

CRYPTOBAP: A Binary Analysis Platform for Cryptographic Protocols

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

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

Hamed Nemati
CISPA Helmholtz Center for
Information Security
hamed.nemati@cispa.de

ABSTRACT

We introduce CRYPTOBAP, a platform to verify weak secrecy and authentication for the (ARMv8 and RISC-V) machine code of cryptographic protocols. We achieve this by first transpiling the binary of protocols into an intermediate representation and then performing a crypto-aware symbolic execution to automatically extract a model of the protocol that represents all its execution paths. Our symbolic execution resolves indirect jumps and supports bounded loops using the loop-summarization technique, which we fully automate. The extracted model is then translated into models amenable to automated verification via ProVerif and CryptoVerif using a third-party toolchain. We prove the soundness of the proposed approach and used CRYPTOBAP to verify multiple case studies ranging from toy examples to real-world protocols. TinySSH, an implementation of SSH, and WireGuard, a modern VPN protocol.

This paper uses colors to distinguish between different abstraction layers in our modeling and verification [63].

CCS CONCEPTS

• Security and privacy → Logic and verification.

KEYWORDS

Formal Verification, Crypto. Protocols, Security, Binary Analysis

ACM Reference Format:

Faezeh Nasrabadi, Robert künnemann, and Hamed Nemati. 2023. CRYPTOBAP: A Binary Analysis Platform for Cryptographic Protocols. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*, November 26–30, 2023, Copenhagen, Denmark. ACM, New York, NY, USA, 23 pages. <https://doi.org/10.1145/XXXXXX.XXXXXX>

1 INTRODUCTION

Cryptographic protocols are a vital part of end-user security on the Internet. Therefore, devising techniques to obtain high-assurance guarantees about their correctness and security is highly desirable. There have been many attacks targeting either the design

of such protocols (e.g., man-in-the-middle attacks like the triple-handshake attack on TLS [15]) or their implementation (e.g., Heartbleed, CVE-2014-0160).

Formal methods suggest a rigorous foundation to find bugs and provide guarantees about the correctness and security of cryptographic protocols. Rigorous techniques that have been used so far to reason about such protocols belong to two main lines of research. One is based on the *Dolev-Yao model*, a symbolic model of cryptography [31], while the other, the *computational approach* [37], is closer to reality and gives stronger and more realistic security assurance.

In recent years, significant progress has been made using these techniques to derive rigorous guarantees based on either abstract protocol specifications or the concrete implementations in high-level programming languages such as *C* or *F#* [3, 4, 6, 16, 27, 38]. Despite this, there still exists a large gap between the correctness of cryptographic protocols' models or high-level implementations and their object code (i.e., machine code generated by compilers) that ultimately will execute on hardware. This gap may result in a lack of security, even when correctness proofs are developed for a model or a high-level implementation of the protocol. A major cause for this gap is the fact that program behavior at the source level can diverge from its actual behavior when executed on hardware, e.g., due to compiler-introduced bugs [73]. This is because enabling aggressive compiler optimizations can lead to missing source-level checks, e.g., code intended to detect integer overflows [69] or null pointers [73]. Compilers can invalidate code for secret scrubbing [66] or even turn constant-time code into a nonconstant-time binary [67]. An example is depicted in Fig. 1 where the compiler (GCC ARM V11.2.1) removes the `memset` function used to erase the secret key from memory. If such a function is part of a protocol specification, its removal can potentially leak the secret key. Alas, even verifying compilers like CompCert [50] are not a cure-all—verifying all optimization stages that developers want to enjoy in practice is tough.

Protocol verification obtains privacy and authenticity guarantees for an abstract model of the protocol that emphasizes concurrency and communication, typically abstracting cryptographic primitives with a fixed set of computation rules for the attacker, known as the Dolev-Yao model [31]. On the other hand, *program verification* obtains functional guarantees but also confidentiality in a program model that is typically sequential and focused on a single machine. The attacker is not bound by computation rules or runtime restrictions. Several works have adopted features from one domain in the other—we discuss them in detail in Sec. 2—but this comes at the cost of complex, inflexible execution models and high development cost. We advocate for *composing* these techniques, thereby bridging

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '23, November 26–30, 2023, Copenhagen, Denmark.

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0050-7/23/11...\$15.00
<https://doi.org/10.1145/XXXXXX.XXXXXX>

C Code	Simplified Assembly
<pre> fun : char K[64]; // secret key if (getKey(K, sizeof(K))) { do something } memset(K, 0, sizeof(K)); ... </pre>	<pre> fun : mov w1, 64 mov x0, #key K address bl getKey cbz w0, .L3 do something .L3: ret </pre>

Figure 1: An example of how compilers can invalidate code for secret scrubbing

the gap between tools and verification technologies that can thus continue to evolve independently.

Our goal in this paper is to extend the verification of security protocols to their machine code. This eliminates the need for trusting compilers and provides a higher assurance about the security and correctness of protocols. To achieve this, we extend HolBA [53] for the verification of RISC-V and ARM binaries with a method for *symbolic execution* that handles interactions with an arbitrary attacker and trusted cryptographic code. Our symbolic execution resolves indirect jumps and supports (compile-time) bounded loops using the summarization technique [68], which we *fully automate*. We also devise a sound translation from symbolic execution to an intermediate layer that is amenable to automated verification with the protocol verifiers ProVerif [20, 70] and CryptoVerif [21]. Fig. 2 depicts the pipeline of the CRYPTO-BAP.

Implementation-level vulnerabilities like the stack-based buffer overflow in Sami FTP Server 2.0.1 are then covered by the symbolic execution, while protocol-level vulnerabilities like the triple-handshake attack on TLS [15] are detected by those protocol verifiers. This is guaranteed by our soundness results and has been evidenced by the discovery of two flaws in CSur and NSL. If there is an error in the implementation of the protocol specification, it could either lead to a model that deviates from the intended behavior, or an error during the extraction process. For instance, if a cryptographic value is erroneously copied within the program, the symbolic execution precisely tracks the memory and detects any subsequent library call operating on the wrongly formatted data. As a result, abstraction would fail, indicating an implementation error; the failure is in the sense that the abstract operations do not apply to incorrectly encoded cryptographic data.

Our method is inspired by and builds on a line of work by Aizatulin et al. [3, 4]; Fig. 2 highlights differences between the two approaches (see Sec. 2 for more details on difference of the two approaches.) We adopt their process calculus *intermediate model language* (IML¹) and translation from IML to ProVerif and CryptoVerif. However, we extract the IML model from the machine code of protocols, providing more reliable security guarantees that are independent of the compiler used to generate the object code of protocols. We demonstrate the effectiveness of our approach by covering the case studies of Aizatulin et al. [4], including the CSur case study (which they could not verify due to limitations of their

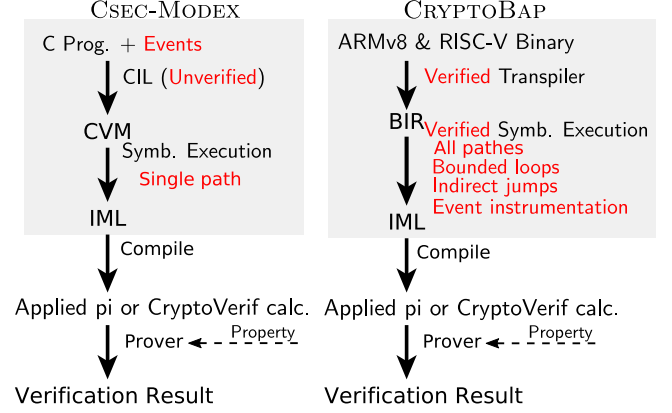


Figure 2: CRYPTO-BAP vs. CSEC-MODEX

framework in handling network messages as C structures), as well as an implementation of SSH called TinySSH [60]. Moreover, we have verified WireGuard [32], a modern VPN protocol integrated into Linux, by automatically extracting its ProVerif model.

Our threat model includes a set of functions whose input and output behavior is controlled by the attacker. This can represent a network attacker (when these functions are syscalls for network I/O) or a VM running in parallel (when these functions are hypercalls in a security hypervisor, e.g., [30, 43]). The execution platform is trusted to implement the machine-code semantics correctly and includes a set of trusted cryptographic functions. These are likewise assumed to be correct. Also, when outputting to ProVerif, the attacker is assumed to be Dolev-Yao, i.e., cryptography is perfect.

Outline of the CRYPTO-BAP’s approach. As in Fig. 2, CRYPTO-BAP (source code is available at <https://github.com/FaezehNasrabadi/CryptoBAP>) takes as input:

- the protocol participants’ binary;
- the symbolic model of cryptographic functions;
- and, the *security property* we verify for event traces of executions, formally defined in Sec. 8.

The properties we consider are *safety properties over event traces*; currently, CRYPTO-BAP supports verification of *authentication* and *weak secrecy* [23]. To verify these properties, we transpile the participants’ binary into the BIR representation, the internal language of the HolBA framework (see Sec. 3.1). Our transpiler is formally verified and guarantees to preserve the semantics of the machine code. However, BIR as defined in [53] is not suitable for reasoning about the security of cryptographic protocols. We address this by extending BIR in Sec. 4, e.g., to support network communication, random number generation, etc.

To export the property checking to off-the-shelf verifiers, we extract a model of protocol participants that these verifiers can process. To automate this, we symbolically execute the BIR translation to instrument on-the-fly the BIR code with events that flag completion of certain operations and occurrence of errors, and build its execution tree that contains all program execution paths. We translate the resulting execution tree to obtain equivalent programs in IML (see Sec. 3.2 and Sec. 6). The extracted IML model is then passed

¹We depict BIR in black, bold roman, SBIR in RedOrange, sans serif and IML in RoyalBlue, text italic. Elements common to all languages are typeset in black, italic.

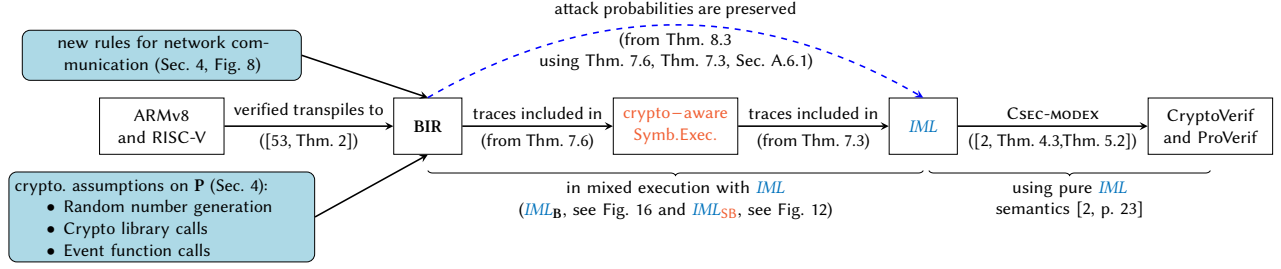
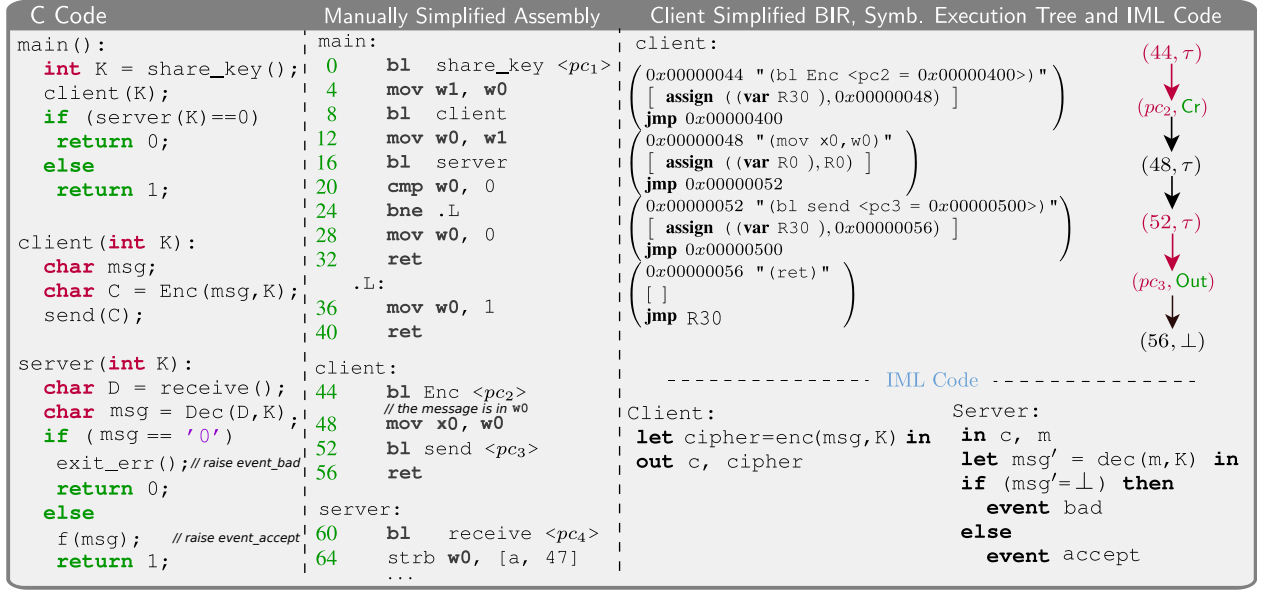


Figure 3: Organization of the CRYPTOBAP approach.

Figure 4: Running example. Function addresses are chosen randomly. Line numbers and pc_i are addresses of instructions and functions in the memory.

to CSEC-MODEX [1, 3], which applies algebraic rewriting to convert the model into the input language of ProVerif and CryptoVerif.

Akin to prior work [3, 4, 6, 16, 19, 27, 38, 40, 42], we trust the cryptographic primitives and abstract them with function symbols that are linked to a Dolev-Yao model when exporting to ProVerif, or to complexity-theoretic assumptions when exporting to CryptoVerif. Analyzing these primitives' correctness is important, but it requires a different methodology, e.g., weakest precondition propagation. Projects like MSR's Project Everest [14] and the resulting HACLS library [75], CompCert in conjunction with FCF [11, 74] or VALE [24] or synthesis approaches like Fiat-Crypto [35] address this equally challenging problem.

To prove that the extracted *IML* model preserves the behavior of the actual binary and to relate back the verified properties to the protocol binary, we define a mixed execution semantics. The mixed execution enables protocol participants from different abstraction layers to run in parallel and communicate (see Sec. 7.1 and Fig. 7.2). Such a mixed semantic also frees our symbolic execution from dealing with concurrency. Fig. 3 shows the interconnection between different layers in our approach. To summarize:

- We present CRYPTOBAP to automate the verification of cryptographic protocols' binary. Our framework explores all execution paths of protocols, resolves indirect jumps, and handles (compile-time) bounded loops automatically.
- We extend the vanilla symbolic execution engine in HolBA to automate the model extraction of cryptographic protocols. The way we extend the engine is significant. While HolBA's vanilla symbolic execution considers the program a holistic entity and thus basically encodes the semantics of the language (*BIR*), our semantics regards only a part of the whole program and abstracts cryptographic libraries, attacker calls, and random number generation.
- We formally verify the soundness of our approach and show that verified properties can be transferred back to the binary of analyzed protocols.
- To evaluate CRYPTOBAP, we have successfully verified multiple case studies, ranging from toy examples, e.g., CSur, to TinySSH and WireGuard.

Running example. Our running example, Fig. 4, consists of a client and a server that use a symmetric-key encryption scheme to communicate securely. This example shows a weak form of authentication, called *aliveness* [57]: the server will accept the connection to the (single) client only if it can successfully decrypt the received message using the pre-shared key. First, the client encrypts a message using the shared key, and sends it to the server. Second, the server receives the encrypted message at the other end and decrypts it using the same key. Depending on whether the decryption succeeds or fails, either `event_accept`, to show acceptance of the connection with the client, or `event_bad` will be released.

2 RELATED WORK

In the last decade, cryptographers started to employ and even develop theorem provers to develop verifiable proofs [39]. This started with the CertiCrypt framework for Coq [8], which subsequently developed into EasyCrypt [7]. These tools support reasoning about probabilistic programs and classes of (e.g., poly-time restricted) adversaries via probabilistic and probabilistic relational Hoare logic. There are also embeddings of probabilistic reasoning like FCF [64], Vrypto [5] and CryptHOL [55], which are easier to combine with other techniques (this was demonstrated [12, 64] for C code), but they require tedious manual analysis and a deep understanding of the underlying relational probabilistic logic.

