise 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 some minimum combination of digits, symbols, and uppercase letters. Although these constraints can make it difficult for an attacker to guess the correct password efficiently, there are other ways in which this secrete could be leaked—also called *side-channel attacks*. In particular, the secrecy of interest in this case study is the *size* of a server-side stored password: If the password checker leaks information about the size, the attacker may be able to leverage this information and increase its chance of successfully guessing the correct password.

	HyperQube			
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.

Algorithm 3: Password checker spec (K_{sp})

```
   Input: int[] input
   Output: {true,false}
1  boolean matches = true;
2  for i ∈ {0, ..., input.length} do
3      if input[i] ≠ password[i] then
4          matches ← false
5      end
6  end
7  return matches
```

Algorithm 4: Password Checker Impl (K_{imp})

```
   Input: int[] input
   Output: {true,false}
1  if (input.length ≠ password.length) then
2      return false
3  end
4  for i ∈ {0, ..., input.length} do
5      if input[i] ≠ password[i] then
6          return false
7      end
8  end
9  return true
```

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

	HyperQube			
length of secret	$\|S\|$	genQBF [s]	QuAbS [s]	**Total [s]**
3	49×3^2	8.06	3.52	11.58
5	49×5^2	13.78	4.61	18.39
7	49×7^2	19.45	5.36	24.81
10	49×10^2	33.91	9.82	43.73
13	49×13^2	49.23	8.41	57.64
15	49×15^2	60.19	10.19	70.38
17	49×17^2	79.83	12.51	92.34
20	49×20^2	111.35	15.86	127.21

TABLE II: Synthesis times for the password checking problem. BMC with $k = 24$.

equal to the size of the stored password:

$$P_{size} \equiv \square \Big(input.\mathsf{size}() = pass.\mathsf{size}() \Big)_\pi$$

Hence, P_{size} is true if and only if on this trace, the length of $input$ and $pass$ are equal.

The secrecy-preserving property w.r.t. P_{size} is defined as:

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

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

Synthesized refinement mapping: Under time-insensitive equivalence ($eq \equiv eq_{\mathbf{w}}$), our tool was able to find a secrecy-preserving refinement mapping for the implementation K_{imp} of password checker program. The synthesized mapping achieves the simulation condition, by mapping the state of K_{imp} that corresponds to the early *false* return (line 2 in Algorithm 4) to the state of K_{sp} that returns with $\mathsf{matches} = \mathsf{false}$. Furthermore, if we disregard the number of computational steps, K_{imp} does not leak any more information than K_{sp} (including the size of the password) and thus, preserves the secrecy after refinement.

However, under time-sensitive equivalence ($eq_{\mathbf{s}}$), the tool is unable to find a valid refinement mapping. Intuitively, this is because the length of a trace in K_{imp} varies depending on whether the user input matches the password in length, while this is not the case in K_{sp}, resulting in leakage of secret P_{size}.

Analysis of results: The synthesis times (for varying lengths of password) are shown in Table II. A contrary trend can be seen here in comparison to the battleship game. Note that genqbf takes more time but QuAbs finds a solution relatively quickly. This is because the password checker programs contain loops and a larger transition system; as a result, the time taken by genqbf grows fast when we increase the possible length of user input and password. On the other hand, in our setting, the only source of non-determinism is in the size of the user input, not its content (e.g., the attacker may provide a series of strings containing only '0' characters, with the goal of determining the size of the password). Thus, the number of inputs for the QuAbs solver to explore remain relatively small even with the varying sizes of the password—hence, the relatively slow growth in the solving time.

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 hyperproperties 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 $\exists^*\forall^*$ fragment and for any fragment that includes a $\forall\exists$ 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 $\exists^*$ and $\exists^*\forall$ fragments [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

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.

[2] S. Agrawal and B. Bonakdarpour. Runtime verification of k-safety hyperproperties in HyperLTL. In *Proceedings of the IEEE 29th Computer Security Foundations (CSF)*, pages 239–252, 2016.

[3] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[4] R. Alur, P. Cerný, and S. Zdancewic. Preserving secrecy under refinement. In *Proceedings of the 33rd Automata, Languages and Programming, International Colloquium (ICALP)*, pages 107–118, 2006.

[5] C. Baier and J-.P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[6] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In *IEEE Computer Security Foundations Symposium (CSF)*, pages 328–343, 2018.

[7] C. Baumann, M. Dam, R. Guanciale, and H. Nemati. On compositional information flow aware refinement. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 79–94, 2021.

[8] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[9] B. Bonakdarpour and B. Finkbeiner. The complexity of monitoring hyperproperties. In *Proceedings of the 31st IEEE Computer Security Foundations Symposium (CSF)*, pages 162–174, 2018.

[10] B. Bonakdarpour and B. Finkbeiner. Program repair for hyperproperties. In *Proceedings of the 17th Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 423–441, 2019.

[11] B. Bonakdarpour and B. Finkbeiner. Controller synthesis for hyperproperties. In *Proceedings of the 33rd IEEE Computer Security Foundations Symposium (CSF)*, pages 366–379, 2020.

[12] B. Bonakdarpour, C. Sánchez, and G. Schneider. Monitoring hyperproperties by combining static analysis and runtime verification. In *Proceedings of the 8th Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 8–27, 2018.

[13] N. Brett, U. Siddique, and B. Bonakdarpour. Rewriting-based runtime verification for alternation-free HyperLTL. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 77–93, 2017.

