Symbolic Parallel Composition for Multi-language Protocol Verification

Faezeh Nasrabadi
CISPA Helmholtz Center for Information Security & Saarland University
faezeh.nasrabadi@cispa.de

Robert Künnemann
CISPA Helmholtz Center for Information Security
robert.kuennemann@cispa.de

Hamed Nemati
Department of Computer Science KTH Royal Institute of Technology
hnnemati@kth.se

***Abstract*—The implementation of security protocols often combines different languages. This practice, however, poses a challenge to traditional verification techniques, which typically assume a single-language environment and, therefore, are insufficient to handle challenges presented by the interplay of different languages. To address this issue, we establish principles for combining multiple programming languages operating on different atomic types using a symbolic execution semantics. This facilitates the (parallel) composition of labeled transition systems, improving the analysis of complex systems by streamlining communication between diverse programming languages. By treating the Dolev-Yao (DY) model as a symbolic abstraction, our approach eliminates the need for translation between different base types, such as bitstrings and DY terms. Our technique provides a foundation for securing interactions in multi-language environments, enhancing program verification and system analysis in complex, interconnected systems.**

***Index Terms*—distributed systems security, formal methods and verification, security protocols**

I. Introduction

In the rapidly evolving landscape of computer programming, it has become a norm that different programming languages coexist and interact within the same application. This is especially pronounced in complex systems like network protocols and operating systems, where components written in different languages must communicate seamlessly. Traditional approaches in program verification and system analysis often fall short in these multi-language environments, as they typically assume a homogeneous language framework. This assumption overlooks the challenges presented by the interplay of different programming languages, each with its unique syntax, semantics, and operational paradigms.

To overcome this limitation, we establish principles upon which two languages that operate on different atomic types can be combined. A typical use case is the analysis of network protocol implementations. At a minimum, they combine a party written in a real programming language, their communication partner(s) operating by specification and modeled abstractly, e.g., in the applied pi calculus, and an attacker, which is underspecified but usually limited by some threat model, e.g., the Dolev-Yao (DY) model or the cryptographic model of a time-bounded probabilistic Turing machine. As protocol properties extend over multiple parties in the presence of an attacker, an implementation-level analysis needs to reason

To this end, we extend *parallel asynchronous composition*, which combines two systems communicating with an unspecified 'outside' into a single interacting system. The state of art [1]–[5] describes the heterogeneous system as a composition of labeled transition systems (LTS). LTS are very flexible; they can abstract any programming language. Hence, the composition of LTS is the key to capturing cross-language communication, be it at runtime [2] or compile time [5]. In practice, protocols consist of components in different languages (e.g., the Apache server communicates with Firefox in TLS) and an altogether unknown attacker. Current composition approaches insist on *translating* base values that are truly incompatible, e.g., bitstrings and abstract DY terms. This leads to shortcomings that we describe in detail (and solve) in Sec. II. In short, the translation approach is:

- **Hard to apply due to strong parsing assumptions:** For instance, keys must always be syntactically distinguishable from bitstrings used elsewhere and network messages must use known encodings [3], [4]. We can avoid this assumption by not requiring a 'universal' translation a priori, but instead by tracking what the application actually does. We elaborate on this in Sec. II-B1 and use Example 1 to show how we solve this problem.
- **Limited in the ability to capture adversarial bit-level reasoning:** The translation approach notoriously struggles with mixed values, for instance, abstract encryption terms or keys that the implementation manipulates on the bit level. In Sec. II-B2, we further explain this issue and discuss our solutions using Examples 4 and 5.
- **Not truly versatile:** The complexity-theoretic computational attacker, e.g., is not compatible with standard language semantics. We argue the compatibility of our framework with a computational attacker in Sec. II-B3.

We solve these issues by forgoing the translation step between such different base types as, e.g., bitstrings and DY terms. The crux is, in our view, that the DY model is a symbolic abstraction (it is sometimes called the 'symbolic model of cryptography' [6]), whereas the translation approach and the above method of composition treats DY terms as if they were concrete values (e.g., see [3, Sec. 4.2]). The first two of the above issues are artifacts of this mismatch.

Consequently, the DY model ought to be composed with LTS that describes interacting components at the same

DOI 10.1109/CSF64896.2025.00030

level of abstraction, that is, with *symbolic execution semantics*. Symbolic execution follows the program, assuming symbolic values (i.e., variables at the object level, so neither program variables nor meta-mathematical variables) for inputs and thus computing symbolic expressions instead of concrete values. Symbolic values can form the 'glue' for communication, allowing us to describe message-passing without translating values from one semantics into the other.

Assuming we have a symbolic execution semantics—devising them is a standard task—we can define a class of LTS with a little more structure, aka *symbolic LTS*, and a new parallel composition operator. The operator exploits symbols for communication and covers the transfer of logical statements made about symbols between both semantics.

This paper is organized into two major parts: Sec. II builds up our framework (source is available at [7]) and introduces the necessary background whenever needed. It starts from LTS and traditional composition, discusses the aforementioned problems in detail, then provides our new method for composition, along with helpful results for composition, refinement and DY attackers. It presents a *framework* for multi-language composition. The second part (Sec. III) presents a challenging application: we instantiate our framework with different languages for the representation of machine code, for the specification of other parties and for the specification of the threat model, and demonstrate the sound extraction of a protocol model from its low-level implementation. Our soundness result ensures the end-to-end correctness of our toolchain, distinguishing it from existing works [8], [9]. Finally, to show the efficacy of our framework, we apply it to verify the TinySSH and WireGuard protocols. To summarize, we make the following contributions:

- We propose a framework for the parallel asynchronous composition of components written in different languages with applications for various methods of analysis, e.g., secure compilation, code-level verification, model extraction (such as ours), or monitoring (see also Sec. IV).
- Using this framework and additional theorems, we support integrating DY attackers into arbitrary languages. This is necessary to make the end-to-end proof feasible.
- We discuss three methods of improving symbolic execution engines with DY support using combined deduction relations (i.e., $\vdash_{12}$) in Sec. II-G.[1]
- We formalized our framework and proved its soundness and the DY attacker and library properties in HOL4 [11].
- We extend CryptoBap [8] toolchain and provide a sound, mechanized verification methodology for the verification of ARMv8 and RISC-V machine code. Thanks to our framework, we simplify the proofs in [8] and fully mechanize them, which was previously unrealistic due to the complexity of the employed computational soundness framework and lack of compositionality.
- We took the opportunity to translate the symbolic results from CryptoBap into Sapic$^+$ [12], a calculus that (soundly) translates to a range of protocol verification backends such as Tamarin [13], ProVerif [14] and DeepSec [15].
- We compare the performance of Sapic$^+$'s backends by proving mutual authentication and forward secrecy within the symbolic model for the implementations of TinySSH and WireGuard.

II. Parallel Composition of Symbolic Semantics

We now present our framework for the composition of symbolic labeled transition systems, starting with revisiting the standard definition of LTS and the conventional communicating sequential processes (CSP)-style [16] asynchronous parallel composition[2]. We use a few illustrative examples to better highlight the translation approach's limitations, which we compensate for by a novel form of parallel composition in a symbolic semantics. In our framework, we distinguish the specific roles of the DY attacker and the DY library. Also, we discuss its capability to deal with other attackers alongside the DY attacker. Finally, we demonstrate the correctness of our approach and, for each theorem, provide access to proofs that we mechanized in HOL4.

A. LTS and their composition

LTS provides a generic semantic model for capturing the operational semantics of systems [17], [18]. An LTS consists of a set of states (aka configurations) $\mathcal{C}$ connected by a transition relation $\xrightarrow{\alpha} \subseteq \mathcal{C} \times \mathbb{E} \times \mathcal{C}$ that releases an event $\alpha \in \mathbb{E}$ when the system moves between states, and an initial state $c \in \mathcal{C}$ within that space. Given that a language has a formalized semantics, a program behavior can typically be described as an LTS. Thus, it is interesting to combine LTS to reason about heterogeneous systems, wherein some transitions are asynchronous, e.g., programs are performing internal computations independently, while others are synchronized, e.g., one program sends a message and the other one receives it.

CSP-style asynchronous parallel composition supports both types of transitions and can be applied to LTS. Transitions are synchronous if both carry the same event ($\alpha \in \mathbb{E}_1 \cap \mathbb{E}_2$), and all others are asynchronous. Hence, in a composed state (c_1, c_2), we move synchronously to (c_1', c_2') with event α provided *both* systems can move ($c_1 \xrightarrow{\alpha} c_1'$ and $c_2 \xrightarrow{\alpha} c_2'$) and otherwise ($\alpha \notin \mathbb{E}_1 \cap \mathbb{E}_2$), we move to (c_1', c_2) or (c_1, c_2') if either of the systems can make a transition.

Synchronizing events can be used to transmit messages [1], [2]. For example, when combining two systems A and P with a shared event $A2P(m)$, system A can have a rule that determines m from its current state, whereas P has a rule that non-deterministically accepts $A2P(m^*)$ for any m^* and incorporates it into the follow-up state. Combining both systems via asynchronous parallel composition, we obtain synchronous message passing from A to P.

[1] We use **RoyalBlue, math bold**, RedOrange, sans serif, and Plum, typewriter to differentiate between different languages [10]. Elements common to all languages, including symbols, are typeset in *black ita*

[2] We prefer a less descriptive, but shorter name, assuming that modern ems require both synchronous and asynchronous transitions.

B. Message passing and Dolev-Yao attackers

A very appealing proposal is to let A designate a DY attacker and P a program in some general-purpose language, as to obtain a semantics to reason about the interaction of any such P with a network adversary. Most recently, this approach was used to leverage separation logic for the verification of network systems [2]–[4], and earlier to provide sound analyses for Dalvik bytecode [1].

The DY model is a model of cryptography where the attacker only makes deductions defined by a set of rules. It has been enormously successful in verifying security protocols, as it automates the verification procedure [19]. These rules do not necessarily cover all possible attacks and require additional justification [6], [20]. Typically, the DY attacker and the protocol share an unbounded set of *names* that represents keys and other hard-to-guess values. The model ensures the attacker and protocol always draw fresh names, hence key collisions are impossible. Names and public values can be combined with free function symbols to terms. E.g., $\mathsf{senc}(m, k)$ is a term that represents an encryption. It is not interpreted further. This so-called *term algebra* is complemented by a small set of rules that allows operations beyond the application of these symbols. E.g., a rule for decryption that says from $\mathsf{senc}(m, k)$ and k, the attacker learns m. We will make these notions explicit later. When parallel composing a DY attacker with a language where keys and messages are represented as bitstrings, it is necessary to translate DY terms to bitstrings and vice versa. This, however, has several caveats.

1) Parsing assumptions: First, it requires strong and unrealistic parsing assumptions to transform bitstrings back into terms that have more structure. For instance, keys must always be distinguished from bitstrings used elsewhere [4]. When we consider the space of AES keys, which (in reality) covers all bitstrings of length 128 (or 192 or 256), this requires (artificial) tagging to distinguish those from other bitstrings of that size, which real-world implementations do not have and actively avoid for performance. Another issue is the use of bitstring manipulation for message formatting.

Example 1 (Bitstring manipulation)**.** *Concatenation is essential in the implementation of crypto protocols. It is associative and, hence, not easy to reason about automatically; thus, usually, this operation is not part of the DY term algebra. Consider the example where a message* $\mathbf{m}$ *is concatenated with its length to simplify parsing. Without further workarounds, the DY attacker can not determine* $\mathbf{m}$ *from* $\mathsf{senc}(\mathbf{m}\|\mathsf{len}(\mathbf{m}), k)$*, even if it possesses the encryption key* k*. As we show later, the DY attacker can derive* $\mathbf{m}$ *from* $\mathbf{m}\|\mathsf{len}(\mathbf{m})$ *by employing the deduction combinator* $\vdash^{\mathsf{bit}}_{12}$ *in Eq. bit defined in Sec. II-G3.*

The translation approach supports message formatting, of course, otherwise it would be impractical. It works around this issue by modeling every message format that is used as a DY function symbol [3, Sec. 3.1]. For full TLS, there are at least 189 message formats [21, Sec. 5.1]. Clever refactoring may reduce this number (formats can be nested), but this is non-trivial and tedious. Most importantly, we would like our mechanism to be protocol-agnostic, even if it is to a particular set of cryptographic functions. In contrast, techniques like DY^* and Comparse [22], [23] integrate bit-level and DY reasoning within the same tool, enabling the analysis of a (protocol-specific) set of message formats at the bit-level and then performing a DY analysis on abstract types. This avoids the problem and is discussed in Sec. IV.

2) Loss of bit-level information: Manipulating DY terms in the context of another language's semantics produces non-DY bitstrings that cannot be properly represented and translated back into their correct form. Therefore, these bitstrings become untraceable to their DY origins and an irreversible element to the transformation. As a result, translation approaches weaken the DY attacker in reasoning about the messages altered by a protocol party using a different language. E.g., say A wants to learn P's secret s and can trick P into encrypting $s{+}\mathbf{0x1}$ with a known key k. The DY attacker receives a bitstring corresponding to $\mathsf{senc}(s{+}\mathbf{0x1}, k)$, and after decrypting with k, has to recognize the transformation applied to $s{+}\mathbf{0x1}$ (and that it requires subtracting $\mathbf{0x1}$). Examining the huge number of possible transformations is out of the question, particularly when considering Turing-complete machine semantics (e.g., the wrappers in [5]). Typically, as bitstring addition $+$ does not correspond to the image of a term constructor, such unknown bitstrings are translated into garbage DY terms [2] or terms we do not know [3], [4]. In contrast to message formats, this output was unintended. Using Examples 4 and 5, we explain our solutions to this problem in Sec. II-G.