These methods are typically used to verify cryptographic constructions. By contrast, complex protocols are analyzed using symbolic models, where cryptographic primitives like encryption, signatures, etc. are abstracted using a term algebra and a set of reduction rules. This makes the analysis of protocols that use such primitives amenable to automation, e.g., using first-order SAT solving [36], Horn clause resolution [20] or constraint solving [65]. To great success: within a decade, protocol verification tools went from analyzing small academic protocols [22, 46] to fully-fledged TLS models [13, 29].

This degree of automation comes at the cost of abstraction, both in terms of the computation environment and the cryptographic primitives. The former is owed to the focus on protocol *specifications* rather than implementations. This makes sense, because often, the task at hand is to evaluate designs or standards rather than specific implementations, which can be incomplete or nonexistent. There are efforts to translate implementations into the protocol specifications, but they are limited to high-level languages such as C [3]. The same holds for verification tools that operate at the source-code level [34, 48].

2.1 Verified crypto protocols' implementation

There have been efforts to verify (annotated) implementations of security protocols using: deductive verification [34], type checking [18], code generation [26], and model extraction. Existing work in model extraction mainly targets implementations in high-level languages like C [27, 38], F# [6, 13, 17] and Java [42, 62]. However, due to the complexity of such languages, the existing works have to limit their scope. For example, [27] does not model floating pointers and [38] omits explicit casts and negative array indexes. Table 1 compares selected works that verify the security of protocols via model extraction in some form.

There are also works that recover formats of protocol messages at the binary level [25, 51, 71, 72]. However, their intent is different from ours. Existing approaches are mostly applied to malware binaries and use heuristics to gain insights into their operation. Thus, they are not concerned with soundness, and the inferred message formats can be wrong.

Comparison to Aizatulin et al. approach. Closely related to CRYPTO-BAP is Aizatulin's work [3, 4]. Analogous to our work, they also proved the soundness of their approach, i.e., showed that all attacks present in the C code are preserved in the extracted *IML* models. The main difference between the two approaches is the fact that we target protocols' binary. Moreover, compared to their approach, which can handle only a single execution path, CRYPTO-BAP handles **all execution paths** of protocols, including those that contain **conditionals**, **bounded loops**, and **indirect jumps**. Also, their translation from C to C *virtual machine* (CVM)—the intermediate language used for extracting the *IML* model of protocols—is not verified. This renders relating the verified properties to the C implementation infeasible. Similarly, they abstracted cryptographic libraries, attacker calls, and random number generation in their verification. However, they do not formulate the requirements on the program under the analysis explicitly. CVM already assumed to include primitives for library calls, attacker calls, and random number generation—which are not primitives of the C language. In this regard, CVM is fairly close to *IML*. Moreover, our translation into *IML* is entirely different (a) because the input language has a more complex state and interaction with the other *IML* entities and (b) because in contrast to Aizatulin's approach, we handle conditionals, hence our symbolic execution is not only a sequence of actions. Finally, while they had to change the source code of protocols and add dummy functions to flag the occurrence of events, CRYPTO-BAP automates releasing events during symbolic execution and minimizes user involvement.

3 BACKGROUND

We next explain the preliminaries before presenting the details of our approach.

3.1 HolBA framework & Vanilla symbolic exec.

CRYPTO-BAP relies on HolBA [53] to transpile the binary of protocols to the BIR representation. BIR is a simple and architecture-agnostic language used as the internal language of HolBA and is designed to simplify the binary analysis of programs and facilitate building analysis tools. HolBA is proof-producing and ensures that the transpilation preserves the semantics of the binary. A BIR program P includes a number of blocks, see Fig. 5, each consisting of a tuple of a unique label—a string or an integer—and a few statements. Each label refers to a particular location in the program and is often used as the target of jump instructions (i.e., `jmp` or `cjmp`). BIR expressions include constants, standard binary and unary operators (ranged over by \diamond_b and \diamond_u) for finite integer arithmetic, memory operations, and conditionals. Fig. 4 presents a BIR snippet for the running example.

A BIR state $(\eta, pc) \in S$ consists of an environment $\eta : Bvar \mapsto Bval$ which maps variables, i.e., registers r_i and memory locations Mem , to values and a program counter pc that holds the label of the

Papers	Language	Abstract Model	Model Type	Property	Soundness
Aiazatulin et al.[3, 4]	C	Applied-pi	Symb. + Comp.	Secrecy, Auth.	✓
Chaki et al.[27]	C	ASPIER	Symb.	Secrecy, Auth.	✓
Goubault-Larrecq et al.[38]	C	Horn clauses	Symb.	Secrecy, Info. flow	✓
Backes et al.[6]	F#	RFC	Symb. + Comp.	Safety properties	✓
Bhargavan et al.[16, 19]	F#	Pi/CryptoVerif	Symb. + Comp.	Secrecy, Auth.	✗
Jürjens[42]	Java	First-order Logic	Symb.	Secrecy, Auth.	✗
HE et al.[40]	Python-JS	Applied-pi	Symb.	Secrecy, Auth.	✗
This work	Binary	Applied-pi	Symb. + Comp.	Secrecy, Auth.	✓

Table 1: Selected model extraction approaches; Symb = Symbolic, Comp = Computational, Auth = Authentication, JS = JavaScript.

$P \in \text{prog} := \text{block}^*$
 $\text{block} := (v, \text{stmt}^*)$
 $v \in \text{Bval} := \text{string} \mid \text{int}$
 $\text{stmt} := \text{assign}(\text{string}, e) \mid \text{assert}(e) \mid \text{halt} \mid \text{jmp}(e)$
 $\quad \mid \text{cjmp}(e, e, e)$
 $e \in \text{Bexp} := v \mid \text{var string} \mid \Diamond_u e \mid e \Diamond_b e \mid \text{ifthenelse}(e, e, e)$
 $\quad \mid \text{load}(e, e, \text{int}) \mid \text{store}(e, e, \text{int})$

Figure 5: The BIR syntax

executing BIR block. The relation $\rightarrow \subseteq S \times S$ models the execution of a BIR block. The execution of n steps is denoted by \rightarrow^* if $n \geq 0$ and \rightarrow^+ or \rightarrow^n , if $n > 0$.

We build CRYPTOBAP on a proof-producing symbolic execution for BIR [52] that formalizes the symbolic generalization of BIR (hereafter SBIR). The symbolic semantics is bisimilar to the concrete one and allows guiding the execution while maintaining a sound set of reachable states from an initial symbolic state (we call this the *symbolic execution structure*). To generalize from BIR to SBIR, symbolic expressions SE are defined that can be interpreted to BIR values Bval via an interpretation $H : \text{SE} \rightarrow \text{Bval}$. In addition to the symbolic environment $\eta : \text{Bvar} \mapsto \text{SE}$, the SBIR state $(\phi, \eta, \text{pc}) \in S$ also contains a path condition $\phi \in \text{SE}$ and a pc that is kept concrete to obtain a concrete control flow.

Let $\rightarrow : S \times S$ be the single-step transition relation of SBIR and \rightarrow^n (or \rightarrow^+) denote a multi-step symbolic transition. We write $s_i \rightarrow_L^n s_j$ to restrict the transition from s_i to s_j to the label set L. For the HolBA's vanilla symbolic execution, Lindner et al. [52] proved that a single SBIR execution step soundly matches a single BIR execution step, characterized by the following simulation theorem:

PROPERTY 1. For all s_i, H, s_j, s_i s.t. $s_i \sim_H s_i$, if $s_i \rightarrow s_j$ then there exist an H' and s_j s.t. $H \subseteq H'$ and $s_i \rightarrow s_j$ and $s_j \sim_{H'} s_j$.

The simulation relation \sim_H asserts the consistency of corresponding BIR and SBIR states, i.e., their program counters are equal, their environments are equal through the interpretation H , and the evaluation of ϕ under H results in *true*. Then the soundness of the symbolic execution structure for multiple steps corresponds to the extension of Property 1 to a multi-step simulation theorem.

$d, e \in \text{Iexp} := \text{expression}$
 $b \in BS, x \in \text{Ivar}$ bitstrings, variables
 $op(e_1, \dots, e_m)$ computation, $op \in \text{Ops}$
 $P, Q \in \text{IML} := \text{process}$
 $0, P \mid Q$ nil, parallel composition
 $!^{i \leq m} P$ replicating P , m times
 $\text{new } x : t; P$ randomness
 $\text{in}(c[e_1, \dots, e_m], x); P$ input
 $\text{out}(c[e_1, \dots, e_m], e); P$ output
 $\text{event}(d_1, \dots, d_m); P$ event
 $\text{if } e \text{ then } P \text{ [else } Q]$ conditional
 $\text{let } x = e \text{ in } P$ assignment
 $\text{assume } e; P$ assumption

Figure 6: A fragment of IML syntax

3.2 CSEC-MODEX toolchain & IML

Aiazatulin et al. [3] proposed an automated technique to verify the security of cryptographic protocols' C implementation. At a high level, CSEC-MODEX takes as input the C code of protocol participants together with a template file for the verifier (ProVerif or CryptoVerif). The toolchain extracts the IML model of the protocol, which is then converted into the verifier's input language. The template encodes assumptions about cryptographic primitives in the implementation, the environment process which spawns the participants and generates shared cryptographic material, and a query for the security property that is checked for the implementation.

The intermediate model language, IML, is a version of the applied-pi calculus extended with bitstring manipulation primitives. In Fig. 6, $BS = \{0, 1\}^*$ is the set of finite bitstrings, Ops is the set of operations, including cryptographic primitives, and $op(e_1, \dots, e_m)$ denotes function application. IML expressions are evaluated with respect to an environment $\eta : \text{Ivar} \mapsto BS \cup \{\perp\}$ which maps variables to bitstrings or \perp .

P and Q represent input/output processes. An executing process is the basic unit of execution in IML and has the form (η, P) , where P is either an input or output process. The input process 0 does nothing. In IML, inputs and outputs are performed using *in* and *out* in which c denotes the channel name and e_1, \dots, e_m indicate the protocol participants' identifier. The construct $\text{new } x : t$ generates

$$\begin{array}{c}
t = \text{fixed}_n \text{ for some } n \in \mathcal{N} \quad |b| = n \\
\hline
(\eta, \text{new } x : t; P), Q \xrightarrow{\text{fr}(b)}_I (\eta[x \mapsto b], P), Q \\
\\
\llbracket e \rrbracket_\eta = b \neq \perp \quad b' = \text{truncate}(b, \text{maxlen}(c)) \\
\forall j \leq m : \llbracket e_j \rrbracket_\eta = b_j \neq \perp \quad Q' = \text{reduce}(\{(\eta, Q)\}) \\
\hline
\exists! (\eta', Q') \in Q : Q' = \text{in}(c[e'_1, \dots, e'_m, x']; P') \wedge \forall j \leq m : \llbracket e'_j \rrbracket_{\eta'} = b_j \neq \perp \\
(\eta, \text{out}(c[e_1, \dots, e_m], e); Q), Q \xrightarrow{I} (\eta'[x' \mapsto b'], P'), Q \sqcup Q' \setminus \{(\eta', Q')\}
\end{array}$$

Figure 7: The semantics of *IML* [2, p. 23] transition relation.

a uniform random number of type t and $\text{event}(d_1, \dots, d_m)$ is used to raise an event during the execution.

An *IML* state $(\eta, P), Q \in S$ includes an output process P and a multiset of executing input processes Q . The initial configuration of an input process Q is defined as $(\emptyset, \text{out}(c, \varepsilon); \emptyset)$, $\text{reduce}(\emptyset, Q)$ where reduce represents a function that executes a sequence of processes inside Q (e.g., $Q = P_1; P_2; \dots$) until an input process waiting for a message from channel c is reached (see 2nd rule in Fig. 7).

We use $\xrightarrow{a}_p \subseteq S \times S$ to denote the *IML* transition relation with the probability p and the event a . The event a may be empty or include a single event of the form $\text{ev}(b_1, \dots, b_m)$, where ev is an event symbol and b_1, \dots, b_m are bitstrings. Moreover, an *IML* trace is defined as $R = s_1 \xrightarrow{a_1}_{p_1} \dots \xrightarrow{a_{n-1}}_{p_{n-1}} s_n \subseteq \mathcal{R}^l(Q)$.

We have borrowed the semantics of the *IML* transition relations from [2, p. 23]. A few representative transition rules of *IML* for random number generation and sending a message on the channel c are presented in Fig. 7 and the rest explained in Sec. A.1. In this figure truncate cuts messages according to the provided length and maxlen is the maximum size of the channel. We extend the random number generation rule with an event fr which represents the creation of a fresh bitstring b . This simplifies stating our invariants but is operationally the same.

4 BIR WITH CRYPTOGRAPHY

BIR, as described in [53], does not support ingredients required to reason about the security of cryptographic protocols. To resolve these issues, we model random number generation and abstract network communications and formulate assumptions on state transformation in certain function calls on top of the existing BIR semantics. Such an extension preserves the verified properties of BIR and, thus, the soundness of binary transpilation.

Using the information in the (unstripped) binaries' header and preprocessing of lifted programs, we split the address space of BIR into five label sets: $L = L_{\mathcal{N}} \sqcup L_{\text{Op}} \sqcup L_{\mathcal{A}} \sqcup L_{\mathcal{R}} \sqcup L_{\mathcal{E}}$. The sets $L_{\text{Op}} = \bigcup_{\text{op} \in \text{Op}}$, $L_{\mathcal{A}}$, $L_{\mathcal{R}}$, $L_{\mathcal{E}}$ correspond to the *cryptographic libraries*, *attacker calls*, *random number generation*, and *event* functions. Addresses outside these label sets are classified as normal execution points in $L_{\mathcal{N}}$. Moreover, a specific label set $L_{\mathcal{L}} \subset L_{\mathcal{N}}$ defines loop entry points. For each label set, we axiomatize the expected behavior of the BIR program by defining a number of assumptions. We also define a specific entry point for each function—denoted by ξ_ℓ for $\ell \in \{L_{\mathcal{N}}, L_{\text{Op}}, L_{\mathcal{A}}, L_{\mathcal{R}}, L_{\mathcal{E}}\}$ —and ensure that function calls are done only through the specified entry points.

In the crypto-aware symbolic execution, these function calls will be treated as atomic operations. We thus introduce some notation to indicate with an event whenever a *sequence* of steps passes via these special functions.

We extend the BIR transition relation with events, $\xrightarrow{a} \subseteq S \times E \times S$, where E is the set of observable events plus the silent transition τ . Then, a multi-step BIR transition $s_0 \xrightarrow{(a_1, \dots, a_m)}^+ s_n$ exists if $s_0 \xrightarrow{a_1}^* s_1 \dots s_{n-1} \xrightarrow{a_m}^* s_n$ where $s_i \xrightarrow{a}^* s_j$ if $s_i \xrightarrow{\tau}^* s' \xrightarrow{a} \tau^* s_j$ is reminiscent to the big-step semantics. Also, we use $\mathcal{R}^b(P, s_0)$ to denote the set of execution traces of P starting from the initial state s_0 .

We define $\text{ret} : S \rightarrow L$ to obtain the next execution point immediately reachable after returning from a call. $\text{mstore} : (\text{Bvar} \mapsto \text{Bval}) \times \text{Bvar} \times 2^{\text{Bvar}} \times \text{BS} \times \mathcal{N} \rightarrow (\text{Bvar} \mapsto \text{Bval}) \times \text{Bval}$ stores bitstrings into the memory in the BIR environment. Given $\text{heap} \in \text{Mem}$ and $l \in \{1, 8, 16, 32, 64, 128\}$ and $|b|$ a multiple of l that can be encoded in 128 bits, $\text{mstore}(\eta, \text{heap}, \text{Mem}, b, l)$ stores b in l -bit chunks, preceded by l (encoded as a word) in the memory $\text{Mem} \subseteq \text{Bvar}$, starting from the pointer stored in heap . mstore returns a new environment and the address of the data within it, as indicated by heap in the *previous* environment η . We also introduce notation for reading this bitstring. Let \parallel be the byte concatenation operator. Then, $\text{mload}(\eta, a) \stackrel{\text{def}}{=} \parallel_{i=1 \dots \eta(a)} \eta(a + i)$ for the given η and the address ' a '.

CRYPTOBAP supports *call-by-value*, *call-by-reference*, and data passing via global variables (which TinySSH uses) call conventions. For brevity, we focus on the call-by-value convention; the others follow a similar pattern.