[14] D. Chaum. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM*, 28(10):1030–1044, 1985.

[15] M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez. Temporal logics for hyperproperties. In *Proceedings of the 3rd Conference on Principles of Security and Trust POST*, pages 265–284, 2014.

[16] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[17] N. Coenen, B. Finkbeiner, C. Hahn, and J. Hofmann. The hierarchy of hyperlogics. In *Proceedings 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13, 2019.

[18] N. Coenen, B. Finkbeiner, C. Sánchez, and L. Tentrup. Verifying hyperliveness. In *Proceedings of the 31st International Conference on Computer Aided Verification (CAV)*, pages 121–139, 2019.

[19] B. Finkbeiner and C. Hahn. Deciding hyperproperties. In *Proceedings of the 27th International Conference on Concurrency Theory (CONCUR)*, pages 13:1–13:14, 2016.

[20] B. Finkbeiner, C. Hahn, and T. Hans. MGHyper: Checking satisfiability of HyperLTL formulas beyond the \exists ^*\forall ^* $\exists_* \forall_*$ fragment. In *Proceedings of the 16th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 521–527, 2018.

[21] B. Finkbeiner, C. Hahn, P. Lukert, M. Stenger, and L. Tentrup. Synthesis from hyperproperties. *Acta Informatica*, 57(1-2):137–163, 2020.

[22] B. Finkbeiner, C. Hahn, and M. Stenger. Eahyper: Satisfiability, implication, and equivalence checking of hyperproperties. In *Proceedings of the 29th International Conference on Computer Aided Verification (CAV)*, pages 564–570, 2017.

[23] B. Finkbeiner, C. Hahn, M. Stenger, and L. Tentrup. RVHyper: A runtime verification tool for temporal hyperproperties. In *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 194–200, 2018.

[24] B. Finkbeiner, C. Hahn, M. Stenger, and L. Tentrup. Monitoring hyperproperties. *Formal Methods in System Design (FMSD)*, 54(3):336–363, 2019.

[25] B. Finkbeiner, C. Hahn, and H. Torfah. Model checking quantitative hyperproperties. In *Proceedings of the 30th International Conference on Computer Aided Verification*, pages 144–163, 2018.

[26] B. Finkbeiner, Ch. Müller, H. Seidl, and E. Zalinescu. Verifying Security Policies in Multi-agent Workflows with Loops. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, 2017.

[27] B. Finkbeiner, M. N. Rabe, and C. Sánchez. Algorithms for model checking HyperLTL and HyperCTL*. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV)*, pages 30–48, 2015.

[28] B. Finkbeiner and S. Schewe. Uniform distributed synthesis. In *Proceedings of the 20th ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 321–330, 2005.

[29] J. G.-Cumming and J. W. Sanders. On the refinement of non-interference. In *Proceedings of the 4th IEEE Computer Security Foundations Workshop (CSFW)*, pages 35–42, 1991.

[30] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symp. on Security and Privacy*, pages 11–20, 1982.

[31] Jan Friso Groote and Frits W. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. In *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 443, pages 626–638, 1990.

[32] C. Hahn, M. Stenger, and L. Tentrup. Constraint-based monitoring of hyperproperties. In *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 115–131, 2019.

[33] Maritta Heisel, Andreas Pfitzmann, and Thomas Santen. Confidentiality-preserving refinement. In *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), 11-13 June 2001, Cape Breton, Nova Scotia, Canada*, pages 295–306. IEEE Computer Society, 2001.

[34] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[35] T.-H. Hsu, C. Sánchez, and B. Bonakdarpour. Bounded model checking for hyperproperties. In *Proceedings of the 27th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2021. To appear.

[36] J. Jacob. On the derivation of secure components. In *Proceedings of the IEEE Symposium on Security and Privacy (S & P)*, page 242–247, 1989.

[37] Jan Jürjens. Secrecy-preserving refinement. In *International Symposium of Formal Methods Europe (FME)*, pages 135–152, 2001.

[38] E. Kang, S. Lafortune, and S. Tripakis. Automated synthesis of secure platform mappings. In *Proceedings of the 31st International Conference Computer Aided Verification (CAV) Part I*, pages 219–237, 2019.

[39] H. Mantel. Preserving information flow properties under refinement. In *IEEE Symposium on Security and Privacy (S&P)*, pages 78–91, 2001.

[40] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symposium on Security and Privacy*, pages 79–93, April 1994.

[41] C. O'Halloran. A calculus of information flow. In *Proceedings of the First European Symposium on Research in Computer Security (ESORICS)*, pages 147–159, 1990.

[42] A. W. Roscoe, J. Woodcock, and Lars Wulf. Non-interference through determinism. *Journal of Computer Security*, 4(1):27–54, 1996.

[43] F. Seehusen and K. Stølen. Maintaining information flow security under refinement and transformation. In *Fourth International Workshop on Formal Aspects in Security and Trust (FAST)*, pages 143–157, 2006.

[44] A. Solar-Lezama. Program sketching. *Springer Journal on Software Tool for Technology Transfer STTT*, 15(5-6):475–495, 2013.

[45] S. Stucki, C. Sánchez, G. Schneider, and B. Bonakdarpour. Graybox monitoring of hyperproperties. In *Proceedings of the 23rd International Symposium on Formal Methods (FM)*, pages 406–424, 2019.

[46] L. Tentrup. CAQE and quabs: Abstraction based QBF solvers. *Journal of Satisfiability Boolean Modeling and Computation*, 11(1):155–210, 2019.