3) Not truly versatile, compatibility with computational model: The DY model is a symbolic abstraction that is well-accepted in protocol verification, but not throughout information security. It is useful to be able to replace DY attackers with computational attackers for flexibility or to validate the DY attacker's soundness. This is incompatible or difficult, depending on how the translation approach is realized. In [3], [4], a function translates from terms to bitstrings (i.e., the inverse direction to parsing discussed above). In the computational model, this relationship is not functional. For instance, a DY term representing a key, i.e., a *name*, may translate to many different bitstrings, depending on how they are sampled. Consequently, the computational attacker in these works is not an attacker in the traditional sense (an arbitrary probabilistic algorithm limited only in runtime) but the DY attacker inside a function translating from and to bitstrings.

Fortunately, a long line of work on computational soundness [6] explored requirements for such a translation, which must be probabilistic. Alas, known results come with a long list of requirements, both on programs and cryptographic primitives they use, that are hard to fulfill. To even formulate these requirements, the target semantics need to be equipped with a probabilism, non-determinism for communication and a notion of polynomial runtime in the length of some parameter that governs the key size and similar parameters. While there are methods to encode all of these, programming languages are rarely formalized with these features in mind. We can point to Aizatulin's Ph.D. thesis [9] as a case study for such a semantics the required technical machinery.

C. Symbolic Execution Semantics

Symbolic execution explores all program execution paths using symbolic values—introduced at the object level—instead of concrete ones for inputs. An example is a language with a memory that maps registers to bitstrings. Its symbolic execution allows the memory to map registers to either symbols or bitstrings. Starting from an initial symbolic state, the execution explores all possible paths and collects the execution effects in a final symbolic state for each path. Each symbolic state, in addition to a map from variables to symbolic expressions (i.e., where symbols represent initial state variables), also contains a path condition that is a logical predicate describing what is known about the symbol. For instance, $r_A = \texttt{0x0} \vee r_A = \texttt{0x1}$ if register r_A is known to be either 0 or 1 because it passed some condition. To combat the path explosion problem, symbolic execution engines make logical deductions on these predicates to prune paths that are unreachable (e.g., using an SMT solver). The more powerful the deduction engine, the fewer paths need to be explored, but the more computationally expensive these deductions are.

We capture these elements—symbols, predicates, and deductions—by giving our LTS more structure. Let τ be the silent transition, then:

Definition 1 (Symbolic LTS)**.** *A symbolic LTS is an LTS* $(\tilde{\mathcal{C}}, \mathbb{E}, \rightarrow)$ *for which there is a symbol space* $\mathcal{E}$*, a predicate space* $\mathcal{P}$*, and a deduction relation* $\vdash \subseteq 2^{\mathcal{P}} \times \mathcal{P}$ *such that:*

- $\tilde{\mathcal{C}} = 2^{\mathcal{E}} \times 2^{\mathcal{P}} \times \mathcal{C}$ *for some state space* $\mathcal{C}$ *and*
- *For any predicate set* Π*, predicate* φ*, symbols set* Σ*, and state* c*, we have:* $\Pi \vdash \varphi \implies (\Sigma, \Pi, c) \xrightarrow{\tau} (\Sigma, \Pi \cup \{\varphi\}, c)$.

For brevity, we denote such LTS with $(\mathcal{E}, \mathcal{C}, \mathbb{E}, \rightarrow, \mathcal{P}, \vdash)$.

The second condition establishes the relation between the logical deduction relation which is language-specific, and the current predicate set: logical deductions can be made at any time, and the knowledge we conclude (encoded inside the predicate) is added to the symbolic state. Typically, the state space $\mathcal{C}$ and event space $\mathbb{E}$ are built on the symbol space $\mathcal{E}$, e.g., in the example above, the symbolic memory was a function from registers to the union of bitstrings and the set of symbols $\Sigma \subseteq \mathcal{E}$. This is only implicit in the mathematical notation, it is, however, explicit in our HOL4 formalization, where the types of $\mathcal{C}$ and $\mathbb{E}$ are parametric in the (polymorphic) type $\mathcal{E}$. The first element Σ mainly tracks which symbols have been used so far, increasing monotonically.

Every symbolic LTS, also referred to as a component, must transmit only references to their messages in the form of symbols to other components. Symbols relate to the values that are transmitted like a variable n relates to the set of integers, i.e., as a representation. When a value is manipulated, the relation between the original and the changed value, each represented by a different symbol, is itself represented with a predicate connecting the two symbols. Consequently, a symbol always signifies the same value (in a run), and the predicates associated with distinct components articulate the same properties.

D. Symbolic Parallel Composition

We now define a parallel composition that behaves like CSP-style asynchronous parallel composition but has an important twist: it is parametric in a *combined deduction relation*, which serves to transfer judgments from one system into the other. In the follow-up, we show that there are several ways to define this that increases the set of possible deductions and, thus, the precision of the analysis, while also being compatible with almost all judgments made in programming languages.[3] Let $\lfloor_i : 2^{\mathcal{P}_1 \uplus \mathcal{P}_2} \rightarrow 2^{\mathcal{P}_i}$ denote the projection[4] to $i \in \{1, 2\}$, then:

Definition 2 (Symbolic Parallel Composition)**.** *Given two symbolic LTS* $S_i = (\mathcal{E}, \mathcal{C}_i, \mathbb{E}_i, \rightarrow_i, \mathcal{P}_i, \vdash_i)$*,* $i \in \{1, 2\}$ *with* identical symbol space $\mathcal{E}$ *and a combined deduction relation* $\vdash_{12} \subseteq 2^{(\mathcal{P}_1 \uplus \mathcal{P}_2)} \times (\mathcal{P}_1 \uplus \mathcal{P}_2)$*, we define their symbolic parallel composition* $\mathbf{S_1} \parallel^{\vdash_{12}} S_2$ *as the symbolic LTS* $(\mathcal{E}, \mathcal{C}_1 \times \mathcal{C}_2, \mathbb{E}_1 \cup \mathbb{E}_2, \rightarrow_{12}, \mathcal{P}_1 \uplus \mathcal{P}_2, \vdash_{12})$*, where*

- $\rightarrow_{12}$ *moves asynchronously, i.e., either* $(\Sigma, \Pi_{12}, \mathbf{c_1}, c_2) \xrightarrow{\alpha_1}_{12} (\Sigma', \Pi'_{12}, \mathbf{c'_1}, c_2)$ *or* $(\Sigma, \Pi_{12}, \mathbf{c_1}, c_2) \xrightarrow{\alpha_2}_{12} (\Sigma', \Pi'_{12}, \mathbf{c_1}, c'_2)$*, if, for* $i \in \{1, 2\}$*, we can move with* $\alpha_i \in \mathbb{E}_i \setminus (\mathbb{E}_1 \cap \mathbb{E}_2)$*, i.e.,* $(\Sigma, (\Pi_{12} \lfloor_i), c_i) \xrightarrow{\alpha_i}_i (\Sigma', (\Pi'_{12} \lfloor_i), c'_i)$*, keeping the complement's* [5] *predicate set untouched* $\Pi_{12} \lfloor_{\bar{i}} = \Pi'_{12} \lfloor_{\bar{i}}$*, or*
- $\rightarrow_{12}$ *moves synchronously, i.e.* $(\Sigma, \Pi_{12}, \mathbf{c_1}, c_2) \xrightarrow{\alpha}_{12} (\Sigma', \Pi'_{12}, \mathbf{c'_1}, c'_2)$*, if, for* $i \in \{1, 2\}$*,* $(\Sigma, (\Pi_{12} \lfloor_i), c_i) \xrightarrow{\alpha}_i (\Sigma'_i, (\Pi'_{12} \lfloor_i), c'_i)$*,* $\alpha \in \mathbb{E}_1 \cap \mathbb{E}_2$*, and* $\Sigma' = \mathbf{\Sigma'_1} \cup \Sigma'_2$.
- *From the second condition of Def. 1 we have:* $\Pi_{12} \vdash_{12} \varphi_{12} \implies (\Sigma, \Pi_{12}, \mathbf{c_1}, c_2) \xrightarrow{\tau}_{12} (\Sigma, \Pi_{12} \cup \{\varphi_{12}\}, \mathbf{c_1}, c_2)$.

Def. 2 preserves fundamental properties of parallel composition like symmetry and associativity (see *Symmetry* and *Associativity* for the mechanized proofs in HOL4).

Note that even if $\vdash_{12}$ is empty (short: $\parallel^{\text{def}}_{=} \parallel^{\emptyset}$) the symbolic parallel composition is different from the classical parallel composition of the corresponding LTS. The symbol set is shared between both symbolic LTS even when they move asynchronously. Typically, symbolic LTS uses the symbol set to ensure that new symbols are fresh; as we use symbols for communication, we want to ensure they are globally fresh.

Moreover, if $\vdash_{12}$ is not empty, it allows deriving judgments in one system from judgments in the other system. Of course, we want to avoid this relation to be overly tied to one or the other system. Before we discuss how to do that, we will showcase how the DY model is represented as a symbolic LTS. This provides us with concrete examples to illustrate how $\vdash_{12}$ can overcome the issues from Sec. II-B (cf. Examples 4 and 5 for their solutions).

E. DY Attackers

The DY model considers an attacker that exploits logical weaknesses in a protocol, but is not able to break cryptographic

[3] From hereon, we will combine different systems with oftentimes incompatible base types. To make it easier for the reader to type-check our statements, we will use colors to remark which system we speak of.

[4] We present this using a disjoint union ($\uplus$) for familiarity and simpler presentation, while we employ a sum type in our HOL4 formalization.

[5] .e., $\bar{i} \in \{1, 2\}$ with $\bar{i} \neq i$.

$$\frac{\mathcal{K}(t) \in \Pi_A}{\Pi_A \vdash_A \mathcal{K}(t)}\ \text{K}_0 \qquad \frac{\Pi_A \vdash_A \mathcal{K}(t_1) \cdots \Pi_A \vdash_A \mathcal{K}(t_n) \quad f \in \mathcal{F}^n}{\Pi_A \vdash_A \mathcal{K}(f(t_1, \ldots, t_n))}\ \text{App}$$

$$\frac{n \in \mathcal{N}_{\text{pub}}}{\Pi_A \vdash_A \mathcal{K}(n)}\ \text{Pub} \qquad \frac{\Pi_A \vdash_A \mathcal{K}(t_1) \quad \Pi_A \vdash_A t_1 \doteq t_2}{\Pi_A \vdash_A \mathcal{K}(t_2)}\ \text{Subst}$$

$$\frac{t_1 =_E t_2}{\Pi_A \vdash_A t_1 \doteq t_2}\ \text{Eq} \qquad \frac{\Pi_A \vdash_A \mathcal{K}(x) \quad \Pi_A \vdash_A x \mapsto t}{\Pi_A \vdash_A \mathcal{K}(t)}\ \text{Al-Subst}$$

Fig. 1: The DY attacker's deduction rules. Top-left to bottom-right: the attacker knows the messages it received and can apply function symbols. If a name is public, the attacker knows this name, and equivalent names to fresh names are also fresh. Equivalence modulo E translates into an equivalence judgment, and if a term is known, any equivalent terms are also known. Any terms that correspond to a given symbol are known if the symbol itself is known.

primitives. Cryptography is assumed to be perfect; events that can occur in the real world, albeit with negligible probability—for example, guessing a key—are altogether impossible in this model. A small set of rules governs how messages can be manipulated on an abstract level; every other manipulation is excluded. Concretely, messages are modeled as *terms*[6]. Constants are taken from an infinite set of names $\mathcal{N}$, divided into public names $\mathcal{N}_{\text{pub}}$ (e.g., agent names) and secret names $\mathcal{N}_{\text{priv}}$ (like keys and nonces). We also assume a set of variables $\mathcal{V}$ for values that the DY attacker receives. The set of terms $\mathcal{T}$ is then constructed over names in $\mathcal{N}$, variables in $\mathcal{V}$ and applications of function symbols in $\mathcal{F}$ on terms. Let $f \in \mathcal{F}^n$ denote a function symbol with arity n. For the moment, we consider only two function symbols, $\mathcal{F} = \{\mathsf{senc}, \mathsf{sdec}\} = \mathcal{F}^2$. The term $\mathsf{senc}(m, k)$ models the symmetric encryption of another term m with the key $k \in \mathcal{N}_{\text{priv}}$. A set of equations $E \subset \mathcal{T} \times \mathcal{T}$ provides these terms with a meaning. Let us define $E = \{\mathsf{sdec}(\mathsf{senc}(x, y), y) = x\}$ to account for the fact that decryption reverses encryption (for the same key). We can define an equivalence relation $=_E$ as the smallest equivalence relation containing E that is closed under the application of function symbols and substitution of variables by terms. Now, $\mathsf{sdec}(\mathsf{senc}(m, k), k) =_E m$.

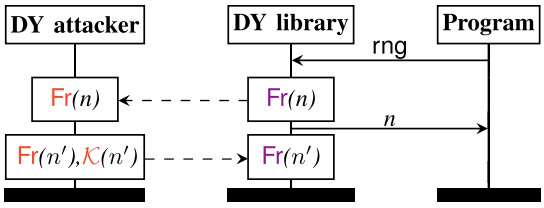
The predicate set of the DY attacker has three types of facts: their knowledge of a term t, written as $\mathcal{K}(t)$, is derivable from the set of predicates Π_A seen or derived so far, two terms are considered equivalent $t_1 \doteq t_2$ according to $=_E$ and a name n is fresh $\mathsf{Fr}(n)$. The deduction relation (Fig. 1, see caption) mostly describes how $\mathcal{K}(\cdot)$ is derived. $t_1 \doteq t_2$ and $x \mapsto t$ represent $=_E$ and $\mapsto$ (i.e., denotes the mapping of variables to terms) at the logical level, respectively. With the deduction relation in place, we can now define the transition relation (Fig. 2). Besides the symbol set Σ and the predicate set Π_A, the DY attacker is stateless, indicated by ϵ for the empty state.

A message is received by synchronization with the event $P2A(x)$ emitted by another component. As the DY adversary cannot process the incoming message type (e.g., bitstrings) directly, we must assume x is a symbol. Therefore, we set $\mathcal{V} = \mathcal{E}$. Hence, in P2A, x is determined by the environment (e.g., the sending component) and the attacker record the fact that it is known.