Random number generation (RNG). BIR is a deterministic language; as a result, we are unable to draw cryptographic keys without an external source of randomness that the attacker cannot predict. Thus, we allocate a memory region RM in the initial state for storing $k \in \mathcal{N}$ random values of size $l \in \mathcal{N}$, for the security parameter $n = lw$ that is a multiple of some supported word length w . We assume RM is an ordered list of kl consecutive addresses. To track the number of words read from RM , we define a counter \mathbf{r}_k and store it in the environment. Given an initial state s_0 and a random tape $\text{rm}_k \in \{0, 1\}^{kl \times w}$ the state $s_0. \eta[\text{RM} \mapsto \text{rm}_k, \mathbf{r}_k \mapsto 0]$ is an instance of this initial state. To extract a random number of size l from RM , we define $\mathfrak{R} : (\text{Bvar} \rightarrow \text{Bval}) \times \mathcal{N} \rightarrow \text{BS}$ which returns a value from RM yet unread: $\mathfrak{R}(\eta, n) \stackrel{\text{def}}{=} \text{let } \mathbf{x} = \eta[\mathbf{r}_k] \text{ in } \parallel_{i=0 \dots l-1} \eta[\text{RM} + \mathbf{x} + i]$. This construction is reminiscent of probabilistic Turing machines, only that the random number generator is finite due to the finite-memory restriction of BIR's memory.

We call a function from $L_{\mathcal{R}}$ with $\mathbf{l}_R \in L_{\mathcal{R}}$ one of its entry points an RNG function, if for any entering state (η_0, \mathbf{l}_R) for which $\mathfrak{R}(\eta_0, n)$ is defined, and execution point (η'', \mathbf{l}) after returning from RNG, i.e., $\mathbf{l} = \text{ret}(\eta_0, \mathbf{l}_R)$, the output register holds the address of a copy of the random value and \mathbf{r}_k is updated, i.e., $\eta'' = \eta'[\mathbf{r}_0 \mapsto a; \mathbf{r}_k \mapsto \eta_0[\mathbf{r}_k] + l]$ with $(\eta', a) = \text{mstore}(\eta_0, \text{heap}, \text{Mem}, x_i, 128)$ s.t. $x_i = \mathfrak{R}(\eta_0, n)$. We denote RNG steps as $(\eta_0, \mathbf{l}_R) \xrightarrow{\text{Fr}(x_i)}^*_{L_{\mathcal{R}}} (\eta'', \mathbf{l})$, with $i = \left\lfloor \frac{\eta_0[\mathbf{r}_k]}{l} \right\rfloor + 1$ being a counter for the number of times the RNG function was called.

$$\begin{array}{c}
\text{pc} \in \xi_{\mathcal{A}_s} \quad \text{pc}' = \text{ret}(\eta, \text{pc}) \quad \mathbf{a} = \eta[\mathbf{r}_0] \quad \mathbf{e} = \text{mload}(\eta, \mathbf{a}) \\
\hline
\mathcal{P} \vdash (\eta, \text{pc}), \mathbf{c}[\mathbf{e}_1, \dots, \mathbf{e}_m] \xrightarrow{\text{Out}(\mathbf{e}, (\mathbf{e}_1, \dots, \mathbf{e}_m))} (\eta, \text{pc}'), \mathbf{e} :: \mathbf{c}[\mathbf{e}_1, \dots, \mathbf{e}_m] \quad \mathcal{A}_s
\end{array}
\quad
\begin{array}{c}
\text{pc} \in \xi_{\mathcal{A}_r} \quad \text{pc}' = \text{ret}(\eta, \text{pc}) \quad (\eta', \mathbf{a}) = \text{mstore}(\eta, \text{heap}_{\mathcal{A}}, \text{Mem}_{\mathcal{A}}, \mathbf{e}', 128) \\
\hline
\mathcal{P} \vdash (\eta, \text{pc}), \mathbf{e}' :: \mathbf{c}[\mathbf{e}_1, \dots, \mathbf{e}_m] \xrightarrow{\text{In}(\mathbf{e}', (\mathbf{e}_1, \dots, \mathbf{e}_m))} (\eta'[\mathbf{r}_0 \mapsto \mathbf{a}], \text{pc}'), \mathbf{c}[\mathbf{e}_1, \dots, \mathbf{e}_m] \quad \mathcal{A}_r
\end{array}$$

Figure 8: The semantics of BIR network communication. BIR uses *IML* channels \mathbf{c} for communication.

Network communication. Protocols relate events on different participants. Therefore, a setting where multiple parties run in parallel is essential to analyze protocols' correctness. Sec. 7 introduces a mixed execution, in which BIR programs run in parallel with *IML* processes. The latter model protocol participants for which we do not have a BIR implementation, but also the adversary.

Our BIR programs rely on an *IML* channel for communication that has the form $\mathbf{c}[\mathbf{e}_1, \dots, \mathbf{e}_m]$, where $\mathbf{e}_1, \dots, \mathbf{e}_m$ are expressions which identify communicating parties and their channel \mathbf{c} . To send a message \mathbf{e} ($\xi_{\mathcal{A}_s}$ in Fig. 8), we fetch the value of \mathbf{e} from the memory address $\eta[\mathbf{r}_0]$, put it on the channel $\mathbf{e} :: \mathbf{c}[\mathbf{e}_1, \dots, \mathbf{e}_m]$, and release the $\text{Out}(\mathbf{e}, (\mathbf{e}_1, \dots, \mathbf{e}_m))$ event.

To receive a message $\mathbf{e}' :: \mathbf{c}[\mathbf{e}_1, \dots, \mathbf{e}_m]$, represented by $\xi_{\mathcal{A}_r}$ in Fig. 8, we store it in a buffer that is only accessible to libraries and return (via register \mathbf{r}_0) the address, i.e., ' \mathbf{a} ', of the memory region where the message \mathbf{e}' is stored. Passing the address via \mathbf{r}_0 is just one way to model the send and receive functions that also accommodates passing the buffer address by reference. CRYPTOBAP models these functions according to the implementation.

Crypto library. We establish a set of concrete assumptions on the way crypto libraries operate. That is, a crypto-library call, like **op**, computes the correct result, never invokes another function, and only changes its own memory, i.e., Mem_{Op} . We denote library

steps with $(\eta_0, \text{l}_{\text{Op}}) \xrightarrow{\text{Cr}(\mathbf{v})}^*_{\text{L}_{\text{Op}}} (\eta'', \mathbf{l})$, and expect transitions using labels outside L_{Op} do not change the memory of library calls.

We call $\text{L}_{\text{Op}} \subseteq \bigcup_{\text{op} \in \text{Op}} \text{L}_{\text{Op}}$ the library implementation of **op** (with arity m) and $\text{l}_{\text{Op}} \in \xi_{\text{Op}}$ one of its entry points, if for any entering state $(\eta_0, \text{l}_{\text{Op}})$ and the return state (η'', \mathbf{l}) , the function result $\mathbf{v} = \text{op}(\mathbf{b}_1, \dots, \mathbf{b}_m)$ for $\mathbf{b}_i = \text{mload}(\eta, \mathbf{r}_i)$, $i \in \{1, \dots, m\}$, is stored in a heap and its address is put into \mathbf{r}_0 : $\eta''[\mathbf{r}_0 \mapsto \mathbf{a}]$ where $(\eta', \mathbf{a}) = \text{mstore}(\eta_0, \text{heap}_{\text{Op}}, \text{Mem}_{\text{Op}}, \mathbf{v}, 128)$.

Event functions. Event functions identify specific steps in our program that we want to argue about. For example, when a protocol ends with the establishment of a key, that key is used to transmit some data. We want to show that, whenever this step is reached, it is authenticated, i.e., the purported communication partner has requested the execution of this step (e.g., $\mathbf{f}(\text{msg})$ in Fig. 4). What happens in this step is not important for us, only that it is reached. We hence assume, for simplicity, that such functionality is replaced by stand-ins we call event functions. These only raise a visible event, but do not alter the memory. We denote the transition corresponding to an event function call with $(\eta_0, \text{l}_{\mathcal{E}}) \xrightarrow{\text{Ev}(\mathbf{b}_1, \dots, \mathbf{b}_m)}^*_{\text{L}_{\mathcal{E}}} (\eta_0, \mathbf{l})$ where $\text{l}_{\mathcal{E}} \in \text{L}_{\mathcal{E}}$ is the entry point, $\mathbf{b}_i = \text{mload}(\eta, \mathbf{r}_i)$ for $i \in \{1, \dots, m\}$ are event parameters, and $\mathbf{l} \in \text{L}_{\mathcal{N}}$.

5 CRYPTO-AWARE SYMBOLIC EXECUTION

We have *significantly* extended HolBA's vanilla symbolic execution [52] to handle network communication, calls to crypto primitives and event functions, and random number generation, which are essential to reason about protocols' security. We defined the rules for our symbolic execution in Fig. 9. In this figure, $\text{range}(f)$ returns the image of f . For library calls, we define an *oracle* $L : \xi_{\text{Op}} \times (\text{Bvar} \mapsto \text{SE}) \rightarrow \text{SE}$ to compute the result of the invoked function w.r.t. the current **pc** and symbolic environment. For the symbolic execution, we initialize the memory region to store random numbers **RM** with symbolic values. Thus, \mathfrak{R} signifies the symbolic lifting of \mathfrak{R} , and RNG generates a fresh symbolic expression to represent the extracted value.

Similar to BIR transitions, we extend the symbolic transition relation of **SBIR** with events, i.e., $\xrightarrow{\mathbf{a}} \subseteq \mathcal{S} \times \mathcal{E} \times \mathcal{S}$, and use $\mathcal{R}^{\mathfrak{S}}(\mathbf{P}, \mathbf{s}_0)$ to denote the set of symbolic traces of **P** starting at \mathbf{s}_0 .

Bounded Loops. Loops can naively be handled by *unrolling*. This, however, is inefficient in most cases and can quickly result in a path explosion. To avoid this, we summarize loops following Strejček [68]. The algorithm summarizes the loops' effect on program variables and path conditions to compute a necessary condition on the loop's inputs to reach a specific execution point in the program. The summary is computed in terms of a tuple of *iterated symbolic state* and *looping condition*. The iterated symbolic state computes for each variable modified within the loop its symbolic value based on the initial value of the program's variables and *path counters*. Each path counter indicates the number of iterations of a specific path within the loop leading from the loop entry point to itself. For each path in the loop, a path condition is computed, and the conjunction of all such conditions is the looping condition.

We have *automated* the loop summarization process in our symbolic execution. In Fig. 9, the function $\text{processLoop} : \mathcal{S} \rightarrow \text{SE} \times (\text{Bvar} \mapsto \text{SE})$ represents our implementation to summarize loops' effect. It takes as input the symbolic state of the loop entry point and reflects the effect of the loop body in its exit state (computed by $\text{exit} : \xi_{\mathcal{L}} \rightarrow \text{L}$). The rule also raises the event **loop**(\mathbf{t}) with \mathbf{t} being the number of loop iterations.

Loops in protocol implementations are often not bounded; typically, each session runs in a *while(true){..}* loop until the server is externally terminated. However, the semantics of *IML*, like most cryptographic standard models, assumes a bound on the protocol. Thus, we need to assume that such loops are externally terminated after some polynomial time in the security parameter. This is captured by our automated loop summarization and by translation to the replication operator.

Indirect Jumps. If during symbolic execution of the code, we encounter an indirect jump, e.g., **jmp** \mathbf{e} , we evaluate \mathbf{e} w.r.t. the current state to get an expression \mathbf{e}' ; we then query the SMT solver for a satisfiable assignment to $\mathbf{tgt} = \mathbf{e}' \wedge \phi$, assuming that \mathbf{tgt} does

$$\begin{array}{c}
\frac{\text{pc} \in \xi_{\mathcal{A}_s} \quad \text{pc}' = \text{ret}(\phi, \eta, \text{pc}) \quad \mathbf{a} = \eta[r_0] \quad \mathbf{e} = \text{mload}(\eta, \mathbf{a})}{\text{P} \vdash (\phi, \eta, \text{pc}), \mathbf{c}[e_1, \dots, e_m] \xrightarrow{\text{Out}(\mathbf{e}, (e_1, \dots, e_m))} (\phi, \eta, \text{pc}'), \mathbf{e} :: \mathbf{c}[e_1, \dots, e_m]} \mathcal{A}_s \quad \frac{\text{pc} \in \xi_{\mathcal{E}} \quad \text{pc}' = \text{ret}(\phi, \eta, \text{pc}) \quad (d_1, \dots, d_m) \notin \text{range}(\eta)}{\text{P} \vdash (\phi, \eta, \text{pc}) \xrightarrow{\text{Ev}(d_1, \dots, d_m)} (\phi, \eta, \text{pc}')} \text{event} \\
\\
\frac{\text{pc} \in \xi_{\mathcal{L}} \quad \text{pc}' = \text{exit}(\text{pc}) \quad \mathbf{t} \notin \text{range}(\eta) \quad (\phi', \eta') = \text{processLoop}(\phi, \eta, \text{pc})}{\text{P} \vdash (\phi, \eta, \text{pc}) \xrightarrow{\text{loop}(\mathbf{t})}^+ (\phi', \eta', \text{pc}')} \text{loop} \\
\\
\frac{\text{pc} \in \xi_{\mathcal{R}} \quad i = \left\lfloor \frac{\eta[r_k]}{l} \right\rfloor + 1 \quad \mathbf{x}_i = \mathcal{R}(\eta, n) \quad (\eta', \mathbf{a}) = \text{mstore}(\eta, \text{heap}, \text{Mem}, \mathbf{x}_i, 128) \quad \eta'' = \eta'[r_0 \mapsto \mathbf{a}; r_k \mapsto \eta[r_k] + l]}{\text{P} \vdash (\phi, \eta, \text{pc}) \xrightarrow{\text{Fr}(\mathbf{x}_i)} (\phi, \eta'', \text{pc}')} \text{RNG}(n) \\
\\
\frac{\text{pc} \in \xi_{\mathcal{A}_r} \quad \text{pc}' = \text{ret}(\phi, \eta, \text{pc}) \quad \mathbf{e} \notin \text{range}(\eta) \quad (\eta', \mathbf{a}) = \text{mstore}(\eta, \text{heap}_{\mathcal{A}}, \text{Mem}_{\mathcal{A}}, \mathbf{e}, 128)}{\text{P} \vdash (\phi, \eta, \text{pc}), \mathbf{e} :: \mathbf{c}[e_1, \dots, e_m] \xrightarrow{\text{In}(\mathbf{e}, (e_1, \dots, e_m))} (\phi, \eta'[r_0 \mapsto \mathbf{a}], \text{pc}'), \mathbf{c}[e_1, \dots, e_m]} \mathcal{A}_r \\
\\
\frac{\text{pc} \in \xi_{\text{Op}} \quad \text{pc}' = \text{ret}(\phi, \eta, \text{pc}) \quad \mathbf{v} \notin \text{range}(\eta) \quad \mathbf{v} = L(\text{pc}, \eta) \quad (\eta', \mathbf{a}) = \text{mstore}(\eta, \text{heap}_{\text{Op}}, \text{Mem}_{\text{Op}}, \mathbf{v}, 128)}{\text{P} \vdash (\phi, \eta, \text{pc}) \xrightarrow{\text{Cr}(\mathbf{v})} (\phi, \eta'[r_0 \mapsto \mathbf{a}], \text{pc}')} \text{library}
\end{array}$$

Figure 9: Crypto-aware symbolic execution semantics. Here, $\mathbf{e}, \mathbf{d}, \mathbf{v}, \mathbf{x}_i \in \text{SE}$.

not occur in \mathbf{e}' and ϕ . The solver returns one possible target, say \mathbf{t} . We repeat this procedure, each time asking the solver to exclude found targets, until the query becomes unsatisfiable. This technique was sufficient for our experiments; however, for more complex cases, some optimizations would be required, e.g., considering only a subset of possible targets instead of enumerating all.

We symbolically execute BIR programs to instrument them with events that facilitate clear observation of implementation behavior and to obtain their execution tree, which is later used to obtain the corresponding IML model. A node in this tree is either a branching node $\text{Branch}(\gamma, T_1, T_2)$ with the condition γ and sub-trees T_1 , or an event node $\text{node}(\text{pc}, \text{ev})$ with pc specifying where the event occurred. We add a **halt** statement at the end of each complete path, i.e., leaves are due to **halt** statements with \perp as the event. An edge connects two nodes iff they are in the transition relation.

The tree is constructed from a BIR program and an initial symbolic state as follows: the root is the initial state. For any node, including the root, the crypto-aware symbolic execution gives us up to two successors states. If the node represents a branching statement, we obtain two successor states. We store the statement's condition in a branching node and proceed to translate the two successor states into subtrees. If the node represented any other statement, there can only be one or no successor state, and we store an event node with, or respectively without, a successor tree. Since we abstract function calls and loops, we safely assume that each node in the tree can be uniquely identified by the pc of its statement. We define the selection operator $[\]$ to extract the node for a given program counter, e.g., $T[\text{pc}]$ will return a node indexed by pc .

Fig. 4 shows a fragment of the symbolic execution tree for the Client of our running example. Note that, each function call is depicted with two nodes: the first node loads the address of the callee into pc , and the second node is the actual call, represented as an atomic transition.

$T = \text{node} :: T'$	Event tree
$\langle \text{Leaf} :: T' \rangle \mapsto 0; \langle T' \rangle$	
$\langle \text{node} :: T' \rangle :=$	Events nodes
$\langle (\text{pc}, \tau) :: T' \rangle$	$\mapsto \langle T' \rangle$
$\langle (\text{pc}, \text{Ev}(d_1, \dots, d_m)) :: T' \rangle$	$\mapsto \text{event}(d_1, \dots, d_m); \langle T' \rangle$
$\langle (\text{pc}, \text{In}(\mathbf{v}, (e_1, \dots, e_m))) :: T' \rangle$	$\mapsto \text{in}(\mathbf{c}[e_1, \dots, e_m], \mathbf{v}); \langle T' \rangle$
$\langle (\text{pc}, \text{Out}(\mathbf{e}, (e_1, \dots, e_m))) :: T' \rangle$	$\mapsto \text{out}(\mathbf{c}[e_1, \dots, e_m], \mathbf{e}); \langle T' \rangle$
$\langle (\text{pc}, \text{Cr}(\mathbf{v})) :: T' \rangle$	$\mapsto \text{let } x = \mathbf{v} \text{ in } \langle T' \rangle \text{ where } x \text{ is fresh}$
$\langle (\text{pc}, \text{Fr}(\mathbf{x}_i)) :: T' \rangle$	$\mapsto \text{new } \mathbf{x}_i; \langle T' \rangle$
$\langle (\text{pc}, \text{loop}(\mathbf{t})) :: T' \rangle$	$\mapsto \text{!}^{\mathbf{t} \leq m} \text{LoopProc}(\text{pc}); \langle T' \rangle$
$\langle (\text{pc}, \text{Branch}(\gamma, T_1, T_2)) \rangle \mapsto \text{if } \langle \gamma \rangle \text{ then } \langle T_1 \rangle \text{ else } \langle T_2 \rangle$	
$\langle \gamma \rangle :=$	Expressions
$\langle \mathbf{b} \in \text{Bval} \rangle \mapsto \langle \mathbf{b} \rangle \in \text{BS}$	
$\langle \mathbf{x} \in \text{Bvar} \rangle \mapsto \langle \mathbf{x} \rangle \in \text{Ivar}$	
$\langle \gamma_1 \diamond_b \gamma_2 \rangle \mapsto \langle \gamma_1 \rangle \langle \diamond_b \rangle \langle \gamma_2 \rangle$	Binary Ops.
$\langle \diamond_b \rangle \mapsto \begin{cases} \wedge & \text{AND} \\ \vee & \text{OR} \\ = & \text{Equal} \\ + & \text{Plus} \\ \dots & \end{cases}$	
$\langle \diamond_u \gamma' \rangle \mapsto \langle \diamond_u \rangle \langle \gamma' \rangle$	Unary Ops.
$\langle \diamond_u \rangle \mapsto \begin{cases} \neg & \text{Not} \\ \perp & \text{otherwise} \end{cases}$	
$\langle f(e_1, \dots, e_m) \rangle \mapsto \langle f \rangle(\langle e_1 \rangle, \dots, \langle e_m \rangle)$	

Figure 10: Rules for the translation of the symbolic execution tree T to IML model.

6 MODEL EXTRACTION

We now proceed to explain how to automatically extract the IML model from protocols' BIR representation. Our model extraction approach relies on translating the symbolic execution tree T of the protocol under adversarial semantics into its corresponding IML

model. We translate T into an executing process Q^{full} according to the rules depicted in Fig. 10, where (T) represents the compiled process, i.e. $Q^{full} = (T)$. Since T contains all possible execution of protocols and their interactions with the crypto primitives and the attacker, the extracted model includes all behaviors of the protocol at its binary representation (i.e., all attacks present at the binary level are preserved in the extracted model).

Our translation converts leaf nodes into a *nil* process 0 . For internal nodes, we translate the event stored in each node into its *IML* counterpart. Loops are modeled using the replication operator of *IML*; $LoopProc : \xi_L \rightarrow P$ converts the loop body into its corresponding *IML* process using the defined rules. Notice that we do not translate τ events. Fig. 10 also presents our rules to translate symbolic *BIR* expressions. Intuitively, the symbolic execution is used to symbolically compute the effects of such transitions, while the protocol model only contains the interactions with the network. Our rules to translate expressions are standard. The only interesting one is the translation of the function application, which is used, e.g., to translate memory *load/store* and bitwise operations. For example, this rule translates a memory load operation $load(mem, pa, l)$, for $l \in \{1, 8, 16, 32, 64, 128\}$, into $read(x_1, l)$, where x_1 is the fresh name chosen for the symbolic value in mem at the address pa .

Fig. 4 presents the *IML* model of the running example. In this model, c is the input and output channel, bad is the event that we release if the decryption is not successful, and enc (dec) is the encryption (resp., the decryption).

6.1 CRYPTOBAP vs. CSEC-MODEX *IML* models

Our derived *IML* models are simpler than CSEC-MODEX without losing accuracy. To demonstrate this, we use the simple XOR case study from CSEC-MODEX set of case studies. Simple XOR implements a protocol in which the one-time pad includes both protocol parties. The methodology we employ to derive the *IML* model from binary code differs significantly from that used in CSEC-MODEX which leads to much simpler models. The most glaring difference between CRYPTOBAP and CSEC-MODEX is that our analysis produces a symbolic tree that is translated into an *IML* process with conditionals and replication (see Fig. 10) instead of a single path that is translated into a linear *IML* process. Even for the linear subprocesses, our toolchain produces processes that are shorter and often human-readable (at least for small case studies, e.g., XOR). This is because the CSEC-MODEX translates each symbolic CVM process (their intermediate representation of C) into *IML* and performs the most simplification steps concerning the bitwise operations (concatenation, extraction, etc.) later at the translation step into CryptoVerif and ProVerif. For instance, in CSEC-MODEX's *IML* model for the client side of simple XOR case study, shown in Fig. 11, $(1)^{[u, 1]} \mid \text{nonce1}$ is simplified to $\text{conc1}(\text{nonce1})$ in its CryptoVerif model. Instead, we leverage the support for simplification of these operations at symbolic execution time, because (i) support there is much more mature (ii) will benefit from future development (iii) simplified constraints can also be used for path elimination and simplification in follow-up states. In addition to this, our *IML* model for the client side of the simple XOR case study is more concise since several assumptions made in CSEC-MODEX's *IML* model were unnecessary for verifying this particular case study. Fig. 11 presents the different *IML* models

produced by CRYPTOBAP and CSEC-MODEX for the client side of simple XOR case study.

7 SOUNDNESS OF CRYPTOBAP'S APPROACH

The extracted *IML* model should preserve the *BIR* program's behaviors to ensure that we can transfer the verified properties back to the binary of protocols.

7.1 Soundness of translation into *IML*

To show that our extracted *IML* model preserves the semantics of the protocols' binary, we need to prove that our translation from a crypto-aware symbolic execution tree into an *IML* process is sound, i.e., *for each path in the symbolic execution tree there is an equivalent *IML* execution trace.*

Our symbolic execution supports communication with the attacker, which, like honest protocol parties given by specification, is represented as an *IML* process. Thus, we need to prove soundness in the context of an *IML* process, i.e., that each execution trace obtained by symbolically executing the *BIR* program in parallel with an *IML* attacker has an equivalent *IML* trace where the translated processes run in parallel with the same attacker. Our strategy to prove this is to construct an *IML-SBIR*, *IML_{SB}*, mixed execution semantics to facilitate the communication of *BIR* programs and the *IML* attacker. *IML_{SB}* is generic and considers *BIR* programs and *IML* processes as independent entities running parallel and communicating through a channel.

The *IML* process already describes the parallel execution of parties and how they share secrets. We only need to integrate *BIR* into this framework. Therefore, we extend *IML* with a construct $run(pc, (y_1, \dots, y_m))$ to initialize *BIR* symbolic memory and transfer control to the *BIR* program specified by the pc . To share the secrets, we generate fresh symbolic values y_1, \dots, y_m and store them in the environment η of the *BIR* program.

In the following, we use $I\{P\}$ as a pair of an *IML* process I extended with the run construct and a *BIR* program P that defines the entry points therein. Slightly misusing notation, $I\{s_i\}$ also denotes states of the mixed semantics.

Fig. 12 shows the operational semantics of *IML_{SB}* which combines *IML* input and output processes [2, p. 23] with the transition relations of *SBIR* in Fig. 9. In the figure, rules *ItoSB* and *SBtoI* define the communication between the symbolic *BIR* program and the *IML* process. Using these rules a protocol participant can receive a sent message if its channel identifiers have the same evaluation as the channel identifiers of the sender. To send a message, i.e., when pc is in the label set $\xi_{\mathcal{A}_s}$, we first fetch the symbolic value e from the memory location r_0 , truncate the interpretation of the message according to the maximum length of the *IML* channel c ,² and then place it in the *IML* environment η . When pc is in $\xi_{\mathcal{A}_r}$ and the *BIR* program receives input from the *IML* channel c , we receive the truncated bitstring b' from an *IML* state and generate a fresh symbolic value e such that the interpretation of e is equal to bitstring b' . Then, we store the symbolic value e in the memory and return its address in r_0 .

²A requirement from CSEC-MODEX's correctness proof for the translation to ProVerif and CryptoVerif. As the attacker's polynomial bound is chosen after the process, the attacker could send a large message that the process runs out of time reading it.

CSEC-MODEX's IML model	CRYPTOBAP's IML model
<pre> assume argv0 = argv0 in : new var1: fixed_20; let nonce1 = var1 in assume Defined(pad) in assume len(pad) = 21 in let xor1 = XOR((1)^[u,1] nonce1, pad) in : let msg1 = xor1 in out(c, msg1); </pre>	<pre> new OTP_48: fixed_64; let Conc1_66 = conc1(OTP_48) in let XOR_70 = exclusive_or(Conc1_66, pad) in out(c, XOR_70); </pre>

Figure 11: IML models produced by CRYPTOBAP vs. CSEC-MODEX for the client side of simple XOR case study

$$\begin{array}{c}
\text{pc} \in \mathcal{L}_{\mathcal{N}} \uplus \xi_{\text{Op}} \uplus \xi_{\mathcal{R}} \uplus \xi_{\mathcal{E}} \quad (\phi, \eta, \text{pc}) \xrightarrow{a} (\phi', \eta', \text{pc}') \\
\hline
\text{P} \vdash (\phi, \eta, \text{pc}), Q^{i^{\text{XS}}} \xrightarrow{a} {}_{i,H}(\phi', \eta', \text{pc}'), Q^{i^{\text{XS}}} \quad \text{normal}
\end{array}$$

$$\begin{array}{c}
\llbracket e \rrbracket_{\eta} = b \neq \perp \quad b' = \text{truncate}(b, \text{maxlen}(c)) \quad \forall j \leq m : \llbracket e_j \rrbracket_{\eta} = b_j \neq \perp \quad Q^{i^{\text{XS}'}} = \text{reduce}(\{(\eta, Q)\}) \\
\exists!((\phi, \eta, \text{pc}), c[e_1, \dots, e_m]) \in Q^{i^{\text{XS}}} : \text{pc} \in \xi_{\mathcal{A}_r} \wedge \forall j \leq m : H(\eta[e_j]) = b_j \neq \perp \quad \text{pc}' = \text{ret}(\phi, \eta, \text{pc}) \\
e \notin \text{range}(\eta) \quad H(e) = b' \quad (\eta', a) = \text{mstore}(\eta, \text{heap}_{\mathcal{A}}, \text{Mem}_{\mathcal{A}}, e, 128) \quad Q^{i^{\text{XS}''}} = \{(\phi, \eta, \text{pc}), c[e_1, \dots, e_m]\} \\
\hline
\text{P} \vdash (\eta, \text{out}(c[e_1, \dots, e_m], e); Q), Q^{i^{\text{XS}}} \xrightarrow{\text{In}(e, (e_1, \dots, e_m))} {}_{i,H}((\phi, \eta'[r_0 \mapsto a], \text{pc}'), c[e_1, \dots, e_m]), Q^{i^{\text{XS}}} \uplus Q^{i^{\text{XS}'}} \setminus Q^{i^{\text{XS}''}} \quad \text{ItoSB}
\end{array}$$

$$\begin{array}{c}
\text{pc} \in \xi_{\mathcal{A}_s} \quad \text{pc}' = \text{ret}(\phi, \eta, \text{pc}) \quad Q^{i^{\text{XS}'}} = \{(\phi, \eta, \text{pc}')\} \quad a = \eta[r_0] \quad e = \text{mload}(\eta, a) \quad H(e) = b \neq \perp \quad b' = \text{truncate}(b, \text{maxlen}(c)) \\
\forall j \leq m : H(\eta[e_j]) = b_j \neq \perp \quad \exists!((\eta, Q) \in Q^{i^{\text{XS}}} : Q = \text{in}(c[e_1, \dots, e_m], x); P \wedge \forall j \leq m : \llbracket e_j \rrbracket_{\eta} = b_j \neq \perp \quad Q^{i^{\text{XS}''}} = \{(\eta, Q)\} \\
\hline
\text{P} \vdash ((\phi, \eta, \text{pc}), c[e_1, \dots, e_m]), Q^{i^{\text{XS}}} \xrightarrow{\text{Out}(e, (e_1, \dots, e_m))} {}_{i,H}(\eta[x \mapsto b'], P), Q^{i^{\text{XS}}} \uplus Q^{i^{\text{XS}'}} \setminus Q^{i^{\text{XS}''}} \quad \text{SBtoI}
\end{array}$$

$$\begin{array}{c}
\forall j \leq m : \llbracket y_j \rrbracket_{\eta} = b_j \neq \perp \quad (y_1, \dots, y_m) \notin \text{range}(\eta_0) \quad \forall j \leq m : H(y_j) = b_j \neq \perp \quad \forall j \leq m : (\eta_{j+1}, a_i) = \text{mstore}(\eta_j, \text{heap}, \text{Mem}, y_j, 128) \\
\hline
\text{P} \vdash (\eta, \text{run}(\text{pc}, (y_1, \dots, y_m))), Q^{i^{\text{XS}}} \xrightarrow{} {}_{i,H}(\phi, \eta_{m+1}[r_0 \mapsto a_1, \dots, r_m \mapsto a_m], \text{pc}), Q^{i^{\text{XS}}} \quad \text{run}
\end{array}$$

Figure 12: The mixed semantics of SBIR and IML shown by IML_{SB}.

We use the standard notion of *trace inclusion* to show the translations' soundness (see Thm. 7.3), i.e., the set of IML_{SB} execution traces is a subset of the IML execution traces. To prove this formally, we define a simulation relation $\sim_{H, \langle \cdot, \cdot \rangle} \subseteq S^{i^{\text{XS}}} \times S$ between states/events of these two abstraction layers and show that it is preserved by the single-step executions. The simulation relation, $I\{s_i\} \sim_{H, \langle \cdot, \cdot \rangle} s_i$, checks if (i) the IML output process in the given IML state is the correct translation of the symbolic state in T according to the rules in Fig. 10, i.e., $s_i.P = \langle \llbracket T[s_i, \text{pc}] \rrbracket \rangle$, and (ii) the environments of the two abstractions are related through the interpretation H , i.e., for all $x \in \text{dom}(I\{s_i\}.\eta)$ there are $x \in \text{dom}(s_i.\eta)$ and an H s.t. $H(I\{s_i\}.\eta[x]) = s_i.\eta[x]$. Lemma 7.1 shows that the initial states of IML_{SB} and the derived IML process are in the relation.