If the predicate set Π_A witnesses that the symbol x from the set Σ represents a value known to the DY attacker, the attacker can send x to another component (A2P). But not all knowledge predicates within Π_A are over symbols; encryption terms, for instance. Hence, the Alias rule can be used to introduce a new symbol, which can be transmitted. Recall that the Al-Subst rule in Fig. 1 introduces the required $\mathcal{K}(\cdot)$-predicate via the deduction relation. The transition rule Ded integrates the deduction relation. It is simply the minimal rule required to satisfy the condition in Def. 1.

Fresh names can be drawn by the attacker, but also by other components (see Fr-L2A in Fig. 3). Fr-A2L is a synchronous step between the DY attacker and other components that deals with the first case, where the attacker learns the name, which is marked as fresh and thus 'taken'. In the second case, another component, typically the crypto library of some party, picks a key (or another high-entropy value) that is marked as fresh. The DY attacker must also mark those names as 'taken', hence the other component synchronizes their picking of this value using the synchronous Fr-L2A rule in Fig. 3. This synchronization is necessary, as Example 2 shows.

Example 2 (DY communicates with crypto library)**.** *To generate a random number, a request needs to be sent to the library (e.g.,* rng*). The library maintains a record of the generated random numbers within its predicate set (i.e.,* $\{\mathsf{Fr}(n)\}$*) to ensure the creation of unique names. The DY attacker has the ability to choose a name (e.g.,* n'*) as long as it differs from the choice made by the library for the program (i.e.,* n*).*



The library's predicate set is updated using the synchronous Fr-L2A *rule in Fig. 3 and* $\mathsf{Fr}(n)$ *is added to the predicate set of the attacker by the synchronous* Fr-L2A *rule in Fig. 2 for the library's initial random number generation. The second update is performed by the attacker using the synchronous* Fr-A2L *rule in Fig. 2 and the attacker is not able to pick the library chosen name* n *as* $\mathsf{Fr}(n)$ *exists in the attacker predicate set* Π_A*. Therefore, the attacker chooses a fresh name* n'*, and the* $\mathsf{Fr}(n')$ *is added into the predicate set of the library by the synchronous* Fr-A2L *rule in Fig. 3.*

In our examples, **solid** arrows represent direct communications between components, while **dashed** arrows denote the implicit flow of facts between the DY library and the DY attacker. Also, each step within the action box of components signifies the logical predicates added to their predicate sets ing execution.

[6]The DY attacker and the DY library (but not the program) use the same terms. To simplify the presentation, we typeset them in *black italics*. as otherwise, they would be orange *and* purple.

$$\frac{x \notin \Sigma \quad \Pi_A' = \Pi_A \cup \{\mathcal{K}(x)\}}{(\Sigma, \Pi_A, \epsilon) \xrightarrow{P2A(x)}_A (\Sigma \cup \{x\}, \Pi_A', \epsilon)}\ \text{P2A} \qquad \frac{\mathcal{K}(x) \in \Pi_A \quad x \in \Sigma}{(\Sigma, \Pi_A, \epsilon) \xrightarrow{A2P(x)}_A (\Sigma, \Pi_A, \epsilon)}\ \text{A2P}$$

$$\frac{\begin{array}{ccc} x \notin \Sigma & \forall i \le n : t_i \in \mathcal{T} & 0 < n \\ f \in \mathcal{F}^n & \Pi_A' = \Pi_A \cup \{x \doteq f(t_1, \ldots, t_n)\} & \end{array}}{(\Sigma, \Pi_A, \epsilon) \xrightarrow{\mathsf{Alias}(x, f(t_1, \ldots, t_n))}_A (\Sigma \cup \{x\}, \Pi_A', \epsilon)}\ \text{ALIAS}$$

$$\frac{\begin{array}{cc} n \in \mathcal{N}_{\mathtt{priv}} & \Pi_A' = \Pi_A \cup \{\mathsf{Fr}(n)\} \\ \mathsf{Fr}(n) \notin \Pi_A & \end{array}}{(\Sigma, \Pi_A, \epsilon) \xrightarrow{SFr(n)}_A (\Sigma, \Pi_A', \epsilon)}\ \text{FR-L2A} \qquad \frac{\begin{array}{cc} n \in \mathcal{N}_{\mathtt{priv}} & \Pi_A' = \Pi_A \cup \{\mathsf{Fr}(n), \mathcal{K}(n)\} \\ \mathsf{Fr}(n) \notin \Pi_A & \end{array}}{(\Sigma, \Pi_A, \epsilon) \xrightarrow{Silent(n)}_A (\Sigma, \Pi_A', \epsilon)}\ \text{FR-A2L}$$

$$\frac{\Pi_A \vdash_A \pi \quad \Pi_A' = \Pi_A \cup \{\pi\}}{(\Sigma, \Pi_A, \epsilon) \xrightarrow{\tau}_A (\Sigma, \Pi_A', \epsilon)}\ \text{DED}$$

Fig. 2: The transition relation rules for Dolev-Yao attacker model.

Event	Purpose	Involved components
FCall	Library calls	Program and Library
SFr	Calls to RNG	Program, Library and Attacker
A2P, *P2A*	Network communication	Program and Attacker
Silent	Ensure freshness	Library and Attacker

TABLE I: Summary of synchronizing events. *FCall*, *SFr*, *A2P*, and *P2A* synchronize with the program component, whereas *Silent* is internal to the DY Libary and DY Attacker.

Observe that DY attackers do not pick honestly generated names (e.g., in Example 2) due to the synchronization on freshness facts. The ALIAS rule (Fig. 2) only generates new symbols, not names (i.e., chooses x from the symbol set Σ). Contrast this with [2], where the syntactic structure of names binds them to protocol roles, including the attacker, or the follow-up work [3] where names do not need to carry structure, but a global restriction on traces is applied to ensure uniqueness. In both cases, this aspect of the DY attacker is thus hard-coded into the (global) trace model, which we can avoid.

F. Dolev-Yao Libraries

To equip a programming language with a DY semantics, we also need to mark crypto operations as such, i.e., specify when a function output ought to be abstracted by an encryption term like $\mathsf{senc}(\cdot,\cdot)$. One way is to integrate the term algebra into the predicate space $\mathcal{P}$ and mark crypto outputs via equalities, e.g., a logical predicate saying 'symbol z is equivalent to the DY term $\mathsf{senc}(x, y)$' (where x and y can be other symbols). A more generic way to achieve the same effect is by composing the function calls with a DY library that performs those abstraction steps. Fig. 3 shows how the composition can be done, with the FCALL applying a function symbol similar to the APP but including the ALIAS. Like the DY attacker, the DY library is stateless.

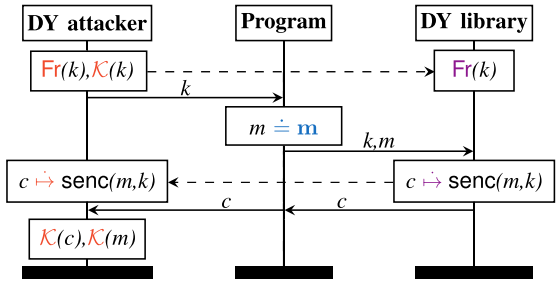
The deduction relation of the DY library is defined via an equivalence relation $=_E$. In verification tools like SAPIC$^+$, the relation $=_E$ arises as the smallest congruence relation that is closed under substitutions and contains the set of equations E provided by the user. For the sake of the formalization, $=_E$ is an arbitrary equivalence relation. The equations E used in our case studies are provided in the SAPIC$^+$ input file.

As we know the DY library and DY attacker and their respective predicate sets, we can use the combined deduct relation $\vdash^{\dot{\mapsto}}_{LA}$ to share the mapping predicates, as follows:

$$\Pi_L \uplus \Pi_A \vdash^{\dot{\mapsto}}_{LA} x \mapsto t \;\Leftrightarrow\; x \mapsto t \in \Pi_L$$
$$\Pi_L \uplus \Pi_A \vdash^{\dot{\mapsto}}_{LA} x \mapsto t \;\Leftrightarrow\; x \mapsto t \in \Pi_A \qquad (\vdash^{\dot{\mapsto}}_{LA})$$

Table I summarizes the interface to the DY attacker and library from the perspective of a protocol component, which could be, for instance, a **BIR** program, as in our case studies. For instance, if the protocol wanted to generate a random number, it would use *SFr*, which synchronizes with FR-L2A in Fig. 2 and Fig. 3.

Example 3 (Logical truth)**.** *Predicates can be shared between components without explicit communication.*



Thus, the DY attacker can uncover the message $\mathbf{m}$ *using the known key* k *and the mapping* $c \mapsto \mathsf{senc}(m, k)$*, without communicating with the library. The steps to acquire the message* $\mathbf{m}$ *are as follows: upon receiving* c *from the program and obtaining* $c \mapsto \mathsf{senc}(m, k)$ *through* $\vdash^{\dot{\mapsto}}_{LA}$*, the attacker uses the* AL-SUBST *rule to get* $\mathcal{K}(\mathsf{senc}(m, k))$*. Next, the attacker utilizes their knowledge and* $\mathsf{sdec} \in \mathcal{F}^2$ *to learn* $\mathsf{sdec}(\mathsf{senc}(m, k), k)$ *using the* APP *rule in Fig. 1. Leveraging the relation* $=_E$ *detailed in Sec. II-E, along with the* EQ *and* SUBST *rules (Fig. 1), the attacker obtains the knowledge of* m*. Without the mapping predicate linking the ciphertext and encryption term, the attacker would lack the necessary knowledge to apply the* APP *rule for decryption, leaving the encryption term undisclosed.*

Example 3 shows how the DY library, the attacker, and the program cooperate when the library generates a ciphertext using an adversarial key. As a nice extra, such a library allows us to prove a composition property that is convenient when different programs use multiple libraries (cf. full version [24, ppendix C]).

$$\frac{n \in \mathcal{N}_{\text{priv}} \quad \mathsf{Fr}(n) \notin \Pi_{\text{L}} \quad \Pi_{\text{L}}' = \Pi_{\text{L}} \cup \{\mathsf{Fr}(n)\}}{(\Sigma, \Pi_{\text{L}}, \epsilon) \xrightarrow{SFr(n)}_{\text{L}} (\Sigma, \Pi_{\text{L}}', \epsilon)}\ \textsc{Fr-L2A} \qquad \frac{n \in \mathcal{N}_{\text{priv}} \quad \mathsf{Fr}(n) \notin \Pi_{\text{L}} \quad \Pi_{\text{L}}' = \Pi_{\text{L}} \cup \{\mathsf{Fr}(n)\}}{(\Sigma, \Pi_{\text{L}}, \epsilon) \xrightarrow{Silent(n)}_{\text{L}} (\Sigma, \Pi_{\text{L}}', \epsilon)}\ \textsc{Fr-A2L}$$

$$\frac{y \notin \Sigma \quad \forall i \le n : x_i \in \Sigma \quad f \in \mathcal{F}^n \quad \Pi_{\text{L}}' = \Pi_{\text{L}} \cup \{y \doteq f(x_1, \ldots, x_n)\}}{(\Sigma, \Pi_{\text{L}}, \epsilon) \xrightarrow{FCall(f, x_1, \ldots, x_n, y)}_{\text{L}} (\Sigma \cup \{y\}, \Pi_{\text{L}}', \epsilon)}\ \textsc{FCall}$$

Fig. 3: The transition relation rules for Dolev-Yao library model.

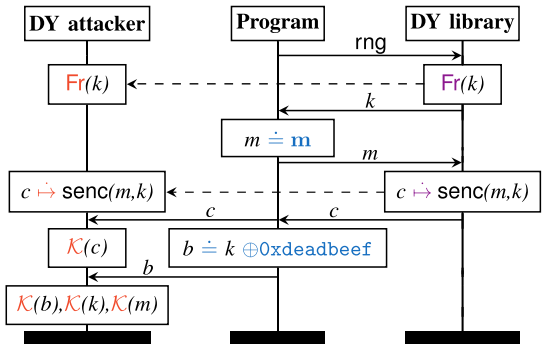
G. Deduction combiners

Symbolic parallel composition's strength lies in its ability to transfer judgments between systems. There is a trade-off between precision and generality. We discuss some useful combiners from the most general to the most precise.

1) Generic over- and under approximation: In general, bitstring operations can reveal cryptographic information. Example 4 shows how to under-approximate or over-approximate the adversaries' capabilities on operating with bitstrings.

Example 4 (Masked encryption key)**.** *In this example, the attacker obtains a message* $\mathbf{m}$ *encrypted with a fresh key* k*, followed by the key masked with a known constant* `0xdeadbeef`*. Using the combined deduction relation* $\vdash^{\doteq}_{LA}$ *(which is defined specifically for DY attacker and DY library), the mapping* $c \doteq\!\!\mapsto \mathsf{senc}(m, k)$ *transfers from DY library to DY attacker.*

The last message b *ought to reveal the plaintext* $\mathbf{m}$*. In the following, we introduce an* over-approximating *deduction combiner* $\vdash^{\top}_{\mathbf{12}}$ *(Eq. over-approx) that allows the DY attacker to infer* $\mathcal{K}(k)$ *from* $\mathcal{K}(b)$ *and* $b \doteq k \oplus$ `0xdeadbeef` *and thus the plaintext* $\mathbf{m}$ *(from* $\mathcal{K}(c)$*,* $c \doteq\!\!\mapsto \mathsf{senc}(m, k)$ *and* $\mathcal{K}(k)$*).*



With an empty deduction combiner, the masked bitstring in the second network message is only accessible via the symbol b. The DY attacker can perform DY operations on the symbol b, but there is no way to access the k symbol without reasoning about the bitstring. Hence the empty deduction combiner under-approximates the adversaries' capabilities on operating with bitstrings. This is equivalent to the view in [2]–[4], where the concrete attacker is simply a translation function around the DY attacker. If a bitstring that cannot be parsed is encountered, it can only be ignored.

At the opposite end of the spectrum, Backes et al. [1] aimed for computational soundness, which entails that all attacks that could be mounted by a Turing machine must be captured by the DY attacker. As the Turing machine can reverse $\oplus$ operation in the above example, this required an over-approximation where all bitstring operations were represented in the DY model as *transparent* function symbols, i.e., function symbols whose input parameters are fully accessible.

We can generically represent this over-approximation in our framework, if we have an equality predicate $\doteq$ in the program's predicate set and we can identify the set of symbols that appear on either side, say, using a function named *symbols*:

$$\Pi_{\mathbf{1}} \uplus \Pi_2 \vdash^{\top}_{\mathbf{12}} \mathcal{K}(z) \Leftrightarrow \exists\ x, y.$$
$$\mathcal{K}(x) \in \Pi_2 \wedge x \doteq y \in \Pi_{\mathbf{1}} \wedge z \in \mathit{symbols}(y) \quad \text{(over-approx)}$$

This over-approximation can introduce spurious equalities that lead to false attacks. For example, it is reasonable that a logic for bitstrings can conclude $a \doteq a \oplus x \oplus x$ for any a and x. This could easily introduce a spurious dependency between some a transmitted to the attacker and an arbitrary symbol x.

2) Sharing equalities: If we can identify at least one equality predicate in both component's predicate space, however, we can find a much more useful middle ground between both extremes (i.e., over- and under approximation). Connecting equality judgments in both systems may allow tracking data flow across system boundaries, while requiring nothing more than to point out the equality predicates.[7]

Let $\mathcal{P}_{\mathbf{1}}$ and $\mathcal{P}_2$ contain atoms $\doteq$ and $\doteq$ such that symbols can appear on each side of either of them, i.e., $x \doteq_i y \in \mathcal{P}_i$ for $i \in \{\mathbf{1}, 2\}$ and $x, y \in \mathbf{\Sigma_1} = \Sigma_2$. Then we can transfer equalities with the minimal deduction combiner defined by the following statements:

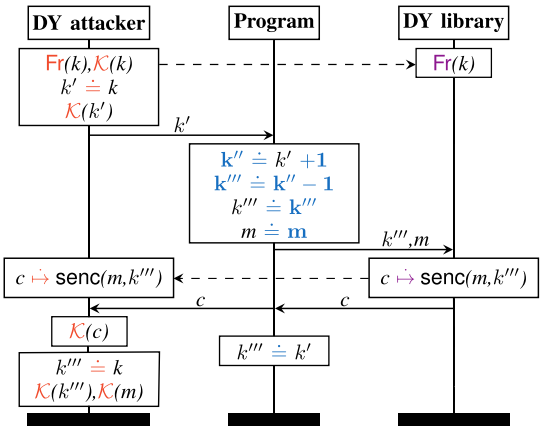
$$\Pi_{\mathbf{1}} \uplus \Pi_2 \vdash^{\mathsf{eq}}_{\mathbf{12}} x \doteq z \Leftrightarrow \exists y.\ x \doteq y \in \Pi_{\mathbf{1}} \wedge y \doteq z \in \Pi_2 \quad (\doteq)$$
$$\Pi_{\mathbf{1}} \uplus \Pi_2 \vdash^{\mathsf{eq}}_{\mathbf{12}} x \doteq z \Leftrightarrow \exists y.\ x \doteq y \in \Pi_{\mathbf{1}} \wedge y \doteq z \in \Pi_2 \quad (\doteq)$$

The following example shows how we address the *loss of bit-level information* discussed in Sec. II-B2 where the DY attacker could not analyze cryptographic secrets with bit-level modifications. Even though the DY attacker still cannot directly analyze bitstrings, they can now leverage the program's analysis by transferring equivalences through equality combiners (Eq. $\doteq$ and Eq. $\doteq$). This is essential when the protocol implementation involves packing (i.e., formatting messages so that the other party on the network can read them) and unpacking (i.e., extracting the message).

Example 5 (Transferable equalities)**.** *Equality can easily be transferred to accrue logical deduction relations. A* $\doteq$ *predicate can be added to the program's predicate set, e.g., similar*

[7]This task could even be automated by (heuristically) identifying an equality predicate of arity two that is symmetric, reflexive and transitive.

to what will be discussed in Sec. III-A1. In the following procedure block, the deduction relation $\vdash_1$ is used to deduce $k''' \doteq k'$. Given $k''' \doteq k'$ and $k' \doteq k$, the attacker infers $k''' \doteq k$ using Eq. $\doteq$. Knowledge of k''' is derived from $\mathcal{K}(k)$ and $k''' \doteq k$ employing the SUBST *rule in Fig. 1. Consequently, the attacker learns* $\mathbf{m}$ *by knowing k''', c, and $c \mapsto \mathsf{senc}(m, k''')$.*



3) Combined reasoning: Equality sharing can transfer many statements derived from the other component into the predicate space of the DY attacker, but (a) only those that discuss the relation between term sent or deduced by the attacker (as only those have symbols), and, (b) only if the other component has sufficient information to derive an equality judgment.

Coming back to Example 4, we see that the masking around the encryption key must be removed to deduce k from b. But as the program does not perform that operation, the necessary equality (between k and the potential result of such an operation) is not produced. The *ability* to perform this operation must be described via the $\mathcal{K}(\cdot)$ predicate rather than $\doteq$. A sound way of doing that would be to enhance the DY attacker with bitstring manipulation via constant values.

$$\Pi_1 \uplus \Pi_2 \vdash^{\mathsf{bit}}_{12} \mathcal{K}(x) \Leftrightarrow \exists\, y, c.\ \mathcal{K}(y) \in \Pi_2 \wedge y \doteq \mathsf{op}(x, c) \in \Pi_1 \wedge \mathsf{const}\ c \in \Pi_1 \quad \text{(bit)}$$

This combinator depends on the predicate space $\mathcal{P}_1$ providing a predicate $\mathsf{const}\ c$ that indicates a constant and needs to explicitly list all binary operators $\mathsf{op}(x, c)$. It thus cannot be regarded as generic, although these concepts (operators and constants) should apply to many programming languages. Again recalling Example 4, we can use $\vdash^{\mathsf{bit}}_{12}$ to derive $\mathcal{K}(k)$, from $\mathcal{K}(b)$, $b \doteq k \oplus \texttt{0xdeadbeef}$ and $\mathsf{const}\ \texttt{0xdeadbeef}$.

Similarly, when we come back to Example 1, in Fig. 4 we can see how $\vdash^{\mathsf{bit}}_{12}$ helps the DY attacker derive $\mathcal{K}(m)$. As $\mathsf{len}(m)$ is a constant and $\|$ is an operation applied to m and $\mathsf{len}(m)$, the DY attacker obtains $\mathcal{K}(m)$ from $\mathcal{K}(b)$, $b \doteq m \| \mathsf{len}(m)$ and $\mathsf{const}\ \mathsf{len}(m)$. We have now addressed the issue of parsing assumptions (Sec. II-B1) in Example 1 and the loss of bit-level information (Sec. II-B2) in Example 4.

In summary, the symbolic view on composition improves the accuracy of judgment in particular when combining with the DY attacker as Examples 1, 4 and 5 witness. This is hardly surprising, as the translation approach sets up both DY attac

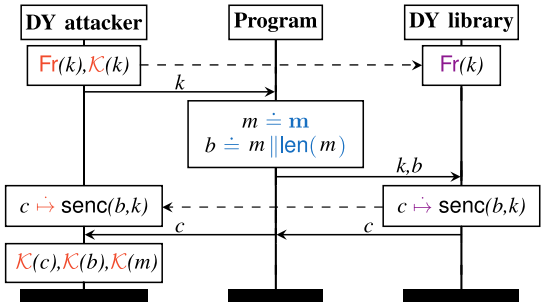


Fig. 4: A DY attacker removing bit-level masking using $\vdash^{\mathsf{bit}}_{12}$ in Example 1

and program in a concrete execution semantics with concrete (classical) composition, although the DY attacker is symbolic in nature. By instead lifting the language to the symbolic level, we turn the composition approach back on its feet and observe—at the level of the composed system—that we have two methods of deduction at our disposal. What is surprising, is that we can achieve a significant improvement with relatively simple deduction combinators. It should be difficult to find logics where one *cannot* find an equality predicate. Even a closer integration as sketched in the previous paragraph, would apply to a large set of programming languages while yielding immediate benefits.

H. Beyond DY Attackers

Besides the DY model, which is used in protocol verification, there are two other attacker models that we want to discuss in the context of this framework. The first is the *unbounded attacker* used in programming languages and system-level verification. This attacker is used in settings where cryptographic primitives are either not used at all, or where their security guarantees are built into the language semantics [25]. The unbounded attacker can be a program or program context in the same language as the program under verification, or the trace of inputs that the program interacts with. In both cases, computational limitations (even decidability) are rarely relevant to the security argument. The decoupling of the attacker is thus only interesting if the attacker is intended to communicate with multiple other components. In this case, there is no need for deduction by the unbounded attacker (each of its inputs is arbitrary, so fresh symbols) but deduction combiners can be useful for components that share information, e.g., via the attacker.

The second attacker model is the *computational attacker*, mentioned in Sec. II-B3. There, we discussed how the translation approach struggles with probabilistic choice, unless the language provides the means to draw random keys. A naive formulation of the computational attacker encodes the Turing machine semantics or any other probabilistic semantics. E.g., when a key is drawn, there are approximately 2^n possible next states, with n being the key length, each describing a different value of this key after sampling. It is clear that such a modeling has little use for verification, as the state space is enormous.

Instead, we can apply our previous argument that a symbolic ıantics for the program ought to be composed with a

symbolic semantics for the attacker and library. Thus, we should find a symbolic representation of the random process producing, e.g., a distribution over keys. Bana and Comon propose a model where symbolic rules with a computational interpretation are individually proven sound, but can be used to reason symbolically [26], [27]. It can be reasoned about interactively with the SQUIRREL prover [28]. We only sketch the idea here and leave a full realization for future work. As for the DY attacker, the computationally-complete symbolic attacker (CCSA) represents messages as terms over a set of function symbols and names, however, they are interpreted w.r.t. a security parameter. A name describes the process of sampling a random bitstring. A term describes a recursive process of evaluating each function symbol using some polynomial algorithm and sampling each name as described (but only once). In contrast to the DY model, where the function symbols define what the attacker can do (and everything else is disallowed), the CCSA model retains compatibility with the computational model by symbolically formulating what the attacker *cannot do* (and everything else is allowed). Consequently, there is no equational theory; equality is evaluated literally on the resulting bitstrings (in the interpretation). Instead, CCSA features axioms that are proven sound w.r.t. the above mentioned interpretation of terms as probabilistic polynomial-time Turing machines. The CCSA is thus simpler to define than the DY attacker: it does not retain a predicate set or an equality predicate. The predicate set, however, is the first-order logic described by Bana and Comon [26]. Scerri's decision procedure allows handling a fragment of these formulas [29], hence there is even potential for automation.

I. Correctness

The correctness of parallel composition ($\|$) is defined in terms of a partially synchronized interleaving ($|||$) of the traces of each component, i.e., a permutation of the union of trace sets that maintains the relative order of elements within each set (see extended version [24, Appendix A] for the formal definition). This is stronger than trace inclusion; for instance, it implies that all non-synchronizing traces of $\mathfrak{T}(\mathbf{M})$ or $\mathfrak{T}(\mathsf{M})$ are contained in $\mathfrak{T}(\mathbf{M}) \mid\mid\mid \mathfrak{T}(\mathsf{M})$, where $\mathfrak{T}(M)$ is the set of traces produced by an LTS M. The correctness result covers *all* events, including synchronizing events for the DY attacker, the DY library and non-synchronizing events that occur only in the program we translate. Verification methodology will typically only consider a specific subset. In our case studies, for instance, the program emits non-synchronizing events when special functions are reached and the verification tool describes security properties as trace properties over these events.

More precisely, a trace $\mathfrak{t} \in \mathfrak{T}$ is a sequence of events. We denote the symbolic parallel composition by $\|^{:}_{s}$ and traditional parallel composition for concrete systems by $\|_c$. To avoid any ambiguity, we use notation like $\mathfrak{t}_{12} \in \mathfrak{T}_{12}(\mathbf{M} \parallel \mathsf{M})$ to refer to the sequence of events produced by a composite system and $\mathfrak{T}^s$ to distinguish the set of symbolic semantics traces from the set of concrete semantics traces $\mathfrak{T}^c$.

Theorem 1 (Symbolic Composition Correctness)**.** *For any symbolic LTS* $\mathbf{M}$ *and* M*, and for any combined deduction relation* $\vdash_{12}$*:*

1) *If all predicates* $\vdash^{\mathsf{ena}}_{12}$ *produces may enable additional transitions, but not disable them, we call* $\vdash^{\mathsf{ena}}_{12}$ enabling *and* $\mathfrak{T}^s_{12}(\mathbf{M} \parallel^{\vdash^{\mathsf{ena}}_{12}}_{s} \mathsf{M}) \supseteq \mathfrak{T}^{\mathbf{s}}(\mathbf{M}) \mid\mid\mid \mathfrak{T}^{\mathsf{s}}(\mathsf{M})$.
2) *If all predicates* $\vdash^{\mathsf{dis}}_{12}$ *produces may disable transitions, but never enable new transitions, we call* $\vdash^{\mathsf{dis}}_{12}$ disabling *and* $\mathfrak{T}^s_{12}(\mathbf{M} \parallel^{\vdash^{\mathsf{dis}}_{12}}_{s} \mathsf{M}) \subseteq \mathfrak{T}^{\mathbf{s}}(\mathbf{M}) \mid\mid\mid \mathfrak{T}^{\mathsf{s}}(\mathsf{M})$.