LEMMA 7.1. *For a symb. execution tree T of the BIR program P, an IML process I and any $k \in \mathcal{N}$ the size of the random memory, let $I\{s_0\} = (\text{True}, \eta_0[\text{RM} \mapsto \text{rm}_k, \text{r}_k \mapsto 0], \text{pc}_0)$ be an initial symbolic state in IML_{SB} and $s_0 = (\eta_0, Q^{\text{full}})$ the corresponding initial IML state. Then, for all $H: I\{s_0\} \sim_{H, \langle \cdot, \cdot \rangle} s_0$.*

Next, we show that single-step transitions preserve the simulation relation.

LEMMA 7.2 (STATE/EVENT EQUIVALENCE). *Let P be a BIR program and I be an IML process, then, for all $s_i, I\{s_i\}, I\{s_j\}$ and H s.t. $I\{s_i\} \sim_{H, \langle \cdot, \cdot \rangle} s_i$ and $I\{s_i\} \xrightarrow{a^{i^{\text{XS}}}} {}_{p,H} I\{s_j\}$, there exist an H' and s_j s.t. $H \subseteq H', s_i \xrightarrow{a} {}_p s_j$ and $I\{s_j\} \sim_{H', \langle \cdot, \cdot \rangle} s_j$ and if $a \neq \perp$ then $a^{i^{\text{XS}}} =_{H'} a$.*

We then show the translation's soundness by extending the simulation relation to execution traces, i.e., $\sim_{H, \langle \cdot, \cdot \rangle, k} \subseteq \mathcal{R}^{i^{\text{XS}}} \times \mathcal{R}^i$, w.r.t an upper bound³ $k \in \mathcal{N}$ on the number of RNG steps of the execution $\text{rng} : R \rightarrow \mathcal{N}$. That is, $R^{i^{\text{XS}}} \sim_{H, \langle \cdot, \cdot \rangle, k} R$ holds iff: $\text{rng}(R^{i^{\text{XS}}}) \leq k$ and for all $I\{s\}, a^{i^{\text{XS}}} \in R^{i^{\text{XS}}}$ there exist $s, a \in R$, and H s.t. $I\{s\} \sim_{H, \langle \cdot, \cdot \rangle} s$ and $a^{i^{\text{XS}}} =_H a$.

Finally, we show that executions of the mixed IML and symbolic execution and IML preserve the simulation relation. Note that, in the following, we assume a single BIR program that implements different protocol participants with distinct sets of program counters.

³This bound k is needed because of BIR's finite memory model.

The results can be extended to multiple BIR programs, as presented in Appendix A.7.

THEOREM 7.3 (TRACE INCLUSION). *Let P be a BIR program, I be an IML process, and $k \in \mathcal{N}$ is any upper bound on the number of RNG steps, then, for all mixed IML and symb. execution traces $R^{I \times S} \in \mathcal{R}^{I \times S}(I\{P\}, \eta_0[RM \mapsto rm_k, r_k \mapsto 0])$ s.t. $rng(R^{I \times S}) \leq k$, there are an IML trace $R \in \mathcal{R}^I(I\{P\})$ and an H s.t. $R^{I \times S} \sim_{H, k} R$.*

PROOF. The goal is to show that for all IML_{SB} traces, there is an equivalent IML trace that are in the simulation relation through the interpretation H . We prove the theorem by induction on the length of the execution traces:

- **Base case.** Follows from Lem. 7.1.
- **Inductive case.** Follows from Lem. 7.2.

We present the proof of lemmas 7.1 and 7.2 in Appendix A.4. \square

Thm. 7.3 is the first step to relating the properties we verify for the IML model to the actual binary of the protocol. We show that the IML model resulting from translation covers all behaviors in the IML_{SB} semantics. Recall that we have to talk about behavioral properties in the mixed semantics (as opposed to the pure BIR semantics) as protocol properties typically concern more than one party. Next, we show that these symbolic behaviors cover all concrete behaviors.

7.2 Soundness of Symbolic Execution

To ensure that the extracted IML model preserves the semantics of the protocol's binary, we have to prove further that our symbolic execution is behaviorally equivalent to the transpiled BIR code. To show this, we construct a mixed IML-BIR execution semantics, hereafter IML_B, that allows the BIR program to communicate with the same IML attacker at the IML_{SB} level. The IML_B execution semantics is presented in Appendix A.3-Fig 16—the rules are similar and have the same meaning as those defined for IML_{SB}.

Our proof strategy to show the behavioral equivalence of IML_B and IML_{SB} is similar to our technique to prove the soundness of the IML translation. That is, we first show the state/event equivalence between the two abstractions and then use this to prove the trace inclusion of IML_B in IML_{SB}.

We show the state/event equivalence by extending the simulation relation of Property 1 to a relation $\sim_H \subseteq S^{I \times b} \times S^{I \times S}$ between IML_B and IML_{SB}. The relation $I\{s_i\} \sim_H I\{s_i\}$ checks that $I\{s_i\}.pc = I\{s_i\}.pc$, and for all $x \in dom(I\{s_i\}.\eta)$ there exist $x \in dom(I\{s_i\}.\eta)$ and an interpretation H s.t. $H(I\{s_i\}.\eta[x]) = I\{s_i\}.\eta[x]$. The first step in showing this simulation relation between the two layers is to prove that the initial states are in the relation using Lem. 7.4:

LEMMA 7.4. *For a BIR program P , an IML process I and any upper bound $k \in \mathcal{N}$ on the number of RNG steps, let $I\{s_0\} = (\eta_0[RM \mapsto rm_k, r_k \mapsto 0], pc_0)$ be an initial BIR state in IML_B and $I\{s_0\} = (True, \eta_0[RM \mapsto rm_k, r_k \mapsto 0], pc_0)$ be the corresponding initial state in IML_{SB}. Then, $I\{s_0\} \sim_H I\{s_0\}$ for all H .*

We then prove that the single-step transitions of IML_B and IML_{SB} preserve the simulation relation using Lem. 7.5.

LEMMA 7.5 (STATE/EVENT EQUIVALENCE). *Let P be a BIR program and I be an IML process, then, for all $I\{s_i\}, I\{s_i\}, I\{s_j\}$ and H s.t.*

$$I\{s_i\} \sim_H I\{s_i\} \text{ and } I\{s_i\} \xrightarrow{a^{I \times b}}^+ P I\{s_j\}, \text{ there exist an } H' \text{ and } I\{s_j\} \text{ s.t.} \\ H \subseteq H', I\{s_i\} \xrightarrow{a^{I \times S}}^+ P, H' I\{s_j\}, I\{s_j\} \sim_{H'} I\{s_j\} \text{ and } a^{I \times b} =_{H'} a^{I \times S}.$$

We show the behavioral equivalence between the two layers by extending the simulation relation to execution traces $\sim_{H, k} \subseteq \mathcal{R}^{I \times b} \times \mathcal{R}^{I \times S}$ w.r.t an upper bound $k \in \mathcal{N}$ on the number of RNG steps. That is, $R^{I \times b} \sim_{H, k} R^{I \times S}$ holds, iff, $rng(R^{I \times b}) \leq k$, and for all $I\{s\}, a^{I \times b} \in R^{I \times b}$ there exist $I\{s\}, a^{I \times S} \in R^{I \times S}$ and an H s.t. $I\{s\} \sim_H I\{s\}$ and $a^{I \times b} =_H a^{I \times S}$.

THEOREM 7.6 (TRACE INCLUSION). *Let P be a BIR program, I be an IML process, and $k \in \mathcal{N}$ is any upper bound on RNG steps, then, for all IML_B traces $R^{I \times b} \in \mathcal{R}^{I \times b}(I\{P\}, \eta_0[RM \mapsto rm_k, r_k \mapsto 0])$ s.t. $rng(R^{I \times b}) \leq k$, there are an IML_{SB} trace $R^{I \times S} \in \mathcal{R}^{I \times S}(I\{P\}, \eta_0[RM \mapsto rm_k, r_k \mapsto 0])$ and an H s.t. $R^{I \times b} \sim_{H, k} R^{I \times S}$.*

PROOF. Thm. 7.6 shows that for all IML_B traces, there is an equivalent IML_{SB} trace through a properly chosen interpretation H . We prove Thm. 7.6 by induction on the length of the traces.

- **Base case.** The base case can be proved using Lem. 7.4.
- **Inductive case.** The inductive step can be proved using Lem. 7.5.

Appendix A.5 presents the proof of lemmas 7.4 and 7.5. \square

Thm. 7.6 shows that, for an appropriately chosen interpretation and random memory, symbolic and concrete executions of a BIR program are behaviorally equivalent. This holds in the mixed IML-(S)BIR semantics, i.e., when coupled with the same IML attacker and protocol partners.

8 SECURITY PROPERTIES

From the simulation results between concrete BIR, symbolic BIR and extracted IML, we will now conclude our target result, which argues that probabilistic security results translate across these levels of abstraction. The security properties we consider, i.e., authentication and weak secrecy, are safety properties over event traces. Specifically, we consider a security property ψ as a polynomially decidable prefix-closed set of event traces.

Example. For SSH, we show authentication between the events $Acpt_S(PK_S, PK_C)$ (in the server model derived from the TinySSH binary) and $Acpt_C(PK_S, PK_C)$ (in the client model implemented based on the SSH specification) where PK_S and PK_C are the server's public key and the client's public key, respectively.

$$\text{Auth} = \{t \in \psi \mid \forall i \in \mathcal{N} : t[i] = \text{Accept}_S(PK_S, PK_C) \\ \implies (\exists j \in \mathcal{N} : j < i \wedge t[j] = \text{Accept}_C(PK_S, PK_C))\}$$

We quantify the probability of a protocol remaining secure by considering the complementary probability: the sum of the probabilities of each violation. To avoid double counting, we only sum over the set of shortest violating prefixes, i.e., $\psi_\neg = \{t \notin \psi \mid \forall t'. t' \text{ is prefix of } t \implies t' \in \psi\}$. As security properties are prefix-closed, this captures the probability of a violation. The system we analyze consists of the protocol implementations in BIR. Say T^α denotes a set of event traces obtained from the respective set of execution traces $\mathcal{R}^\alpha \downarrow_{\text{events}}$, pr is a probability distribution function that computes the probability of an event trace and BS_n^k is a set of bit strings for generating k random numbers of length n , then:

Definition 8.1 (BIR insecurity). For a BIR program P , an IML process I , a security parameter $n \in \mathcal{N}$, and $k \in \mathcal{N}$ the size of BIR's random memory, the insecurity of $I\{P\}$ w.r.t. ψ is: $\text{insec}(I\{P\}, n, k, \psi) = 2^{-n \cdot k} \cdot \sum_{\substack{rm_k \in BS_n^k \\ t^{i \times b} \in T_{rm_k}^{i \times b} \cap \psi_{-}}} pr(t^{i \times b})$ where $T_{rm_k}^{i \times b} = T^{i \times b}(I\{P\}, \eta_0[RM \mapsto rm_k, r_k \mapsto 0])$.

After translating P into $\langle P \rangle$, we define insecurity in terms of IML's probabilistic semantics.

Definition 8.2 (IML insecurity). The insecurity of an IML process I w.r.t a trace property ψ and a security parameter n is $\text{insec}(I, n, \psi) = \sum_{t \in T(I, n) \cap \psi_{-}} pr(t)$ where $pr(s_0 \rightsquigarrow p_1 s_1 \cdots s_{n-1} \rightsquigarrow p_n s_n) = \prod_{1 \leq i \leq n} p_i$.

Note that definitions 8.1 and 8.2 coincide on IML_B processes that do not contain the *run*-construct, as in this case, the RNG rule (like any other BIR rule) can never be applied and thus k be chosen to be 0. This applies to the IML_B processes resulting from our translation. Thm. 8.3 shows the translation is sound. Note that I contains BIR programs (via the *run* construct), but also IML processes that represent communication partners and the network attacker.

THEOREM 8.3 (TRANSLATION PRESERVES ATTACKS). *Given a BIR program P , an IML process I , a security parameter $n \in \mathcal{N}$, a trace property ψ and an upper bound $k \in \mathcal{N}$ on the number of RNG steps in $T^{i \times b}(I\{P\}, \eta_0[RM \mapsto rm_k, r_k \mapsto 0])$, we get that*

$$\text{insec}(I\{P\}, n, k, \psi) \leq \text{insec}(I\{\langle P \rangle\}, n, \psi).$$

Via [2, Thm. 4.3, Thm. 5.2] we obtain a bound for $\text{insec}(I\{\langle P \rangle\}, n, \psi)$ from either of the backends, ProVerif or CryptoVerif. In cryptography, probability bounds are expressed as asymptotic functions in the security parameter. CryptoVerif provides a symbolic expression of such a probability bound and, furthermore, proves that the bound is negligible, i.e., it decreases faster than the inverse of any polynomial. On the other hand, ProVerif only confirms the existence of a negligible bound. In both cases, the existence of this negligible upper bound ensures $\text{insec}(I\{P\}, n, k, \psi)$ is negligible.

We present the proof of Thm. 8.3 in Appendix A.6. Based on definitions 8.1 and 8.2, we calculate the probability distribution in both IML and IML_B . While the probability of all transitions except for random number generation is 1, we need to demonstrate other requirements, such as extra randomness, injective event trace inclusion, etc. To this end, we present lemmas in Sec. A.6.1 that show these requirements, which is necessary to prove Thm. 8.3.

9 EVALUATION

We have implemented CRYPTOBAp on the HOL4 theorem prover [41] using its metalanguage SML. CRYPTOBAp relies on HolBA's semantics-preserving transpiler and symbolic execution [52, 53]. We significantly extended the HolBA vanilla symbolic execution to handle crypto primitives, communication with the attacker, indirect jumps, and loops, which are essential to verify the security of protocols. We also adapted the CSEC-MODEX's pipeline to process CRYPTOBAp-generated IML. Table 2 shows the list of protocol implementations that we used in our evaluation.

Simple case studies. Apart from the XOR case study discussed in Sec. 6.1, we also verified other case studies of [3, 4] to evaluate

CRYPTOBAp. The only exception to [3, 4] is *smart metering protocol* which is not open source. For other cases, we obtained the same result. Additionally, in contrast to [3], which could not handle CSur, we successfully verified this case study.

RPC implements the MAC-based remote procedure call protocol [10]. We verified the client request and the server response authenticity under the MAC unforgeability assumption against chosen-message attacks with symbolic and computational guarantees. RPC-enc is an implementation of the RPC protocol that uses authenticated encryption. We also verified the secrecy of the payloads (which is not protected by the MAC-based RPC) with an assumption that authenticated encryption is indistinguishable against the chosen-plaintext attack and provides ciphertext integrity. CSur is the Needham-Schroeder public-key authentication protocol [61]. We verified the secrecy and authentication properties for the CSur binary. Our analysis confirmed that CSur is vulnerable to attack in [56] and leaks protocol parties' nonces. Similar to [4], we also removed the assumption (i.e., all cryptographic material plus nonce are tagged) used in [3] for the Needham-Schroeder-Lowe (NSL) case study to obtain the computational soundness result. We confirmed the flaw in the protocol discovered in [4]: if the nonce of the second party is not tagged and is sent separately, it can be (mis)used as the first protocol message.

Simple MAC implements the first half of the RPC protocol in which a single payload is concatenated with its MAC [10]. We verified the payload authenticity under the unforgeability of MAC against the chosen-message attack assumption. For simple XOR case study, we verified the secrecy of the payload with CryptoVerif. We did not attempt verification with ProVerif, as the analysis of theories with XOR requires extra effort [47], while CryptoVerif's attacker model is strictly stronger.

Verification of TinySSH and WireGuard. We further evaluated CRYPTOBAp by verifying TinySSH and WireGuard protocols. TinySSH is a minimalistic SSH server that implements a subset of SSHv2 features and ships with its own crypto library. To formulate authentication properties, which ought to hold for any communication partner to the TinySSH server that conforms to the SSH protocol specification, we modeled the client side of the SSH protocol in IML; agents at the other end of the communication line are manually developed. We manually specified the cryptographic assumptions about the primitives used by the TinySSH implementation in CryptoVerif and ProVerif templates. We verified mutual authentication with ProVerif and CryptoVerif.