(The interested reader may consult the full version [24, Appendix B] for the formal definitions of transition enabling and disabling.)

Proof. By induction over the length of the composed trace. The base case is trivial (no step is taken). The inductive case is proved by a case distinction over synchronous and asynchronous events. *Correctness-Enable* and *Correctness-Disable* mechanize the proof of Thm. 1's cases in HOL4. □

Thm. 1 enables compositional analysis of symbolic systems, as Lemma 1 shows. Let refinement (or security) be expressed in terms of trace inclusion. Then, if component $\mathbf{M_1}$ refines $\mathbf{M_2}$, written in the same language, and the same holds for components M_1 and M_2, then the combined system $\mathbf{M_1} \parallel^{\vdash_{12}}_{s} \mathsf{M}_1$ refines $\mathbf{M_2} \parallel^{\vdash_{12}}_{s} \mathsf{M}_2$.

Lemma 1 (Symbolic Compositional Trace Inclusion)**.** *Let* $\mathfrak{T}^{\mathbf{s}}_{\mathbf{1}}$*,* $\mathfrak{T}^{\mathbf{s}}_{\mathbf{2}}$*,* $\mathfrak{T}^{\mathsf{s}}_{1}$*, and* $\mathfrak{T}^{\mathsf{s}}_{2}$ *be the sets of traces produced by any symbolic LTS* $\mathbf{M_1}$*,* $\mathbf{M_2}$*,* M_1*, and* M_2*. If* $\mathfrak{T}^{\mathbf{s}}_{\mathbf{1}} \subseteq \mathfrak{T}^{\mathbf{s}}_{\mathbf{2}}$ *and* $\mathfrak{T}^{\mathsf{s}}_{1} \subseteq \mathfrak{T}^{\mathsf{s}}_{2}$*, then:*

- *For the empty combined deduction relation* $\emptyset$*, it holds that* $\mathfrak{T}^s_{12}(\mathbf{M_1} \parallel_s \mathsf{M}_1) \subseteq \mathfrak{T}^s_{12}(\mathbf{M_2} \parallel_s \mathsf{M}_2)$.
- *For any combined deduction relation* $\vdash_{12} \in \{\vdash^{\top}_{12}, \vdash^{\mathsf{eq}}_{12}, \vdash^{\mathsf{bit}}_{12}\}$ *(defined in Sec. II-G), it holds that* $\mathfrak{T}^s_{12}(\mathbf{M_1} \parallel^{\vdash_{12}}_{s} \mathsf{M}_1) \subseteq \mathfrak{T}^s_{12}(\mathbf{M_2} \parallel^{\vdash_{12}}_{s} \mathsf{M}_2)$.
- *For any disabling combined deduction relation* $\vdash^{\mathsf{dis}}_{12}$ *on the refined system (left) and any enabling combined deduction relation* $\vdash^{\mathsf{ena}}_{12}$ *on the abstract system (right), it holds that* $\mathfrak{T}^s_{12}(\mathbf{M_1} \parallel^{\vdash^{\mathsf{dis}}_{12}}_{s} \mathsf{M}_1) \subseteq \mathfrak{T}^s_{12}(\mathbf{M_2} \parallel^{\vdash^{\mathsf{ena}}_{12}}_{s} \mathsf{M}_2)$.

In the extended version [24, Appendix C], we instantiate Thm. 1 to enable merging and splitting DY libraries containing the same or distinct function signatures, as protocol parties often utilize different implementations for crypto libraries.

J. Refinement

While Thm. 1 and Lemma 1 are used throughout our proof in Sec. III, we need an additional theorem to carry the analysis to the concrete system semantics. This follows from the fact that both theorems only make statements about symbolic semantics traces ($\mathfrak{T}^s$) instead of concrete semantics traces ($\mathfrak{T}^c$). We, thus, need to relate the two.

Symbolic execution semantics are usually defined sound using a refinement relation, which we denote as $\sqsubseteq$. To define it, we have to assume that we have a way to apply a (component-specific) *interpretation function*, i.e., a function ι from symbolic variables to concrete values (e.g., the function utilized in [30]), to a symbolic trace. E.g., let *apply* denote

this application, $\mathfrak{t}^c \sqsubseteq \mathfrak{t}^s \Leftrightarrow \exists \iota.\ \mathfrak{t}^c = apply(\mathfrak{t}^s, \iota)$ and likewise for $\sqsubseteq$. With this notation, we describe how refinement transfers to the composed system.

Theorem 2 (Refinement)**.** *For any enabling combined deduction relation $\vdash_{12}^{\mathsf{ena}}$, any concrete LTS $\mathbf{M_c}$ and $\mathsf{M_c}$, any symbolic LTS $\mathbf{M_s}$ and $\mathsf{M_s}$, we have*

$$\frac{\mathfrak{T}^c(\mathbf{M_c}) \sqsubseteq \mathfrak{T}^s(\mathbf{M_s}) \qquad \mathfrak{T}^c(\mathsf{M_c}) \sqsubseteq \mathfrak{T}^s(\mathsf{M_s})}{\mathfrak{T}^c_{12}(\mathbf{M_c} \parallel_c \mathsf{M_c}) \sqsubseteq \mathfrak{T}^s_{12}(\mathbf{M_s} \parallel_s^{\vdash_{12}^{\mathsf{ena}}} \mathsf{M_s})}$$

where $\sqsubseteq$ is defined from apply and apply according to [24, Def. 5] in the full version [24, Appendix D].

In Thm. 2, the enabling deduction relation is to ensure broader coverage of behaviors during symbolic execution compared to concrete execution.

Proof. From the left-hand side, we apply a concrete variant of Thm. 1 ([24, Thm. 4] in the full version [24, Appendix E]) to describe the composed concrete system via interleaving. From the right-hand side, we apply Thm. 1 (case 1) itself, to obtain a similar interleaving, but of the composed symbolic system. We then use the refinements $\sqsubseteq$, $\sqsubseteq$, and $\sqsubseteq$ and instantiate the interpretation functions in $\sqsubseteq$ to map the composed traces in the concrete domain to those in the symbolic domain. See *Refinement* for proof mechanization in HOL4. □

We avoid communication between symbolic and concrete components by avoiding hybrid systems altogether, which is why we need both Lemma 1 (for abstraction within the symbolic domain) and Thm. 2 (for abstraction from the concrete to the symbolic domain).

The reader may wonder if the interpretation function ι, used in the definition of $\sqsubseteq$, would also constitute a translation of the kind we criticized. We criticize the implication of a translation at the object level, i.e., translation *within* the system between concrete programs and symbolic DY attackers *in the same system*. By contrast, the interpretation function resides at the proof level rather than the object level, and (the existence of it) is merely a constraint that the symbolic execution is a consistent abstraction. Concretely, it can be created on the fly and it is not required to be computable or consistent across multiple (symbolic) executions.

III. Instantiations of the Framework

We instantiate our framework with different languages: (a) ARMv8 and RISC-V for verifying implementations of real-world protocols, (b) SAPIC$^+$ for modeling parties from the specification, and (c) DY rules for specifying our threat model. We will consider a case study (WireGuard, see below) where we extract both parties, client and server, from ARMv8 binaries. We will also consider a case study (TinySSH), where the ARMv8 binary describes only the server, and the client behavior is described in SAPIC$^+$, thereby modeling an unknown SSH client implementation that follows the specification [31]. Both cases include a DY attacker, and the second mixes a machine-code language (case a) with a specification language (case b). Both cases are relevant, as protocol standards may not always be available, for instance, if the protocol is not wid

$$\begin{aligned}
\mathbf{P} \in \mathbf{prog} &:= \mathbf{block}^* \\
\mathbf{block} &:= (\mathbf{v}, \mathbf{stmt}^*) \\
\mathbf{v} \in \mathbf{Bval} &:= \mathit{string} \mid \mathit{int} \\
\mathbf{stmt} &:= \mathbf{halt} \mid \mathbf{jmp}(\mathbf{e}) \mid \mathbf{cjmp}(\mathbf{e}, \mathbf{e}, \mathbf{e}) \\
&\quad \mid \mathbf{assign}(\mathit{string}, \mathbf{e}) \\
\mathbf{e} \in \mathbf{Bexp} &:= \mathbf{v} \mid \mathbf{var}\ \mathit{string} \mid \lozenge_{\mathbf{u}}\ \mathbf{e} \mid \mathbf{e}\ \lozenge_{\mathbf{b}}\ \mathbf{e}
\end{aligned}$$

Fig. 5: A fragment of the **BIR** syntax

used or not yet standardized. Vice-versa, protocol standards can be ambiguous or overly-general, so it can be interesting to consider a particular implementation.

To derive the SAPIC$^+$ model of the protocols' parties from their binary implementations, we transform their machine code into **BIR**, symbolically execute them, and translate the resulting execution trees into (a subset of) SAPIC$^+$. This section demonstrates how the theorems presented so far simplified the end-to-end proof, enabling us to mechanize it. Finally, we prove mutual authentication and forward secrecy in the symbolic model for the TinySSH and WireGuard protocols to evaluate our framework.

A. Intermediate representations

1) **The BIR Representation:** We use HolBA [32]—a binary analysis platform in HOL4—to transpile the protocols' binary into the *binary intermediate representation* (**BIR**). **BIR** is a simple and architecture-agnostic language designed to simplify the analysis tasks and is used as the internal language of HolBA to facilitate building analysis tools. The **BIR** transpiler is verified and generates a certifying theorem that guarantees that the semantics of the binary is preserved (see [32, Thm. 2]); this ensures that the analysis results on **BIR** can be transferred back to the binary. Fig. 5 shows the **BIR** syntax. A **BIR** program **P** includes a number of blocks, each consisting of a tuple of a unique label (i.e., a string or an integer) and a few statements. The label of **BIR** blocks is often used as the target of jump instructions—**jmp** or **cjmp**—and refers to a particular location in the program. The **assign** statement assigns the evaluation of a **BIR** expression to a variable, and **halt** indicates the execution termination. **BIR** expressions include constants, standard binary, and unary operators (ranged over by $\lozenge_{\mathbf{b}}$ and $\lozenge_{\mathbf{u}}$) for finite integer arithmetic. **BIR** expressions also include memory operations and conditionals, which we leave out to simplify the presentation and because they are unnecessary in our evaluation.

We use a proof-producing symbolic execution for **BIR** [30] that formalizes the symbolic generalization of **BIR** (hereafter **SBIR**) to find all execution paths of the program. The symbolic semantics aligns with the concrete semantics, enabling guided execution while ensuring a consistent set of reachable states from an initial symbolic state. The set of **SBIR** events is the disjoint union of the set of non-synchronizing events and the set of synchronizing events. The set of synchronizing events encompasses $SFr(n)$ for the secret name $n \in \mathcal{N}_{\mathtt{priv}}$, $A2P(x)$ and $P2A(x)$ for the symbol $x \in \Sigma$, $FCall(f, x_1, \ldots, x_n, y)$ for the function symbol $f \in \mathcal{F}^n$

and symbols $x_1, \ldots, x_n, y \in \Sigma$. The set of non-synchronizing events includes $\mathbf{Ev}(e)$ to indicate the release of a visible event e, **Loop** to denote initiating a loop, and $\mathbf{Asn}(x, \mathbf{e})$ to signify assigning the **BIR** expression $\mathbf{e}$ to the symbol x. Moreover, a sequence of events $\alpha^{\mathbf{s}}{}_{\mathbf{1}}, \ldots, \alpha^{\mathbf{s}}{}_{\mathbf{m}}$ signifies a **SBIR** trace $\mathfrak{t}^{\mathbf{s}} \in \mathfrak{T}^{\mathbf{s}}$ such that $\mathfrak{t}^{\mathbf{s}} = \alpha^{\mathbf{s}}{}_{\mathbf{1}}, \ldots, \alpha^{\mathbf{s}}{}_{\mathbf{m}}$.

Fig. 6 illustrates a sequence of **BIR** statements of the program in Example 4. Note that the **BIR** representation is simplified w.r.t. to the implementation in HolBA. When our symbolic execution engine evaluates each of these **BIR** statements, a logical predicate may be added into the **SBIR** predicate set, denoted as $\Pi_{\mathbf{s}}$, and an **SBIR** event arises (i.e., in the exact way that the CRYPTOBAP performs symbolic execution [8, Sec. 5]). For example, an equality predicate, represented as $\doteq$, is added to the **SBIR** predicate set as a result of processing the **assign** statement. Additionally, the DY attacker's predicate set (i.e., Π_{A}) is updated due to the combined deduction relation $\vdash^{\dot{\mapsto}}_{LA}$ and synchronization (Table I summarizes synchronization events). The parallel composition of **SBIR** and the DY attacker employs the combined deduction relation $\vdash^{\mathsf{bit}'}_{\mathbf{s}A}$, which represents a specialized variant of the combined deduction relation $\vdash^{\mathsf{bit}}_{12}$ for **SBIR**, as presented below:

$$\Pi_{\mathbf{s}} \uplus \Pi_{\text{A}} \vdash^{\mathsf{bit}'}_{\mathbf{s}A} \mathcal{K}(z) \Leftrightarrow \exists\, x, y, w. \\ \mathcal{K}(y) \in \Pi_{\text{A}} \wedge (y \doteq x \diamondsuit_{\mathbf{b}} w) \in \Pi_{\mathbf{s}} \wedge \\ z \in \big(\mathit{symbols}(x) \cup \mathit{symbols}(w)\big) \quad (\text{bit}')$$

As shown in the last column of Fig. 6, the DY attacker gains further logical facts by using the deduction combiner $\vdash^{\mathsf{bit}'}_{\mathbf{s}A}$ together with the DY and **SBIR** predicate sets. We use a number next to each piece of knowledge, to indicate in which order these facts are acquired in our running example.