WireGuard implements virtual private networks akin to IPsec and OpenVPN. It is quite recent and was incorporated into the Linux kernel (stable) in March 2020. We have automatically extracted, *for the first time*, the WireGuard's ProVerif model from its linux implementation binary—all other existing models are hand-written and derived from the specification [33, 44, 54]. Existing WireGuard's symbolic models often utilize pattern matching to verify authentication. This involves constructing the entire message from the initial stage and subsequently comparing it to the received message from the network, which requires further justification. In contrast, our extracted model from the implementation closely represents the actual behavior of WireGuard, as it deconstructs the network message using ChaCha20Poly1305 Decryption for the purpose of

Protocol	ARM Loc	Verified Code Size	Feasible Path	Infeasible Path	IML Loc	CryptoVerif (CV) Loc	ProVerif (PV) Loc	Time (Second)			Verified in	Primitives
								IML	CV	PV		
RPC	1.8K	0.659K	8	178	23	236	102	10	0.035	0.012	CV & PV	UF-CMA MAC
RPC-enc	53K	0.294K	28	348	53	313	118	9	0.073	0.047	CV & PV	IND-CPA INT-CTXT AE
CSur	0.7K	0.382K	11	237	29	277	177	6	0.656	0.035	flaw	IND-CCA2 PKE
NSL	2.8K	0.595K	23	455	56	296	204	35	2.740	0.052	flaw, verified	IND-CCA2 PKE
Simple MAC	1.5K	0.294K	13	149	29	207	101	8	0.047	0.033	CV & PV	UF-CMA MAC
Simple XOR	2.4K	0.100K	2	50	7	141	–	4	0.40	–	CV	XOR
TinySSH	18K	0.476K	136	1079	87	286	190	55	0.077	0.079	CV & PV	CRHF-CDH [†] & UF-CMA SIGN
WG	Initiator	27K	1.323K	68	1482	181	222	52	–	59.646	PV	DH-X25519 [‡] & ROM-hash [★]
	Responder											
			153	1389	464	–		47				

Table 2: Case studies. ARM assembly includes crypto code and the code that is used in preprocessing, e.g., to compute the control flow of the program required in loop summarization. Primitives are standard, except Collision-resistant hash based on computational Diffie-Hellman ([†]), Curve25519 [49] ([‡]) and hash function in the Random Oracle Model [9] ([★]). Also, WG = WireGuard.

authenticating ChaCha20Poly1305 Aauthenticated Encryption with Associated Data (AEAD). As a result, our model obviates the need for additional caution when verifying the authentication property. We model the handshake and first transport message, after which the key exchange is concluded, and prove the protocol participants mutually agree on the resulting keys.

TinySSH and WireGuard are case studies with indirect jumps in their binary, and TinySSH is the only one that features multiple sessions. The other case studies from [3, 4] do not contain loops as [3, 4] did not support them. We handle the loops inside the implementation of TinySSH using the summarization technique Sec. 5, which is automated by CRYPTO-BAP—these loops handle, e.g., reading key files of variable size. TinySSH spawns a new server for each incoming TCP connection by using `inetd`, which we correctly cover in our template files. We hence handle multiple sessions, including potential replay attacks, correctly.

10 CONCLUDING REMARKS

We introduced CRYPTO-BAP to analyze the binary of cryptographic protocols. CRYPTO-BAP enables sound verification of authentication and weak secrecy for protocols’ machine code using model extraction technique. We have automated extracting models from the binary of protocols and formally proved that the extracted IML model preserves the protocol’s behavior at its machine-code level. Additionally, we showed that the verified properties for the IML model could be transferred back up to a mixed execution semantics, where the BIR translation of the protocol can be coupled and communicate with an IML attacker.

The problem of deciding secrecy is generally undecidable [59] even in dedicated protocol specification languages whose semantics are designed for verification. Hence the verification tool at the backend must necessarily sacrifice automation or completeness. In practice, ProVerif sacrifices completeness but verifies many typical protocols. It is as easy to construct programs that are not recognized as secure by ProVerif as it is to write programs that purposefully break our abstractions in symbolic execution (e.g., by using a randomly generated key to guide the control flow). Lacking a syntactic criterion of what protocol implementations should or should not do, we have evaluated the automation and completeness of CRYPTO-BAP via case studies while ensuring soundness through proofs.

The current implementation of CRYPTO-BAP has a few limitations. At the protocol verification level, our limits are currently those

of the protocol verification backend. These are constantly improving and we are currently working on targeting Tamarin [58] as well. An example of our limitations at the code handling level is **infinite loops**. The soundness of handling loops using the replication operator $!^m P$ can hold as long as enough randomness can be stored in the memory, which is finite. Hence, the number of RNG steps is limited by BIR’s finite memory in combination with its deterministic semantics. Thus, BIR needs to be extended with external non-determinism to handle infinite loops. Moreover, similar to other related works, we trust the implementation of cryptographic primitives. This means if there is any bug in these primitives that violates the protocols’ security, these violations will not be detected. The exclusion of these primitives from our verification is because their verification requires a different methodology (e.g., weakest precondition propagation). This is different from the goal we set ourselves in the paper. Also, while the CRYPTO-BAP’s approach is generic, at present, we can analyze only ARM and RISC-V binaries. To extend our analysis to other architecture, the only part that needs to be extended is the HolBA transpiler. Finally, we have tried to reduce manual effort, yet, there still remains a necessity for the user to initialize and steer the verification process. Mainly, the user specifies (i) The code fragments they want to verify: the code-under-analysis to the lifter, and the function names that are trusted (libraries) or untrusted (attacker/network) to the symbolic execution engine; (ii) the symbolic model of cryptographic functions, and (iii) the security assumptions on the cryptographic primitives and security queries in the verifiers’ template files.

In the future, we plan to mechanize our proofs in a proof assistant such as HOL4 [41]. Moreover, to better cover top-level loops, it is necessary to handle non-monotonous global states, i.e., state changes that alter how a protocol reacts to future messages. This problem was recognized in protocol verification, for instance, in the more recent Tamarin verifier [58] or ProVerif’s global state extensions [28]. It would thus be promising to skip Aizatulin’s toolchain altogether and target one of these tools. Tamarin’s multiset-rewrite calculus is well-suited for modeling state machines. Moreover, one could essentially reuse the state-access axioms from SAPIC [45] to handle persistent data stored in the heap.

ACKNOWLEDGMENTS

This work was supported in part by a gift from Intel; and by the German Federal Ministry of Education and Research (BMBF) (FKZ:

13N1S0762). We thank the anonymous reviewers for their valuable feedback during the review process. We also thank Andreas Lindner for helping with the CRYPTOAP implementation.

REFERENCES

- [1] Mihail Aizatulin. 2011. Csec-Modex. <https://github.com/tari3x/csec-modex>.
- [2] Mihail Aizatulin. 2015. *Verifying Cryptographic Security Implementations in C Using Automated Model Extraction*. Ph.D. Dissertation. The Open University. <http://arxiv.org/abs/2001.00806>
- [3] Mihail Aizatulin, Andrew D. Gordon, and Jan Jürjens. 2011. Extracting and Verifying Cryptographic Models from C Protocol Code by Symbolic Execution. In *Proceedings of the 18th ACM Conference on Computer and Communications Security - CCS '11*. ACM Press, Chicago, Illinois, USA, 331.
- [4] Mihail Aizatulin, Andrew D. Gordon, and Jan Jürjens. 2012. Computational Verification of C Protocol Implementations by Symbolic Execution. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security - CCS '12*. ACM Press, Raleigh, North Carolina, USA, 712.
- [5] Michael Backes, Matthias Berg, and Dominique Unruh. 2008. A Formal Language for Cryptographic Pseudocode. In *Logic for Programming, Artificial Intelligence, and Reasoning* (Berlin, Heidelberg), Iliano Cervesato, Helmut Veith, and Andrei Voronkov (Eds.). Springer, 353–376.
- [6] Michael Backes, Matteo Maffei, and Dominique Unruh. 2010. Computationally sound verification of source code. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4–8, 2010*. 387–398.
- [7] Gilles Barthe, Benjamin Gregoire, Sylvain Heraud, and Santiago Zanella Beguelin. 2011. Computer-Aided Security Proofs for the Working Cryptographer. In *Advances in Cryptology – CRYPTO 2011* (Berlin, Heidelberg), Phillip Rogaway (Ed.). Springer, 71–90.
- [8] Gilles Barthe, Benjamin Gregoire, and Santiago Zanella Beguelin. 2009-01-21. Formal certification of code-based cryptographic proofs. 44, 1 (2009-01-21), 90–101.
- [9] Mihir Bellare and Phillip Rogaway. 1993. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*. 62–73.
- [10] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D Gordon, and Sergio Maffei. 2011. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 2 (2011), 1–45.
- [11] Lennart Beringer, Adam Petcher, Q Ye Katherine, and Andrew W Appel. 2015. Verified correctness and security of OpenSSL {HMAC}. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 207–221.
- [12] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. 2015. Verified Correctness and Security of OpenSSL HMAC (SEC'15). USENIX Association, USA, 207–221.
- [13] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. 2017-05. Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate. In *2017 IEEE Symposium on Security and Privacy (SP)*. 483–502. ISSN: 2375-1207.
- [14] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, et al. 2017. Everest: Towards a verified, drop-in replacement of HTTPS. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
- [15] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. 2014. Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18–21, 2014*. 98–113.
- [16] Karthikeyan Bhargavan, Cédric Fournet, Ricardo Corin, and Eugen Zalinescu. 2008. Cryptographically verified implementations for TLS. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27–31, 2008*. 459–468.
- [17] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. 2006. Verified Reference Implementations of WS-Security Protocols. In *Web Services and Formal Methods, Third International Workshop, WS-FM 2006 Vienna, Austria, September 8–9, 2006, Proceedings*. 88–106.
- [18] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. 2010. Modular Verification of Security Protocol Code by Typing. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Madrid, Spain) (POPL '10)*. Association for Computing Machinery, New York, NY, USA, 445–456.
- [19] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. 2008. Verified interoperable implementations of security protocols. *ACM Trans. Program. Lang. Syst.* 31, 1 (2008), 5:1–5:61.
- [20] B. Blanchet. 2001-06. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001*. 82–96. ISSN: 1063-6900.
- [21] Bruno Blanchet. 2008. A Computationally Sound Mechanized Prover for Security Protocols. 5, 4 (2008), 193–207.
- [22] Bruno Blanchet. 2011. Using Horn Clauses for Analyzing Security Protocols. (2011), 86–111. <https://doi.org/10.3233/978-1-60750-714-7-86>
- [23] Matthieu Bloch and Joao Barros. 2011. *Physical-layer security: from information theory to security engineering*. Cambridge University Press.
- [24] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K Rustan M Leino, Jacob R Lorch, Bryan Parno, Ashay Rane, Srinath TV Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code.. In *USENIX Security Symposium*, Vol. 152.
- [25] Juan Caballero and Dawn Song. 2013. Automatic protocol reverse-engineering: Message format extraction and field semantics inference. *Comput. Networks* 57, 2 (2013), 451–474.
- [26] David Cadé and Bruno Blanchet. 2012. From Computationally-proved Protocol Specifications to Implementations. In *Seventh International Conference on Availability, Reliability and Security, Prague, ARES 2012, Czech Republic, August 20–24, 2012*. 65–74.
- [27] Sagar Chaki and Anupam Datta. 2009. ASPiER: An Automated Framework for Verifying Security Protocol Implementations. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8–10, 2009*. 172–185.
- [28] Vincent Cheval, Veronique Cortier, and Mathieu Turuani. 2018. A Little More Conversation, a Little Less Action, a Lot More Satisfaction: Global States in ProVerif. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, Oxford, 344–358.
- [29] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. 2017-10-30. A Comprehensive Symbolic Analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (New York, NY, USA) (CCS '17)*. Association for Computing Machinery, 1773–1788.
- [30] Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, and Oliver Schwarz. 2013. Formal verification of information flow security for a simple arm-based separation kernel. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4–8, 2013*. 223–234.
- [31] D. Dolev and A. C. Yao. 1981. On the Security of Public Key Protocols. In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science (USA) (SFC'81)*. IEEE Computer Society, 350–357.
- [32] Jason A. Donenfeld. 2017. WireGuard: Next Generation Kernel Network Tunnel. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. <https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/wireguard-next-generation-kernel-network-tunnel/>
- [33] Jason A Donenfeld and Kevin Milner. 2017. Formal verification of the WireGuard protocol. *Technical Report, Tech. Rep.* (2017).
- [34] François Dupressoir, Andrew D. Gordon, Jan Jürjens, and David A. Naumann. 2014. Guiding a general-purpose C verifier to prove cryptographic protocols. *J. Comput. Secur.* 22, 5 (2014), 823–866.
- [35] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2020. Simple High-Level Code For Cryptographic Arithmetic: With Proofs, Without Compromises. *ACM SIGOPS Operating Systems Review* 54, 1 (2020), 23–30.
- [36] Benoît Fraikin, Marc Frappier, and Richard St-Denis. 2014. Supervisory control theory with Alloy. *Sci. Comput. Program.* 94 (2014), 217–237.
- [37] Shafi Goldwasser and Silvio Micali. 1984. Probabilistic Encryption. *J. Comput. Syst. Sci.* 28, 2 (1984), 270–299.
- [38] Jean Goubault-Larrecq and Fabrice Parrennes. 2005. Cryptographic Protocol Analysis on Real C Code. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17–19, 2005, Proceedings*. 363–379.
- [39] Shai Halevi. 2005. A plausible approach to computer-aided cryptographic proofs. <https://eprint.iacr.org/2005/181>
- [40] Xudong He, Qin Liu, Shuang Chen, Chin-Tser Huang, Dejun Wang, and Bo Meng. 2020. Analyzing Security Protocol Web Implementations Based on Model Extraction With Applied PI Calculus. *IEEE Access* 8 (2020), 26623–26636.
- [41] HOL development team. 2022. HOL Interactive Theorem Prover. <https://hol-theorem-prover.org>
- [42] Jan Jürjens. 2009. Automated Security Verification for Crypto Protocol Implementations: Verifying the Jessie Project. *Electron. Notes Theor. Comput. Sci.* 250, 1 (2009), 123–136.
- [43] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2010. seL4: formal verification of an operating-system kernel. *Commun. ACM* 53, 6 (2010), 107–115.
- [44] Nadim Kobeissi, Georgio Nicolas, and Karthikeyan Bhargavan. 2019. Noise Explorer: Fully automated modeling and verification for arbitrary Noise protocols. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 356–370.
- [45] Steve Kremer and Robert Künnemann. 2016. Automated Analysis of Security Protocols with Global State. *Journal of Computer Security* 24, 5 (2016), 583–616.