2) **The SAPIC$^+$ Representation:** SAPIC$^+$ is an applied pi calculus similar to PROVERIF and SAPIC [33]. SAPIC$^+$ extends SAPIC with the definition of destructors, let bindings with pattern matching and else branches. SAPIC$^+$ provides a common language that (soundly) translates to both PROVERIF and TAMARIN [12]. Fig. 7 presents a fragment of the SAPIC$^+$'s syntax. The new construct creates fresh values, and in and out receives and sends messages over the channel. The event construct raises events that security properties can refer to, but otherwise does not change the execution. They will be used to capture event functions. SAPIC$^+$ contains the non-deterministic choice operator, denoted as $+$ and introduced in [34]. A process $\mathsf{P} + \mathsf{Q}$ can either move as if it were P, or as if it were Q. SAPIC$^+$ syntax includes *stateful* processes [12] that manipulate globally shared states, i.e., some database, register or memory that can be read and altered by different parallel threads. As we skipped the model extraction of memory manipulation primitives, the stateful processes are omitted in Fig. 7.

SAPIC$^-$ has the same syntax as SAPIC$^+$, but its semantics remove the DY attacker: instead of invoking SAPIC$^+$'s internal DY deduction relation, communication (in, out) in SAPIC$^-$ emits events ($A2P$, $P2A$) that synchronizes with an outside attacker. Since SAPIC$^+$ uses the event K to signify messages coming from the attacker instead of $A2P$, we use $(\!|\cdot|\!)$ translate between trace (sets) of SAPIC$^+$ and SAPIC$^-$/**SBIR**. Besides K, security properties in SAPIC$^+$ can only refer to events in the process, which $(\!|\cdot|\!)$ keeps the same. For a given SAPIC$^-$ process P, $\mathfrak{T}^{sp}(\mathsf{P})$ denotes the set of all possible symbolic traces generated by process P. We define a SAPIC$^-$ trace $\mathfrak{t}^{sp} \in \mathfrak{T}^{sp}$ as a sequence of events such that $\mathfrak{t}^{sp} = \alpha^{sp}{}_1 \ldots \alpha^{sp}{}_m$. In Sec. III-D, we combine SAPIC$^-$ with the DY attacker and library to SAPIC$^+$. As a by-product, this shows the correctness of both w.r.t. the DY semantics in SAPIC$^+$ (which are quite standard).

B. From SBIR To SAPIC$^-$

Using symbolic execution, we derive the execution tree $\mathbf{T}$ of a **BIR** program, which is used to extract the SAPIC$^-$ model. The leaves in $\mathbf{T}$ are due to the **BIR halt** statement that marks the end of a complete path. A node in $\mathbf{T}$ is either a branching node $\mathbf{Branch}(\mathbf{pc}, \mathbf{e}, \mathbf{T_1}, \mathbf{T_2})$, where $\mathbf{pc}$ locates the conditional statement in the program, $\mathbf{e}$ is the condition, and $\mathbf{T_i}$ are the sub-trees for $\mathbf{i} \in \{\mathbf{1}, \mathbf{2}\}$; or an event node $\mathbf{node}(\mathbf{pc}, \mathbf{ev}) :: \mathbf{T}'$ with the sub-tree $\mathbf{T}'$ and $\mathbf{pc}$ specifying where the event $\mathbf{ev}$ occurred. In $\mathbf{T}$, an edge connects two nodes if they are in the transition relation.

We construct $\mathbf{T}$ from a **BIR** program and an initial symbolic state, with the root representing the initial state. The symbolic execution provides us with up to two successor states for any node. We obtain two successor states if the node represents a branching statement (i.e., **cjmp**). In such cases, the condition of the statement is stored in a branching node, and we proceed to construct subtrees from the two successor states. If the node represents any other statement, an event node is recorded with one or no successor tree.

The protocol model is obtained by translating $\mathbf{T}$ into its SAPIC$^-$ model. We translate $\mathbf{T}$ using the rules in Fig. 8 to $[\![\mathbf{T}]\!]$. We translate leaves into a *nil* process 0, and the event $\mathbf{ev}$ from the event nodes into their corresponding SAPIC$^-$ construct. The branching nodes of $\mathbf{T}$ (i.e., $\mathbf{Branch}(\mathbf{pc}, \mathbf{e}, \mathbf{T_1}, \mathbf{T_2})$) are translated into a non-deterministic choice $(+)$ between (the translation of) both possible paths. If these branches are not already pruned by symbolic execution, there might still be bit-level conditions that are relevant for the protocol verifier, but not sufficient to prune the branch during symbolic execution. While we did not encounter this case, it would be possible to translate to $(\text{event } E_1; [\![\mathbf{T_1}]\!] + \text{event } E_2; [\![\mathbf{T_2}]\!])$ for E_1, E_2 some unique events. As SAPIC$^+$ supports restricting the trace set based on formulas, we can reflect necessary conditions that are expressible in these tools. For instance, if the condition was $x \oplus y = \mathbf{0}$, we may require that the occurrence of E_1, i.e., a traversal into the positive branch, entails that $y \neq x+1$, if that helps exclude a false attack. In all our case studies, most paths are pruned by our symbolic execution engine and the remaining not require such a refinement. Nevertheless, this feature would be easy to add (and prove correct for any condition entailed by the combined deduction relation).

In order to illustrate the methodology employed for model extraction, the extracted SAPIC$^-$ process of the **BIR** program from Example 4 is presented in Fig. 6. For technical details ut the lifting of the binary and the symbolic execution, we

BIR Statement	SBIR Predicate	SBIR Event	SAPIC$^-$ Process	DY Predicate	Via $\vdash^{bit'}_{sA}$
0 `[R30=1;]jmp(0x44)//rng`	–	$SFr(k)$	new k;	$\mathsf{Fr}(k)$	–
1 `assign(R1, var(R0))`	$R1 \doteq \mathbf{var}(k)$	$\mathbf{Asn}(R1, \mathbf{var}(k))$	let $R1 = k$ in	–	$\mathcal{K}(k)$ (2)
2 `assign(R0, m)`	$R0 \doteq \mathbf{m}$	$\mathbf{Asn}(R0, \mathbf{m})$	let $R0 = m$ in	–	–
3 `[R30=4;]jmp(0x20)//senc`	$c \mapsto \mathsf{senc}(R0, R1)$	$FCall(\mathsf{senc}, R0, R1, c)$	let $c = \mathsf{senc}(R0, R1)$ in	$c \mapsto \mathsf{senc}(R0, R1)$	–
4 `[R30=5;]jmp(0x04)//send`	–	$P2A(c)$	out(c);	$\mathcal{K}(c)$	$\mathcal{K}(R0)$ (3)
5 `assign(R2, R1⊕0xde..)`	$R2 \doteq R1 \oplus \mathbf{0xde..}$	$\mathbf{Asn}(R2, R1 \oplus \mathbf{0xde..})$	let $R2 = \mathsf{xor}(R1, \text{'}de..\text{'})$ in	–	–
6 `[R30=7;]jmp(0x04)//send`	–	$P2A(R2)$	out($R2$);	$\mathcal{K}(R2)$	$\mathcal{K}(R1)$ (1)

Fig. 6: The sequence of BIR statements of Example 4, along with the corresponding updates resulting from symbolic execution, model extraction and the deduction combiner $\vdash^{bit'}_{sA}$. '$de..$' represents the constant value **0xde..**. Jumps (at lines 0, 3, 4, and 6) are the translation of *branch and link* instruction used for function calls in ARM, which requires updating the *link register* $R30$. We present this register update in [..] to mean that it is not relevant to what we intend to present in this example.

$$
\begin{array}{lll}
\langle \mathsf{P}, \mathsf{Q} \rangle & ::= & \\
\quad 0 & & \mid\ !\mathsf{P} \\
\mid\ \mathsf{in}(x);\ \mathsf{P} & & \mid\ \mathsf{P} \mid \mathsf{Q} \\
\mid\ \mathsf{out}(x);\ \mathsf{P} & & \mid\ \mathsf{P} + \mathsf{Q} \\
\mid\ \mathsf{event}\ e;\ \mathsf{P} & & \mid\ \mathsf{new}\ n;\ \mathsf{P} \\
\mid\ \mathsf{let}\ t_1 = t_2\ \mathsf{in}\ \mathsf{P}\ \mathsf{else}\ \mathsf{Q} & &
\end{array}
$$

Fig. 7: A fragment of the syntax of SAPIC$^+$ process calculus. In this figure, $e, t_1, t_2 \in \mathcal{T}$, $n \in \mathcal{N}_{\mathrm{priv}}$, $x \in \mathcal{V}$.

$$
\begin{array}{lll}
\mathbf{T} = \mathbf{Leaf} \mid \mathbf{node}(\mathbf{pc}, \mathbf{ev}) :: \mathbf{T}' \mid \mathbf{Branch}(\mathbf{pc}, \mathbf{e}, \mathbf{T_1}, \mathbf{T_2}) & & \text{event tree} \\
[\![\mathbf{Leaf}]\!] & \mapsto\ 0 & \\
[\![\mathbf{node}(\mathbf{pc}, \mathbf{ev}) :: \mathbf{T}']\!] & := & \text{events nodes} \\
\quad [\![\mathbf{node}(\mathbf{pc}, \mathbf{Ev}(e)) :: \mathbf{T}']\!] & \mapsto\ \mathsf{event}\ e; [\![\mathbf{T}']\!] & \\
\quad [\![\mathbf{node}(\mathbf{pc}, A2P(x)) :: \mathbf{T}']\!] & \mapsto\ \mathsf{in}(x); [\![\mathbf{T}']\!] & \\
\quad [\![\mathbf{node}(\mathbf{pc}, P2A(x)) :: \mathbf{T}']\!] & \mapsto\ \mathsf{out}(x); [\![\mathbf{T}']\!] & \\
\quad [\![\mathbf{node}(\mathbf{pc}, FCall(f, x_1, \ldots, x_n, y)) :: \mathbf{T}']\!] & \mapsto\ \mathsf{let}\ y = f(x_1, \ldots, x_n)\ \mathsf{in}\ [\![\mathbf{T}']\!]\ \mathsf{else}\ 0 & \\
\quad [\![\mathbf{node}(\mathbf{pc}, \mathbf{Asn}(x, \mathbf{e})) :: \mathbf{T}']\!] & \mapsto\ \mathsf{let}\ x = [\![\mathbf{e}]\!]\ \mathsf{in}\ [\![\mathbf{T}']\!] & \\
\quad [\![\mathbf{node}(\mathbf{pc}, SFr(n)) :: \mathbf{T}']\!] & \mapsto\ \mathsf{new}\ n;\ [\![\mathbf{T}']\!] & \\
\quad [\![\mathbf{node}(\mathbf{pc}, \mathbf{Loop}) :: \mathbf{T}']\!] & \mapsto\ !\ [\![\mathbf{T}']\!] & \\
[\![\mathbf{Branch}(\mathbf{pc}, \mathbf{e}, \mathbf{T_1}, \mathbf{T_2})]\!] & \mapsto\ [\![\mathbf{T_1}]\!] + [\![\mathbf{T_2}]\!] & \\
[\![\mathbf{e} \in \mathbf{Bexp}]\!] & := & \mathbf{BIR}\ \text{expressions} \\
\quad [\![\mathbf{b} \in \mathbf{Bval}]\!] & \mapsto\ \text{'}b\text{'} \in \mathcal{N}_{\mathrm{pub}} & \\
\quad [\![\mathbf{var}\ x]\!] & \mapsto\ x \in \mathcal{V} & \\
\quad [\![\phi_1 \Diamond_\mathbf{b} \phi_2]\!] & \mapsto\ [\![\phi_1]\!] [\![\Diamond_\mathbf{b}]\!] [\![\phi_2]\!] & \text{Binary operations} \\
\quad [\![\Diamond_\mathbf{b}]\!] & \mapsto \begin{cases} = & \mathbf{Equal} \\ \mathsf{plus}, \mathsf{mult}, \ldots & \mathbf{Plus}, \mathbf{Mult}, \ldots \end{cases} & \\
\quad [\![\Diamond_\mathbf{u} \phi']\!] & \mapsto\ [\![\Diamond_\mathbf{u}]\!] [\![\phi']\!] & \text{Unary operations} \\
\quad [\![\Diamond_\mathbf{u}]\!] & \mapsto \begin{cases} \neg & \mathbf{Not} \\ \bot & \text{otherwise} \end{cases} &
\end{array}
$$

Fig. 8: Translating execution tree $\mathbf{T}$ to SAPIC$^-$ model: $e, x, x_1, \ldots, x_n, y \in \Sigma$ are symbols, $n \in \mathcal{N}_{\mathrm{priv}}$ is a secret name, and $\mathsf{plus}, \mathsf{mult}, f \in \mathcal{F}^n$ are function symbols. Observe that $\neg(t_1 = t2) = (t_1 \neq t_2)$ for $t_1, t_2 \in \mathcal{T}$.

refer to the CRYPTOBAP [8] paper, which has introduced this method for model extraction from the binaries, but without a mechanized proof. Our focus here is the end-to-end proof, which builds on the framework from the previous section, the wider range of target backends provided by SAPIC$^+$.

C. Translation correctness

To enable transferring verified properties from the SAPIC$^+$ level back to BIR and then to the protocols' binary, it is essential to prove that the extracted SAPIC$^-$ model preserves the behaviors of the SBIR representation. To this end, we establish a proof that for every path in the symbolic execution tree $\mathbf{T}$, there exists an equivalent SAPIC$^-$ trace derived from executing translated process $[\![\mathbf{T}]\!]$.

Theorem 3 (Trace Inclusion)**.** *Let* $\mathbf{T}$ *be a* SBIR *execution tree. Then, all translated* SBIR *traces of* $\mathbf{T}$, $(\!|\mathfrak{T}^s(\mathbf{T})|\!)$, *are included in the traces of the translated* SAPIC$^-$ *process* $\mathfrak{T}^{sp}([\![\mathbf{T}]\!])$.