- [46] Robert Künnemann and Hamed Nemati. 2020. MAC-in-the-Box: Verifying a Minimalistic Hardware Design for MAC Computation. In *Computer Security - ESORICS 2020 - 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14-18, 2020, Proceedings, Part II*. 525–545.
- [47] Ralf Küsters and Tomasz Truderung. 2008. Reducing protocol analysis with xor to the xor-free case in the horn theory based approach. In *Proceedings of the 15th ACM conference on Computer and communications security*. 129–138.
- [48] Ralf Küsters, Tomasz Truderung, and Juergen Graf. 2012. A Framework for the Cryptographic Verification of Java-Like Programs. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*. 198–212.
- [49] Adam Langley, Mike Hamburg, and Sean Turner. 2016. *Elliptic curves for security*. Technical Report.
- [50] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.
- [51] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. 2008. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*.
- [52] Andreas Lindner, Roberto Guanciale, and Mads Dam. 2023. Proof-Producing Symbolic Execution for Binary Code Verification. *CoRR* abs/2304.08848 (2023). <https://doi.org/10.48550/arXiv.2304.08848> arXiv:2304.08848
- [53] Andreas Lindner, Roberto Guanciale, and Roberto Metere. 2019. TrABin: Trustworthy analyses of binaries. *Sci. Comput. Program.* 174 (2019), 72–89.
- [54] Benjamin Lipp, Bruno Blanchet, and Karthikeyan Bhargavan. 2019. A mechanised cryptographic proof of the WireGuard virtual private network protocol. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 231–246.
- [55] Andreas Lochbihler. 2016. Probabilistic Functions and Cryptographic Oracles in Higher Order Logic. In *Programming Languages and Systems (Berlin, Heidelberg)*, Peter Thiemann (Ed.). Springer, 503–531.
- [56] Gavin Lowe. 1995. An Attack on the Needham-Schroeder Public-Key Authentication Protocol. *Inform. Process. Lett.* 56, 3 (Nov. 1995), 131–133.
- [57] Gavin Lowe. 1997. A Hierarchy of Authentication Specifications. In *Proceedings of the 10th IEEE Workshop on Computer Security Foundations (CSFW '97)*. IEEE Computer Society, USA, 31.
- [58] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. 2013. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Natasha Sharygina, and Helmut Veith (Eds.). Vol. 8044. Springer Berlin Heidelberg, Berlin, Heidelberg, 696–701.
- [59] J Mitchell, A Scedrov, N Durgin, and P Lincoln. 1999. Undecidability of bounded security protocols. In *Workshop on formal methods and security protocols*. Citeseer.
- [60] Jan Mojzisz. 2018. The SSH Library. <https://github.com/janmojzisz/tinyssh>.
- [61] Roger M. Needham and Michael D. Schroeder. 1978. Using Encryption for Authentication in Large Networks of Computers. *Commun. ACM* 21, 12 (Dec. 1978), 993–999.
- [62] Nicholas O'Shea. 2008. Using Elyjah to analyse Java implementations of cryptographic protocols. In *Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (FCS-ARSPA-WITS-2008)*.
- [63] Marco Patrignani. 2020. Why should anyone use colours? or, syntax highlighting beyond code snippets. *arXiv preprint arXiv:2001.11334* (2020).
- [64] Adam Petcher and Greg Morrisett. 2015–07. A Mechanized Proof of Security for Searchable Symmetric Encryption. In *2015 IEEE 28th Computer Security Foundations Symposium*. 481–494. ISSN: 2377-5459.
- [65] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. 2012–06. Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties. In *2012 IEEE 25th Computer Security Foundations Symposium*. 78–94. ISSN: 2377-5459.
- [66] Huzaifa Sidhpurwala. 2019. Security flaws caused by compiler optimizations. <https://www.redhat.com/en/blog/security-flaws-caused-compiler-optimizations>
- [67] Laurent Simon, David Chisnall, and Ross J. Anderson. 2018. What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. 1–15.
- [68] Jan Strejcek and Marek Trtik. 2012. Abstracting path conditions. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*. 155–165.
- [69] Xi Wang, Nikolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (Farmington, Pennsylvania) (SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 260–275.
- [70] Shameng Wen, Chao Feng, Qingkun Meng, Bin Zhang, Ligeng Wu, and Chaojing Tang. 2016. Testing Network Protocol Binary Software with Selective Symbolic Execution. In *12th International Conference on Computational Intelligence and Security, CIS 2016, Wuxi, China, December 16-19, 2016*. 318–322.
- [71] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Krügel, and Engin Kirda. 2008. Automatic Network Protocol Analysis. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*.
- [72] Ming-Ming Xiao, Shi-Long Zhang, and Yu-Ping Luo. 2016. Automatic network protocol message format analysis. *J. Intell. Fuzzy Syst.* 31, 4 (2016), 2271–2279.
- [73] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. 2023. Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs. In *Proceedings of the 32th USENIX Conference on Security Symposium (SEC'23)*. USENIX Association. <https://www.usenix.org/system/files/sec23fall-prepub-123-xu-jianhao.pdf>
- [74] Katherine Q Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W Appel. 2017. Verified correctness and security of mbedTLS HMAC-DRBG. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2007–2020.
- [75] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1789–1806.

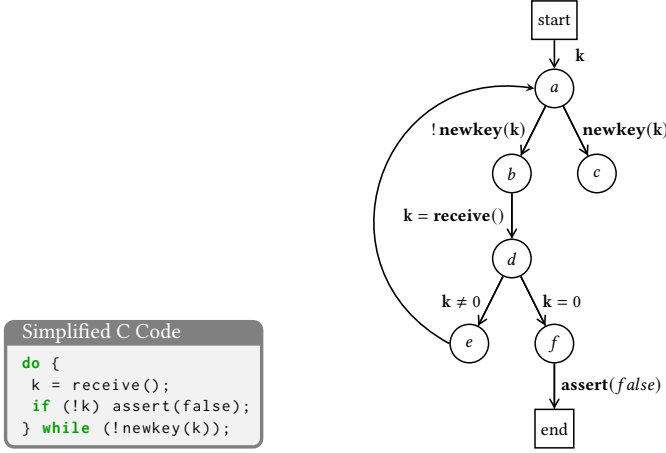


Figure 14: A simplified loop from the TinySSH binary.

Figure 15: Flowgraph of the example loop.

$$\begin{array}{c}
\frac{\llbracket e \rrbracket_\eta = \text{true}}{(\eta, \text{assume } e; P), Q \multimap_I (\eta, P), Q} \\
\\
\frac{\llbracket e \rrbracket_\eta = b \in \{BS \cup \{\perp\}\}}{(\eta, \text{let } x = e \text{ in } P), Q \multimap_I (\eta[x \mapsto b], P), Q} \\
\\
\frac{\llbracket e \rrbracket_\eta = \text{true}}{(\eta, \text{if } e \text{ then } P \text{ else } P'), Q \multimap_I (\eta, P), Q} \\
\\
\frac{\llbracket e \rrbracket_\eta = \text{false}}{(\eta, \text{if } e \text{ then } P \text{ else } P'), Q \multimap_I (\eta, P'), Q} \\
\\
\frac{t = \text{fixed}_n \text{ for some } n \in \mathcal{N} \quad |b| = n}{(\eta, \text{new } x : t; P), Q \xrightarrow[\frac{fr(b)}{2^n}]{\text{fr}(b)} (\eta[x \mapsto b], P), Q} \\
\\
\frac{\forall j \leq m : \llbracket d_j \rrbracket_\eta = b_j \neq \perp}{(\eta, \text{event}(d_1, \dots, d_m); P), Q \xrightarrow{\text{ev}(b_1, \dots, b_m)} (\eta, P), Q} \\
\\
\frac{\llbracket e \rrbracket_\eta = b \neq \perp \quad b' = \text{truncate}(b, \text{maxlen}(c)) \quad \forall j \leq m : \llbracket e_j \rrbracket_\eta = b_j \neq \perp \quad Q' = \text{reduce}(\{(\eta, Q)\})}{\exists! (\eta', Q') \in Q : Q' = \text{in}(c[e_1, \dots, e_m], x'); P' \wedge \forall j \leq m : \llbracket e_j \rrbracket_{\eta'} = b_j \neq \perp} \\
\\
\frac{}{(\eta, \text{out}(c[e_1, \dots, e_m], e); Q), Q \multimap_I (\eta'[x' \mapsto b'], P'), Q \sqcup Q' \setminus \{(\eta', Q')\}}
\end{array}$$

Figure 13: The semantics of *IML* [2, p. 23].

A APPENDIX

A.1 *IML* Semantics

Fig. 13 presents the *IML* semantics. The inference rules for random number generation and sending a message on the channel c are explained in Fig. 7. The construct $\text{event}(d_1, \dots, d_m)$ is used to raise an event $\text{ev}(b_1, \dots, b_m)$ during the execution where ev is an event symbol and b_1, \dots, b_m are bitstrings stored for expressions d_1, \dots, d_m in the *IML* environment η . The $\text{if } e \text{ then } P \text{ else } P'$ construct reduces to P if the expression e evaluates to *true* with respect to the *IML* environment η , otherwise reduces to P' . The construct $\text{assume } e; P$ reduces to P only in the case that the expression e evaluates to *true*

with respect to the *IML* environment η . To execute the construct $\text{let } x = e \text{ in } P$, the bitstring b stored for the expression e in the *IML* environment η is fetched. Subsequently, the value of the variable x is updated with the bitstring b in the environment η and the construct reduces to P .

A.2 Loop Summarization Example

Our symbolic execution engine handles *bounded loops* using the summarization technique of Strejček [68]. We show by an example how this method is used in our symbolic execution to summarize loops. Fig. 14 is (the simplified version of) a loop from the TinySSH's implementation, and Fig. 15 indicates its flowgraph. Our example loop consists of nodes $\{a, b, d, e\}$, with a being the entry node of the loop, and a single path $\pi_1 = abdea$. The program continues to execute from node c if a new key is received.

The loop's effect can be described by an iterated symbolic environment $\eta_{\vec{t}}$ for \vec{t} iterations of the loop. The \vec{t} is a vector of path counters, which in our example, consists of a single counter t_1 that is assigned to π_1 . The only variable which is modified within the loop is k . In our example, the value of k is independent of the number of loop iterations and the initial values of program variables, and it can be described by a symbolic value $\eta_{\vec{t}}[k] = k_{\vec{t}}$.

The looping condition describes the necessary condition to keep looping along the acyclic paths inside a given loop. For our example, the looping condition, $\phi_{\vec{t}} = \phi_1$, consists of a single formula ϕ_1 which describes the required condition to loop along the acyclic path π_1 . The formula ϕ_1 is based on two assertions, namely, $\text{!newkey}(k)$ and $k \neq 0$. Let's say that in the symbolic environment, the stored values of k in $(t'_1 - 1)$ -st and t'_1 -st iteration of the path π_1 are $k_{t'_1-1}$ and $k_{t'_1}$, respectively. Then, to get t_1 iterations of π_1 , the formula ϕ_1 says that for each t'_1 satisfying $0 < t'_1 \leq t_1$, $k_{t'_1-1}$ is not a new key, $k_{t'_1}$ is received from the channel, and it is not equal to zero:

$$\begin{aligned}
\phi_{\vec{t}} &\equiv \phi_1 \\
\phi_1 &\equiv \forall t'_1. (0 < t'_1 \leq t_1 \rightarrow \\
&\quad (\text{!newkey}(k_{t'_1-1}) \wedge k_{t'_1} = \text{receive}() \wedge k_{t'_1} \neq 0))
\end{aligned}$$

The loop summary is characterized by $(\eta_{\vec{t}}, \phi_{\vec{t}})$ which we attach it to the symbolic state at the entry node a . Afterward, we proceed to execute the program from the exit point c .

A.3 *IML*_B Semantics

Fig. 16 presents the *IML-BIR* mixed execution semantics.

A.4 Soundness of Translation into *IML*

LEMMA 7.1. For a symb. execution tree T of the BIR program P , an *IML* process I and any $k \in \mathcal{N}$ the size of the random memory, let $I\{s_0\} = (\text{True}, \eta_0[\text{RM} \mapsto \text{rm}_k, \text{rk} \mapsto 0], \text{pc}_0)$ be an initial symbolic state in *IML*_{SB} and $s_0 = (\eta_0, Q^{\text{full}})$ the corresponding initial *IML* state. Then, for all $H: I\{s_0\} \sim_{H, \langle \cdot, \cdot \rangle} s_0$.

PROOF. By construction of our symbolic tree we know that for pc_0 , we have $\langle T \rangle$. Thus, from translation rule $\langle T \rangle = Q^{\text{full}}$ in Fig. 10 we get that $s_0.P = Q^{\text{full}}$. Then, we can conclude that $(\text{True}, \eta_0[\text{RM} \mapsto \text{rm}_k, \text{rk} \mapsto 0], \text{pc}_0) \sim_{H, \langle \cdot, \cdot \rangle} (\eta_0, Q^{\text{full}})$. \square

$$\begin{array}{c}
\frac{\text{pc} \in \mathcal{L}_{\mathcal{N}} \sqcup \xi_C \sqcup \xi_{\mathcal{R}} \sqcup \xi_{\mathcal{E}} \quad (\eta, \text{pc}) \xrightarrow{a} (\eta', \text{pc}')}{\text{P} \vdash (\eta, \text{pc}), Q^{\text{xb}} \xrightarrow{a} (\eta', \text{pc}'), Q^{\text{xb}}} \text{ normal} \\
\\
\frac{\forall j \leq m : \llbracket y_j \rrbracket_{\eta} = b_j \neq \perp \quad \forall j \leq m : (\eta_{j+1}, a_j) = \text{mstore}(\eta_j, \text{heap}, \text{Mem}, b_j, 128)}{\text{P} \vdash (\eta, \text{run}(\text{pc}, (y_1, \dots, y_m))), Q^{\text{xb}} \xrightarrow{a} (\eta_{m+1}[\text{r}_0 \mapsto a_1, \dots, \text{r}_{m-1} \mapsto a_m], \text{pc}), Q^{\text{xb}}} \text{ run} \\
\\
\frac{\begin{array}{l} \llbracket e \rrbracket_{\eta} = b \neq \perp \quad b' = \text{truncate}(b, \text{maxlen}(c)) \quad \forall j \leq m : \llbracket e_j \rrbracket_{\eta} = b_j \neq \perp \quad Q^{\text{xb}'} = \text{reduce}(\{(\eta, Q)\}) \\ \exists!((\eta, \text{pc}), c[e_1', \dots, e_m']) \in Q^{\text{xb}} : \text{pc} \in \xi_{\mathcal{A}_c} \wedge \forall j \leq m : \eta[e_j'] = b_j \neq \perp \\ (\eta', a) = \text{mstore}(\eta, \text{heap}_{\mathcal{A}}, \text{Mem}_{\mathcal{A}}, b', 128) \quad \text{pc}' = \text{ret}(\eta, \text{pc}) \quad Q^{\text{xb}''} = \{((\eta, \text{pc}), c[e_1', \dots, e_m'])\} \end{array}}{\text{P} \vdash (\eta, \text{out}(c[e_1, \dots, e_m], e); Q), Q^{\text{xb}} \xrightarrow{\text{In}(b', (e_1', \dots, e_m'))} (\eta'[\text{r}_0 \mapsto a], \text{pc}'), c[e_m', \dots, e_m'], Q^{\text{xb}} \sqcup Q^{\text{xb}'} \setminus Q^{\text{xb}''}} \text{ ItoB} \\
\\
\frac{\begin{array}{l} \text{pc} \in \xi_{\mathcal{A}_s} \quad \text{pc}' = \text{ret}(\eta, \text{pc}) \quad Q^{\text{xb}'} = \{(\eta, \text{pc}')\} \quad a = \eta[\text{r}_0] \quad b = \text{mload}(\eta, a) \quad \forall j \leq m : \eta[e_j] = b_j \neq \perp \\ \exists!((\eta, Q) \in Q^{\text{xb}} : Q = \text{in}(c[e_1', \dots, e_m'], x); P \wedge \forall j \leq m : \llbracket e_j' \rrbracket_{\eta} = b_j \neq \perp \quad b' = \text{truncate}(b, \text{maxlen}(c)) \end{array}}{\text{P} \vdash ((\eta, \text{pc}), c[e_1, \dots, e_m]), Q^{\text{xb}} \xrightarrow{\text{Out}(b, (e_1, \dots, e_m))} (\eta[x \mapsto b'], P), Q^{\text{xb}} \sqcup Q^{\text{xb}'} \setminus \{(\eta, Q)\}} \text{ BtoI}
\end{array}$$

Figure 16: The mixed semantics of BIR and IML shown by IML_B.

LEMMA 7.2 (STATE/EVENT EQUIVALENCE). Let P be a BIR process and I be an IML process, then, for all $s_i, I\{s_i\}, I\{s_j\}$ and H s.t.

$I\{s_i\} \sim_{H, \langle \cdot, \cdot \rangle} s_i$ and $I\{s_i\} \xrightarrow{a^{\text{xs}}}^+ p, H I\{s_j\}$, there exist an H' and s_j s.t. $H \subseteq H', s_i \xrightarrow{a} p s_j$ and $I\{s_j\} \sim_{H', \langle \cdot, \cdot \rangle} s_j$ and if $a \neq \perp$ then $a^{\text{xs}} =_{H'} a$.

Proof of lemma 7.2 is done by a case split on the type of label sets in the BIR program P .

PROOF. We prove the statement by a case split based on the type of the program counter $I\{s_i\}.\text{pc}$.

- $\text{pc} \in \mathcal{L}_{\mathcal{E}}$

Since $I\{s_i\} \sim_{H, \langle \cdot, \cdot \rangle} s_i$, we know that $s_i.P = \langle T[s_i.\text{pc}] \rangle$. Based on the translation rules in Fig. 10, we get that $s_i.P = \text{event}(d_1, \dots, d_m); \langle T' \rangle$. Also, by construction of the symbolic execution tree we know that there exist T_0 and T_1 such that $T = T_0 :: (\text{pc}, \text{Ev}(d_1, \dots, d_m)) :: (\text{pc}', \text{ev}) :: T_1$. Therefore, based on the operational semantics of IML processes [2, p. 23] and translation rules in Fig. 10, we get $s_i \xrightarrow{\text{ev}(b_1, \dots, b_m)} s_j$ and $s_j = (\eta, \langle T[s_j.\text{pc}'] \rangle), Q$.