Proof. The proof is done by induction on the length of the translated traces $(\!|\mathfrak{T}^s(\mathbf{T})|\!)$. In the base case, no actions are taken. For the inductive case, we apply the case distinction over synchronous and asynchronous events in the set of SBIR events. We mechanized Thm. 3's proof in HOL4 (see *Symbtree-to-Sapic*). □

Lindner et al. [30, Thm. 4.1] demonstrated that verified properties for SBIR transfer to BIR, ensuring that the verified properties hold for concrete execution semantics.

D. End-to-end correctness result

We then show how the theorems in Sec. II come together to simplify the analysis of our target language, which we will equip with DY semantics. Our analysis below includes embedded links to the mechanized proof for each step. We start with the concrete, complete ARMv8 program in parallel with an unspecified attacker A.

$$\mathfrak{T}^c(C^{ARMv8} \parallel_c A)$$

As is often the case, we take a detour via an intermediary language, in our case, BIR. [32, Thm. 2] justifies this so-called *lifting* step, i.e., shows that this translation is semantics preserving. Thanks to [24, Corollary 2] in the extended version [24, Appendix E], we can use this theorem in context with A.

$$= \mathfrak{T}^c(C^{\mathbf{BIR}} \parallel_c A)$$

We next require (Assumption 1) that $C^{\mathbf{BIR}}$ is trace-equivalent to $P^{\mathbf{BIR}} \parallel_c L^{\mathbf{BIR}}$, i.e., that it can be split into a program-under-verification and a known library. [8, Sec. 4] provides statically checkable criteria for **BIR** to verify this condition automatically. Again, [24, Corollary 2] is used to apply this in context. Afterwards, we use the refinement theorem Thm. 2 and the relations indicated by the underbraces to move from the concrete to the symbolic. An interpretation function ι evaluates **SBIR** symbolic expressions to **BIR** concrete values, as demonstrated in [30]. Because $\parallel_c$ is associative w.r.t. trace equivalence, we have:

$$= \mathfrak{T}^c(\underbrace{P^{\mathbf{BIR}}}_{\text{[30, Thm. 4.1]}} \parallel_c \underbrace{L^{\mathbf{BIR}} \parallel_c A}_{\text{A2: Deduction Soundness}})$$

$$\sqsubseteq \mathfrak{T}^s(\overbrace{P^{\mathbf{SBIR}}}^{\vdash^{\mathsf{bit}'}_{\mathbf{s}A}} \parallel_s \overbrace{L^{DY} \parallel_s}^{\vdash^{\mapsto}_{LA}} A^{DY})$$

The first relation is the soundness of symbolic execution. The second is an assumption on the attacker that we will talk about in a second. Recall that $\parallel_s^{\vdash^{\mapsto}_{LA}}$ uses a deduction combiner specific to the DY attacker and library, while $\parallel_s^{\vdash^{\mathsf{bit}'}_{\mathbf{s}A}}$ utilizes a specialized deduction relation between **SBIR** and DY as defined in Sec. III-A1. Next, $\parallel_s^{\vdash^{[\![\mathsf{bit}']\!]}_{spA}}$ employs a deduction relation similar to $\vdash^{\mathsf{bit}'}_{\mathbf{s}A}$, referred to as $\vdash^{[\![\mathsf{bit}']\!]}_{spA}$, which particularly applies to SAPIC$^-$ and DY predicate sets. The distinction from $\vdash^{\mathsf{bit}'}_{\mathbf{s}A}$ lies in the fact that $\vdash^{[\![\mathsf{bit}']\!]}_{spA}$ incorporates the translation of the binary operators $\Diamond_{\mathbf{b}}$ as function symbols (see Fig. 8 for the translation of the binary operations). We use Lemma 1 (case 3) to apply our translation result from **SBIR** to SAPIC$^-$ (Thm. 3) that we showed in the previous subsection (note that $P^{\text{SAPIC}^-} = [\![P^{\mathbf{SBIR}}]\!]$, $\vdash^{\mathsf{bit}'}_{\mathbf{s}A}$ is disabling, and $\vdash^{[\![\mathsf{bit}']\!]}_{spA}$ is enabling). We have:

$$\subseteq \mathfrak{T}^s(P^{\text{SAPIC}^-} \parallel_s^{\vdash^{[\![\mathsf{bit}']\!]}_{spA}} L^{DY} \parallel_s^{\vdash^{\mapsto}_{LA}} A^{DY})$$

SAPIC$^+$'s semantics include the DY attacker and library, hence, the above system.

$$= \mathfrak{T}^s(P^{\text{SAPIC}^+})$$

We thus have an end-to-end correctness result. Thanks to the framework theorems, this proof can be adapted to many other languages, as the researcher needs to only show the correctness of the language-specific steps (correctness of lifting, splitting, and translation) when adapting. Moreover, they only need to be shown in isolation. Until now, the translation step was usually shown in the presence of the adversary [8], [9], [33].

While Assumption 1 delineates the class of programs that is supported (and can be checked statically), Assumption 2 (short: A2) formalizes our threat model: whatever type of system the attacker controls, it can be abstracted as a DY attacker if we also abstract the library in the same way. We can leave it at that, but we believe that this assumption merits deeper exploration, as discussed further in the full version [24, Appendix F].

In the extended version [24, Appendix G], we extend this proof to two parties (client and server) and an unbounded number of copies thereof.

E. Verification of TinySSH and WireGuard

We have verified the TinySSH and WireGuard protocols to evaluate our framework. Our case studies demonstrate that our methodology does not introduce any artifacts that inhibit verification. Table II shows data we have collected during our evaluation. The LOC of ARM assembly represents the complete assembly code, including crypto functions, which were necessary to consider in our preprocessing step to compute the program's control flow.

The binary of our case studies is unaltered; however, the verifier must manually initialize and steer the verification process. Specifically, the user is required to specify: (a) to the lifter, the code fragments to be analyzed, (b) to the symbolic execution engine, which operates on the output of the lifter, i.e., **BIR** code, the function names grouped as trusted (libraries) or untrusted (network), (c) to the symbolic execution engine, the symbolic model of the cryptographic functions, and (d) the assumptions regarding the crypto primitives and the security properties we proved for our case studies in the SAPIC$^+$ input file.

TinySSH is a minimalistic SSH server that implements a subset of SSHv2 features and ships with its own crypto library. To establish authentication requirements for any parties connecting to TinySSH, we used SAPIC$^-$ to model the client of the SSH protocol. We also automatically extracted the server model of TinySSH from its ARMv8 machine code, hence covering a system composed of three components written in three very different languages, in ARMv8, SAPIC$^-$ and DY rules. We verified mutual authentication [35] and forward secrecy [36] with PROVERIF and TAMARIN.

WireGuard implements virtual private networks akin to IPSec and OpenVPN. It is quite recent and was incorporated into the Linux kernel. We focused on the handshake protocol of WireGuard instead of the record protocol, as the handshake is usually considered more challenging. We have extracted, *for the first time*, the SAPIC$^-$ model of the Linux kernel's WireGuard implementation binary. Our model is more faithful than existing manual models which, for instance, use pattern matching for authentication verification and was extracted automatically. Having extracted the handshake and the first message transmitted upon the completion of exchanging keys, we prove that the protocol participants mutually agree on the resulting keys in both PROVERIF and TAMARIN. Moreover, we show the resulting keys remain unknown to the attacker by proving the forward secrecy property using PROVERIF and TAMARIN.

We employed the combined deduction relations $\vdash^{\mathsf{bit}'}_{\mathbf{s}A}$ and $\vdash^{[\![\mathsf{bit}']\!]}_{spA}$ in our case studies and extracted a model from the **BIR** program in Example 4 using our toolchain to demonstrate their application. The models reflect $\vdash^{\mathsf{bit}'}_{\mathbf{s}A}$ and $\vdash^{[\![\mathsf{bit}']\!]}_{spA}$ as destructors defined in the translation from SAPIC$^-$ (and Co.) to SAPIC$^+$. These destructors derive the same terms that $\vdash^{\mathsf{bit}'}_{\mathbf{s}A}$ derives in Fig. 6. Additionally, as we extracted formal models of TinySSH and WireGuard from their respective implementations, we have identified no instances in which the DY attacker could acquire additional knowledge through the of these combined deduction relations.

Protocol		ARM Loc	Verified Code Size	Feasible Path	Infeasible Path	SAPIC⁻ Loc	TAMARIN Loc	PROVERIF Loc	Time (seconds) SBIR	Time (seconds) SAPIC⁻	Time (seconds) TM	Time (seconds) PV	Verified in	Primitives
TinySSH		18K	0.476K	136	1223	204	107	117	120	493	7.32	0.114	TM & PV	DHKA, SE, DS, HF
WG	Initiator	27K	1.323K	68	1482	260	150	181	60	67	1.28	13.266	TM & PV	DHKA, AEAD, HF
	Responder			153	1389	380			60	50				

TABLE II: Case studies. Abbreviations used: WG (WireGuard), DHKA (Diffie-Hellman Key Agreement), SE (Symmetric Encryption), DS (Digital Signatures), HF (Hash Functions), and AEAD (Authenticated Encryption with Additional Data). We report the runtime for preprocessing and symbolic execution (SBIR), construction of the symbolic tree plus model extraction (SAPIC⁻), and for verification using TAMARIN (TM) and PROVERIF (PV).

Papers	Model Origin	Attacker Model	No Parsing Assum.	Formalized
Sprenger et al. [2]	Required	DY	✗	Isabelle/HOL
Arquint et al. [3], [4]	Required	DY	✗	—
Hahn et al. [37]	Required	Comp.	✓	—
Sammler et al. [5]	Required	Unbounded	✗	Coq
Bhargavan et al. [22]	Code-based	DY	✓	F^*
Wallez et al. [23]	Code-based	DY	✓	F^*
Bhargavan et al. [38], [39]	Extracted	DY / Comp.	✗	—
Aizatulin et al. [9]	Extracted	DY / Comp.	✗	—
Nasrabadi et al. [8]	Extracted	DY / Comp.	✗	—
This work	Extracted	DY	✓	HOL4

TABLE III: Selected approaches; Comp = Computational, No Parsing Assum. = No strict parsing assumptions (see Sec. II-B)

PROVERIF and TAMARIN exhibit significantly different verification times. For WireGuard, TAMARIN verifies our properties in 1.28 seconds, while PROVERIF takes 13.266 seconds. Conversely, for TinySSH, PROVERIF outperformed TAMARIN, completing the verification task in 0.114 seconds compared to TAMARIN's 7.32 seconds.

IV. RELATED WORK

In recent years, several techniques for verifying crypto protocol implementations have emerged. We survey those based on separation logic, model validation, wrappers, and the CompCert framework [40], in this section. Table III compares selected works and our proposed approach.

a) **Separation logic:** [2]–[4], [41]–[43] used separation logic to analyze the implementation of security protocols. Sprenger et al. [2] introduced a methodology where a protocol model is first formalized in Isabelle/HOL [44] and then translated into I/O specifications, which are verified using separation-logic based verifiers. Arquint et al. [3] extended this to the TAMARIN verifier to enable verification against TAMARIN's models. Follow-up work [4] stepped away from verifying against the specification, and directly verified the protocol properties, which are *stable under concurrency*, by building on a programming language that incorporates protocol operations and modeling the attacker in that language.

Others [2]–[4] used verifiers like Nagini [45] for Python, Gobra [46] for Go, and VeriFast [47] for Java and C. Nonetheless, the soundness of these approaches depends on the correctness of utilized verifiers, including their dependencies (e.g., Nagini relies on Viper [48]). In theory, they could be proven sound, but the languages are not ideal for formalized results. By contrast, BIR's decompilation approach already provides a formalized soundness results from lifting to symbolic execution while covering machine code produced by compiling these languages.

Throughout this line of work [2]–[4], a translation function maps between byte-level messages and DY terms (or an injective function in the other direction). Therefore, our arguments in Sec. II-B apply. However, a cursory glance suggests that our framework (i.e., the subject of Sec. II) might help with this, as their proof structure likewise consists of a refinement step, followed by decomposition and translation to a verification language and their communication model builds on a (subclass of) LTS and CSP-style parallel composition (with built-in translation).

b) **Model validation:** Several formalisms were proposed for modeling distributed systems [37], [49]–[51], including a hierarchical modeling language and the hybrid process calculus [49] that focuses on bisimulation notions and congruence results w.r.t. parallel composition. Strubbe et al. [51] introduced a technique to deal with the nondeterminism in distributed systems, which was later extended by Meseguer et al. [50] to handle the asynchrony of communications.

The methodologies in [37], [50], [51] required checking cross-system variable consistency during communication due to shared variables. This direct impact of one component's actions on others poses a challenge. In contrast, our synchronization method relies on events containing symbols. By avoiding the reuse of symbols, cross-system consistency is not a concern for us. This improves our approach's efficacy and makes it ideal for modeling distributed systems.

c) **Model extraction:** The application of our theory builds on CRYPTOBAP [8], which derives the idea of extracting protocol models via symbolic execution from Aizatulin et al. [52], [53]. Both approach build upon computational soundness, which imposes stringent requirements on the use of cryptography and protocols. Computational soundness is incredibly difficult to prove mechanically [54], which was the main motivation for our framework, as it (a) enables us to avoid the detour via computational soundness and (b) enables compositional proofs. Where both [8], [9] rely on pen-and-paper proofs in the cryptographic model, we have a mechanized end-to-end proof.

d) **DY code analysis:** Similar to our case studies, DY^* [22] permits code analysis w.r.t. a DY attacker, but for a high-level language (F^*) that allows conducting proofs using dependent types. Their DY attacker is formulated within F^*, whereas our framework results apply to a DY attacker that may compose with other languages. Proofs in their framework are internal to F^*, while we depend on the correctness of the protocol verifiers. [22, Sec. 1] discusses the trade-off in automation versus modularity. DY^* is complemented by nparse [23] which provides type combiners and lemmas to

deal with packing and unpacking. These lemmas are proven at the bit-level, solving (in many cases) the problems of *limited bit-level reasoning* and *strong parsing assumptions* mentioned in Sec.I —although we emphasize that DY^* and Comparse are not a multi-language composition, instead integrating the DY attacker as a library. Instead of constructing the format types (and their proofs of validity), we extract the formats using our symbolic execution as **BIR**-level terms. We translate those to DY terms, possibly losing bit-level message confusing attacks should the deduction combiner be incomplete. The criteria in [23, Sec. 2] could be useful to judge the soundness of this deduction combiner w.r.t. the message formats that are abstracted in this way, i.e., w.r.t. a given (set of) implementations.

e) **Wrappers:** Research on multi-language semantics has explored translation between languages using a wrapper [5], [55]–[57]. DimSum [5] is the most relevant to our work. DimSum's wrapper-based composition ($\lceil\cdot\rceil_{1\rightleftharpoons 2}$) serves as a translation tool between two components written in different languages as well as between a component and the environment. Like Igloo [2], they reason about an arbitrary number of languages communicating via events and build on CSP-style parallel composition and translate between languages. Instead, we use a shared set of symbols to denote equations and deduce relations between bitstrings in different languages. DimSum requires m^2 wrappers to facilitate communication of m languages, suffering from a complexity blow-up associated with compositional soundness. Our generic deduction combiners (Sec. II-G) can remove this burden. For computational attackers, DimSum's composition does not support probabilistic semantics and it lacks a notion of runtime bounds for attackers. As far as the DY model goes, the issues in Sec. II-B apply (e.g., there is no single suitable DY term that $\lceil\lceil \mathsf{senc}(m,k)\rceil_{\mathcal{T}\rightharpoonup BS} + \mathbf{0x1}\rceil_{\mathcal{T}\leftharpoondown BS}$ should give).

f) **CompCert:** CompCert was also used to verify the multi-language protocols at the assembly-code level [58]–[65]. Among others, [58], [59], [63], [64] achieved multi-language composition by enforcing a common interaction protocol across all languages, while [60]–[62], [65] enforce specific memory-sharing patterns, along with other restrictions, on the interaction between different components. In contrast, we neither depend on a common language nor impose any restrictions on the interaction of components. Our model uses symbols for communication and predicates over these symbols for reasoning, allowing the verification toolchain to understand this interaction.

V. Concluding Remarks

We proposed a framework for symbolic parallel composition that enables composing components operating on different atomic types. Our approach extends the state-of-the-art composition techniques, allowing efficient handling of cross-language communication. Notably, our approach avoids the need to translate incompatible base values and offers a more versatile and applicable solution. By using symbolic values for communication, our method addresses the mismatches encountered in previous translation-based approaches. This provides a more accurate representation of DY terms as symbolic abstractic

Our composition framework is multi-language in this first sense: our WireGuard case study, for instance, combines programs in the SAPIC$^-$, **SBIR**, and DY language in the same system (e.g., [24, Eq. 5] in the extended version [24, Appendix G]). Our case studies are also multi-language in a much more pragmatic sense: any language that compiles to a supported assembly language is supported, independent of the compiler, and whether it is correct.

In the future, we aim to extend our framework with probabilistic reasoning. We will extend our semantic configuration to include the probability of reaching a given state. This will allow us to reason probabilistically about the composition of non-probabilistic languages.

Acknowledgment

We thank anonymous reviewers for their insightful comments. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. We also gratefully acknowledge a gift from Intel and Amazon.

References

[1] M. Backes, R. Künnemann, and E. Mohammadi, "Computational soundness for dalvik bytecode," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*.

[2] C. Sprenger, T. Klenze, M. Eilers, F. A. Wolf, P. Müller, M. Clochard, and D. Basin, "Igloo: soundly linking compositional refinement and separation logic for distributed system verification," *Proceedings of the ACM on Programming Languages*, 2020.

[3] L. Arquint, F. A. Wolf, J. Lallemand, R. Sasse, C. Sprenger, S. N. Wiesner, D. Basin, and P. Müller, "Sound verification of security protocols: From design to interoperable implementations," in *2023 IEEE Symposium on Security and Privacy*.

[4] L. Arquint, M. Schwerhoff, V. Mehta, and P. Müller, "A generic methodology for the modular verification of security protocol implementations," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*.

[5] M. Sammler, S. Spies, Y. Song, E. D'Osualdo, R. Krebbers, D. Garg, and D. Dreyer, "DimSum: A Decentralized Approach to Multi-language Semantics and Verification," *Proceedings of the ACM on Programming Languages*, 2023.

[6] M. Abadi and P. Rogaway, "Reconciling two views of cryptography: The computational soundness of formal encryption," in *IFIP International Conference on Theoretical Computer Science*, 2000.

[7] F. Nasrabadi, R. Künnemann, and H. Nemati, "Symbolic parallel composition for verification of multi-language protocol implementations-source code," 2025. [Online]. Available: https://github.com/FMSecure/CryptoBAP

[8] ——, "Cryptobap: A binary analysis platform for cryptographic protocols," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*.

[9] M. Aizatulin, "Verifying Cryptographic Security Implementations in C Using Automated Model Extraction," Ph.D. dissertation, Open University, 2015.

[10] M. Patrignani, "Why should anyone use colours? or, syntax highlighting beyond code snippets," *arXiv*, 2020.

[11] H. development team, "Hol interactive theorem prover," 2022. [Online]. Available: https://hol-theorem-prover.org/

[12] V. Cheval, C. J. andSteve Kremer, and R. Künnemann, "Sapic+: protocol verifiers of the world, unite!" in *USENIX Security Symposium*, 2022.

[13] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The tamarin prover for the symbolic analysis of security protocols," in *Computer Aided Verification: 25th International Conference*, 2013.

[14] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules." in *14th IEEE Computer Security Foundations Workshop*, 2001.

[15] V. Cheval, S. Kremer, and I. Rakotonirina, "Deepsec: deciding equivalence properties in security protocols theory and practice," in *2018 IEEE symposium on security and privacy*.

[16] S. D. Brookes, C. A. Hoare, and A. W. Roscoe, "A theory of communicating sequential processes," *Journal of the ACM*, 1984.

[17] R. De Nicola and M. Loreti, "Multi labelled transition systems: A semantic framework for nominal calculi," *Electronic Notes in Theoretical Computer Science*, 2007.

[18] G. Plotkin, "An operational semantics for csp," in *Workshop on Logic of Programs*, 1980.

[19] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno, "Sok: Computer-aided cryptography," in *2021 IEEE symposium on security and privacy*.

[20] P. Gupta and V. Shmatikov, "Towards computationally sound symbolic analysis of key exchange protocols," in *Proceedings of the 2005 ACM workshop on Formal methods in security engineering*.

[21] M. Ammann, L. Hirschi, and S. Kremer, "DY Fuzzing: Formal Dolev-Yao Models Meet Cryptographic Protocol Fuzz Testing," in *45th IEEE Symposium on Security and Privacy*, 2024.

[22] K. Bhargavan, A. Bichhawat, Q. H. Do, P. Hosseyni, R. Küsters, G. Schmitz, and T. Würtele, "DY*: A modular symbolic verification framework for executable cryptographic protocol code," *2021 IEEE european symposium on security and privacy*.

[23] T. Wallez, J. Protzenko, and K. Bhargavan, "Comparse: Provably Secure Formats for Cryptographic Protocols," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*.

[24] F. Nasrabadi, R. Künnemann, and H. Nemati, "Symbolic parallel composition for verification of multi-language protocol implementations-extended version," *ArXiv*, 2025. [Online]. Available: https://arxiv.org/pdf/2504.06833

[25] P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov, "A probabilistic poly-time framework for protocol analysis," in *Proceedings of the 5th ACM Conference on Computer and Communications Security*, 1998.

[26] G. Bana and H. Comon-Lundh, "A Computationally Complete Symbolic Attacker for Equivalence Properties," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*.

[27] G. Bana and H. C. Lundh, "Towards unconditional soundness: Computationally complete symbolic attacker," in *International Conference on Principles of Security and Trust*, 2012.

[28] D. Baelde, S. Delaune, C. Jacomme, A. Koutsos, and S. Moreau, "An Interactive Prover for Protocol Verification in the Computational Model," in *2021 IEEE Symposium on Security and Privacy*.

[29] G. Scerri, "Proofs of security protocols revisited," Ph.D. dissertation, Ecole Normale Supérieure de Cachan, 2015.

[30] A. Lindner, R. Guanciale, and M. Dam, "Proof-producing symbolic execution for binary code verification," *arXiv*, 2023.

[31] T. Ylonen and C. Lonvick, "The secure shell (SSH) transport layer protocol," *Request for Comments (No. rfc4253)*, 2006.

[32] A. Lindner, R. Guanciale, and R. Metere, "Trabin: Trustworthy analyses of binaries," *Science of Computer Programming*, 2019.

[33] S. Kremer and R. Künnemann, "Automated analysis of security protocols with global state," *Journal of Computer Security*, 2016.

[34] M. Backes, J. Dreier, S. Kremer, and R. Künnemann, "A novel approach for reasoning about liveness in cryptographic protocols and its application to fair exchange," in *2017 IEEE European Symposium on Security and Privacy*.

[35] G. Lowe, "A hierarchy of authentication specifications," in *Proceedings 10th computer security foundations workshop*, 1997.

[36] K. Cohn-Gordon, C. Cremers, and L. Garratt, "On post-compromise security," in *2016 IEEE 29th Computer Security Foundations Symposium*.

[37] E. M. Hahn, A. Hartmanns, H. Hermanns, and J.-P. Katoen, "A compositional modelling and analysis framework for stochastic hybrid systems," *Formal Methods in System Design*, 2013.

[38] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse, "Verified interoperable implementations of security protocols," *ACM Transactions on Programming Languages and Systems*, 2008.

[39] K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu, "Cryptographically verified implementations for tls," in *Proceedings of the 15th ACM conference on computer and Communications security*, 2008.

[40] X. Leroy, "Formal certification of a compiler back-end or: programming a compiler with a proof assistant," in *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006.

[41] I. Sergey, J. R. Wilcox, and Z. Tatlock, "Programming and proving with distributed protocols," *Proceedings of the ACM on Programming Languages*, 2017.

[42] N. Koh, Y. Li, Y. Li, L.-y. Xia, L. Beringer, W. Honoré, W. Mansky, B. C. Pierce, and S. Zdancewic, "From c to interaction trees: specifying, verifying, and testing a networked server," in *Proceedings of the ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2019.

[43] W. Oortwijn and M. Huisman, "Practical abstractions for automated verification of message passing concurrency," in *Integrated Formal Methods: 15th International Conference*, 2019.

[44] T. N. L. C. Paulson and M. Wenzel, "A proof assistant for higher-order logic," 2013. [Online]. Available: https://isabelle.in.tum.de/

[45] M. Eilers and P. Müller, "Nagini: a static verifier for python," in *Computer Aided Verification: 30th International Conference*, 2018.

[46] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller, "Gobra: Modular specification and verification of go programs," in *Computer Aided Verification: 33rd International Conference*, 2021.

[47] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens, "Verifast: A powerful, sound, predictable, fast verifier for c and java." *NASA Formal Methods*, 2011.

[48] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016*.

[49] E. Brinksma, T. Krilavičius, and Y. S. Usenko, "Process algebraic approach to hybrid systems," *IFAC Proceedings Volumes*, 2005.

[50] J. Meseguer and R. Sharykin, "Specification and analysis of distributed object-based stochastic hybrid systems," in *Hybrid Systems: Computation and Control: 9th International Workshop*, 2006.

[51] S. N. Strubbe and A. van der Schaft, "Compositional modelling of stochastic hybrid systems," *Cassandras and Lygeros*, 2006.

[52] M. Aizatulin, A. D. Gordon, and J. Jürjens, "Extracting and verifying cryptographic models from C protocol code by symbolic execution," in *Proceedings of the 2011 ACM Conference on Computer and Communications Security*.

[53] ——, "Computational verification of C protocol implementations by symbolic execution," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*.

[54] A. Lochbihler, "Probabilistic functions and cryptographic oracles in higher order logic," in *Programming Languages and Systems: 25th European Symposium on Programming*, 2016.

[55] J. Matthews and R. B. Findler, "Operational semantics for multi-language programs," *ACM SIGPLAN Notices*, 2007.

[56] A. Ahmed and M. Blume, "An equivalence-preserving cps translation via multi-language semantics," in *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, 2011.

[57] M. S. New and A. Ahmed, "Graduality from embedding-projection pairs," *Proceedings of the ACM on Programming Languages*, 2018.

[58] T. Ramananandro, Z. Shao, S.-C. Weng, J. Koenig, and Y. Fu, "A compositional semantics for verified separate compilation and linking," in *Proceedings of the 2015 Conference on Certified Programs and Proofs*.

[59] G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel, "Compositional compcert," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2015.

[60] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S.-C. Weng, H. Zhang, and Y. Guo, "Deep specifications and certified abstraction layers," *ACM SIGPLAN Notices*, 2015.

[61] R. Gu, Z. Shao, J. Kim, X. Wu, J. Koenig, V. Sjöberg, H. Chen, D. Costanzo, and T. Ramananandro, "Certified concurrent abstraction layers," *ACM SIGPLAN Notices*, 2018.

[62] Y. Wang, P. Wilke, and Z. Shao, "An abstract stack based approach to verified compositional compilation to machine code," *Proceedings of the ACM on Programming Languages*, 2019.

[63] Y. Song, M. Cho, D. Kim, Y. Kim, J. Kang, and C.-K. Hur, "Compcertm: Compcert with c-assembly linking and lightweight modular verification," *Proceedings of the ACM on Programming Languages*, 2019.

[64] J. Koenig and Z. Shao, "Compcerto: compiling certified open c components," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021.

[65] A. Oliveira Vale, P.-A. Melliès, Z. Shao, J. Koenig, and L. Stefanesco, "Layered and object-based game semantics," *Proceedings of the ACM on Programming Languages*, 2022.