Based on the semantics of IML_{SB}, Fig. 12, we find $I\{s_j\} = (\phi, \eta, \text{pc}'), Q^{\text{xs}}$ that is in the relation $I\{s_i\} \xrightarrow{a^{\text{xs}}}^+ I, H I\{s_j\}$ such that $a^{\text{xs}} = \text{Ev}(d_1, \dots, d_m)$.

Finally, we choose H' by extending H according to the executed operation and such that $H'(d_j) = b_j$ for $0 < j \leq m$. Therefore, we can conclude that $(\phi, \eta, \text{pc}'), Q^{\text{xs}} \sim_{H', \langle \cdot, \cdot \rangle} (\eta, \langle T[s_i.\text{pc}'] \rangle), Q$ and $\text{Ev}(d_1, \dots, d_m) =_{H'} \text{ev}(b_1, \dots, b_m)$.

- $\text{pc} \in \mathcal{L}_{\mathcal{C}}$

Since $I\{s_i\} \sim_{H, \langle \cdot, \cdot \rangle} s_i$, we know that $s_i.P = \langle T[s_i.\text{pc}] \rangle$. Based on the translation rules in Fig. 10, we get that $s_i.P = \text{let } x = v \text{ in } \langle T' \rangle$. Also, by construction of the symbolic execution tree we know that there exist T_0 and T_1 such that $T = T_0 :: (\text{pc}, \text{Cr}(v)) :: (\text{pc}', \text{ev}) :: T_1$.

Therefore, based on the operational semantics of IML processes [2, p. 23] and translation rules in Fig. 10, we get that $s_i \xrightarrow{a^{\text{xs}}} s_j$ and $s_j = (\eta[x \mapsto b], \langle T[s_j.\text{pc}'] \rangle), Q$. Based on the semantics of IML_{SB}, Fig. 12, we find $I\{s_j\} = (\phi, \eta'', \text{pc}'), Q^{\text{xs}}$ that is in the transition relation $I\{s_i\} \xrightarrow{a^{\text{xs}}}^+ I, H I\{s_j\}$ such that $a^{\text{xs}} = \text{Cr}(v)$ and $\eta'' = \eta'[\text{r}_0 \mapsto a]$ where $(\eta', a) = \text{mstore}(\eta, \text{heap}_{\text{Op}}, \text{Mem}_{\text{Op}}, v, 128)$.

Finally, we choose H' by extending H according to the executed operation and such that $H'(v) = b$. Therefore, we can conclude that $(\phi, \eta'[\text{r}_0 \mapsto a], \text{pc}'), Q^{\text{xs}} \sim_{H', \langle \cdot, \cdot \rangle} (\eta[x \mapsto b], \langle T[s_i.\text{pc}'] \rangle), Q$.

- $\text{pc} \in \mathcal{L}_{\mathcal{R}}$

Since $I\{s_i\} \sim_{H, \langle \cdot, \cdot \rangle} s_i$, we know that $s_i.P = \langle T[s_i.\text{pc}] \rangle$. Based on the translation rules in Fig. 10, we get that $s_i.P = \text{new } x; \langle T' \rangle$. Also, by construction of the symbolic execution tree we know that there exist T_0 and T_1 such that $T = T_0 :: (\text{pc}, \text{Fr}(x)) :: (\text{pc}', \text{ev}) :: T_1$.

Therefore, based on the operational semantics of IML processes [2, p. 23] and translation rules in Fig. 10, we get that $s_i \xrightarrow{\text{fr}(b)} s_j$ and $s_j = (\eta[x \mapsto b], \langle T[s_j.\text{pc}'] \rangle), Q$. Based on the semantics of IML_{SB}, Fig. 12, we find $I\{s_j\} = (\phi, \eta'', \text{pc}'), Q^{\text{xs}}$ that is in the relation $I\{s_i\} \xrightarrow{a^{\text{xs}}}^+ I, H I\{s_j\}$ such that $a^{\text{xs}} = \text{Fr}(x)$ and $\eta'' = \eta'[\text{r}_0 \mapsto a; \text{r}_k \mapsto \eta[\text{r}_k] + l]$ where $(\eta', a) = \text{mstore}(\eta, \text{heap}, \text{Mem}, x, 128)$ and $x = \mathcal{R}(\eta, n)$.

Finally, we choose H' by extending H according to the executed operation and such that $H'(x) = b$. Therefore, we can conclude that $(\phi, \eta'[\text{r}_0 \mapsto a; \text{r}_k \mapsto \eta[\text{r}_k] + l], \text{pc}'), Q^{\text{xs}} \sim_{H', \langle \cdot, \cdot \rangle} (\eta[x \mapsto b], \langle T[s_i.\text{pc}'] \rangle), Q$ and $\text{fr}(b) =_{H'} \text{Fr}(x)$.

- $pc_1 \in L_{\mathcal{A}_s} \wedge pc_2 \in L_{\mathcal{A}_r}$

In this case, since $pc_1 \in L_{\mathcal{A}_s}$, we have an *IML* output process *out* in the *IML* state s_i . Based on IOut rule in the operational semantics of *IML* processes [2, p. 23], we get that there is an *IML* input process *in* in the multiset of executing input processes that is ready to receive input from the channel. Therefore, there exists a node in our symbolic execution tree such that its program counter is in the label set $L_{\mathcal{A}_r}$.

More formally, since $I\{s_i\} \sim_{H, \emptyset, \emptyset} s_i$, we know that $s_i.P = \langle T[s_i.pc] \rangle$. Based on the translation rules in Fig. 10, we get that $s_i.P = \text{out}(c[e_1, \dots, e_m], e); \langle T' \rangle$. Based on the operational semantics of *IML* processes [2, p. 23], we get that $s_i \xrightarrow{1s_j}$ and $s_j = (\eta' [v \mapsto b'], P'), Q \uplus Q' \setminus Q''$ such that $Q'' = \{(\eta', \text{in}(c[e_1', \dots, e_m'], v); P')\}$.

Therefore, there exist T_0 and T_1 such that $T' = T_0 :: (pc_2, \text{in}(v, (e_1', \dots, e_m')) :: (pc', \text{ev}) :: T_1$ and it holds that $\text{in}(c[e_1', \dots, e_m'], v); P' = \langle \text{in}(v, (e_1', \dots, e_m')) \rangle; \langle \text{ev} \rangle$.

Since we have an event node $(pc_2, \text{in}(v, (e_1', \dots, e_m')))$, we get that there exist an intermediate state $I\{s_i\} \in Q^{i \times s}$ such that $I\{s_i\} = \{((\phi, \eta', pc_2), c[e_1', \dots, e_m'])\}$ and $pc_2 \in L_{\mathcal{A}_r}$. Based on the semantics of *IML_{SB}*, Fig. 12, we find $I\{s_j\} = ((\phi, \eta''', pc'), c[e_1', \dots, e_m'])$, $Q^{i \times s} \uplus Q^{i \times s'} \setminus Q^{i \times s''}$ that is in

the transition relation $I\{s_i\} \xrightarrow{a^{i \times s}}_{1, H} I\{s_j\}$ such that $a^{i \times s} = \text{Out}(e, (e_1, \dots, e_m)); \dots; \text{In}(v, (e_1', \dots, e_m'))$ and $\eta''' = \eta'' [r_0 \mapsto a]$ where $(\eta'', a) = \text{mstore}(\eta', \text{heap}_{\mathcal{A}}, \text{Mem}_{\mathcal{A}}, v, 128)$ and $pc' = \text{ret}(\phi, \eta', pc_2)$.

Finally, we choose H' by extending H according to the executed operation and such that $H'(v) = b'$. Thus, we can conclude that $((\phi, \eta'' [r_0 \mapsto a], pc'), c[e_1', \dots, e_m']), Q^{i \times s} \uplus Q^{i \times s'} \setminus Q^{i \times s''} \sim_{H', \emptyset, \emptyset} (\eta' [v \mapsto b'], \langle T[s_i.pc] \rangle), Q \uplus Q' \setminus Q''$.

- $pc \in L_N \wedge pc \notin L_L$

We show this case by a case split based on $P(I\{s_i\}.pc)$.

- $P(I\{s_i\}.pc) \neq \text{Branch}(\gamma, T_1, T_2)$

Since $I\{s_i\} \sim_{H, \emptyset, \emptyset} s_i$, we know that $s_i.P = \langle T[s_i.pc] \rangle$. By construction of the symbolic execution tree we know that there exist T_0 and T_1' such that $T = T_0 :: (pc, \tau) :: (pc', \text{ev}) :: T_1'$.

Therefore, based on the operational semantics of *IML* processes [2, p. 23] and translation rules in Fig. 10, we get that $s_j = (\eta', \langle T[s_j.pc] \rangle), Q$ and $s_i \xrightarrow{1s_j}$. Based on the semantics of *IML_{SB}*, Fig. 12, we find $I\{s_j\} = (\phi', \eta', pc'), Q^{i \times s}$ that is in the relation $I\{s_i\} \xrightarrow{1, H} I\{s_j\}$.

Finally, we choose H' by extending H according to the executed operation and such that $H'(I\{s_j\}. \eta) = s_j. \eta$. Therefore, we can conclude that $(\phi', \eta', pc'), Q^{i \times s} \sim_{H', \emptyset, \emptyset} (\eta', \langle T[s_i.pc] \rangle), Q$.

- $P(I\{s_i\}.pc) = \text{Branch}(\gamma, T_1, T_2)$ case *True* (case *False*)

Since $I\{s_i\} \sim_{H, \emptyset, \emptyset} s_i$, we know that $s_i.P = \langle T[s_i.pc] \rangle$. Based on the translation rules in Fig. 10, we get that $s_i.P = \text{if } (\gamma) \text{ then } \langle T_1 \rangle \text{ else } \langle T_2 \rangle$. Also, by construction of the symbolic execution tree we know that there exist T_0, T_1' , and T_2' s.t. $T = T_0 :: (pc, \text{Branch}(\gamma, ((pc', -) :: T_1'), T_2'))$ ($T = T_0 :: (pc, \text{Branch}(\gamma, T_1, ((pc'', -) :: T_2'))$)).

Therefore, based on the operational semantics of *IML* processes [2, p. 23] and translation rules in Fig. 10, we get that $s_j = (\eta, \langle T[s_j.pc'] \rangle), Q$ ($s_j = (\eta, \langle T[s_j.pc''] \rangle), Q$) and $s_i \xrightarrow{1s_j}$.

Based on the semantics of *IML_{SB}*, Fig. 12, we find $s_j = (\phi, \eta, pc')$ ($s_j = (\phi, \eta, pc'')$) that is in the transition relation $I\{s_i\} \xrightarrow{1, H} I\{s_j\}$. Therefore, we can conclude that $(\phi, \eta, pc'), Q^{i \times s} \sim_{H', \emptyset, \emptyset} (\eta, \langle T[s_i.pc'] \rangle), Q$ ($(\phi, \eta, pc''), Q^{i \times s} \sim_{H', \emptyset, \emptyset} (\eta, \langle T[s_i.pc''] \rangle), Q$).

- $P(I\{s_i\}.pc) \in L_L$

Since $I\{s_i\} \sim_{H, \emptyset, \emptyset} s_i$, we know that $s_i.P = \langle T[s_i.pc] \rangle$. Based on the translation rules in Fig. 10, we get that $s_i.P = !^{t \leq m} \text{LoopProc}(s_i.pc)$. Also, by construction of the symbolic execution tree we know that there exist T_0 and T_1 such that $T = T_0 :: (pc, \text{loop}(t)) :: (pc', \text{ev}) :: T_1$ such that $pc' = \text{exit}(pc)$. Therefore, based on the operational semantics of *IML* processes [2, p. 23] and translation rules in Fig. 10, we get $s_j = (\eta', \langle T[s_j.pc'] \rangle), Q$ and $s_i \xrightarrow{1s_j}$ such that $\eta'[t] = b$. Based on the semantics of *IML_{SB}*, Fig. 12 and the computed loop summary, we find $I\{s_j\} = (\phi', \eta', pc'), Q^{i \times s}$ that is in the transition relation $I\{s_i\} \xrightarrow{a^{i \times s}}_{1, H} I\{s_j\}$, $a^{i \times s} = \text{loop}(t)$, and $pc' = \text{exit}(pc)$.

Finally, for every *IML* variable x and symbolic BIR variable x that $I\{s_j\}. \eta[x] \neq I\{s_i\}. \eta[x]$ and $s_j. \eta[x] \neq s_i. \eta[x]$, we choose H' by extending H according to the executed operation such that $H'(I\{s_j\}. \eta[x]) = s_j. \eta[x]$. Moreover, for the symbolic counter t which represents the number of iterations of the loop in symbolic execution and the number of the process replication b , we have $H'(t) = b$. Therefore, we can conclude that $(\phi', \eta', pc'), Q^{i \times s} \sim_{H', \emptyset, \emptyset} (\eta', \langle T[s_j.pc'] \rangle), Q$.

□

A.5 Soundness of Symbolic Execution

LEMMA 7.4. For a BIR program P , an *IML* process I and any upper bound $k \in \mathcal{N}$ on the number of RNG steps, let $I\{s_0\} = (\eta_0[\text{RM} \mapsto \text{rm}_k, \text{r}_k \mapsto 0], \text{pc}_0)$ be an initial BIR state in *IML_B* and $I\{s_0\} = (\text{True}, \eta_0[\text{RM} \mapsto \text{rm}_k, \text{r}_k \mapsto 0], \text{pc}_0)$ be the corresponding initial state in *IML_{SB}*. Then, $I\{s_0\} \sim_H I\{s_0\}$ for all H .

PROOF. We choose H such that for $0 < i \leq k$, $H(I\{s_0\}. \eta. \text{RM}[x_i]) = I\{s_0\}. \eta. \text{RM}[x_i]$. Finally, we can conclude that $(\eta_0[\text{RM} \mapsto \text{rm}_k, \text{r}_k \mapsto 0], \text{pc}_0) \sim_H (\text{True}, \eta_0[\text{RM} \mapsto \text{rm}_k, \text{r}_k \mapsto 0], \text{pc}_0)$. □

LEMMA 7.5 (STATE/EVENT EQUIVALENCE). Let P be a BIR program and I be an *IML* process, then, for all $I\{s_i\}$, $I\{s_i\}$, $I\{s_j\}$ and H s.t.

$I\{s_i\} \sim_H I\{s_i\}$ and $I\{s_i\} \xrightarrow{a^{i \times b}}_{p, H} I\{s_j\}$, there exist an H' and $I\{s_j\}$ s.t. $H \subseteq H'$, $I\{s_i\} \xrightarrow{a^{i \times s}}_{p, H'} I\{s_j\}$, $I\{s_j\} \sim_{H'} I\{s_j\}$ and $a^{i \times b} =_{H'} a^{i \times s}$.

Proof of lemma 7.5 is done by a case split on the type of label sets in the BIR program P .

PROOF. We prove the statement by a case split based on the type of the program counter $I\{s_i\}.pc$.

- $pc \in L_E$

Based on the mixed semantics of *IML* and BIR, Fig. 16, we

get that $I\{s_j\} = (\eta', \text{pc}')$, Q^{xb} and $I\{s_i\} \xrightarrow{a^{\text{xb}}} I\{s_j\}$. Based on the definition of event functions in Sec. 4, we have $\text{pc}' = \text{ret}(\eta, \text{pc})$, $a^{\text{xb}} = \text{Ev}(b_1, \dots, b_m)$ and $\eta' = \eta$. Since $I\{s_i\} \sim_H I\{s_j\}$, we get that $I\{s_i\}.\text{pc} \in \mathcal{L}_E$. Based on the semantics of IML_{SB} , Fig. 12, we find $I\{s_j\} = (\phi, \eta, \text{pc}')$, Q^{xs} that is in the transition relation $I\{s_i\} \xrightarrow{a^{\text{xs}}} I\{s_j\}$ such that $a^{\text{xs}} = \text{Ev}(d_1, \dots, d_m)$.

Therefore, we choose H' by extending H according to the executed operation and such that $H'(d_j) = b_j$ for $0 < j \leq m$. Moreover, since the environment of both s_j and s_i are the same as s_i and s_i , respectively, we get that $I\{s_j\}.\eta =_{H'} I\{s_i\}.\eta$. Finally, we can conclude that $(\eta, \text{pc}'), Q^{\text{xb}} \sim_{H'} (\phi, \eta, \text{pc}'), Q^{\text{xs}}$ and $\text{Ev}(b_1, \dots, b_m) =_{H'} \text{Ev}(d_1, \dots, d_m)$.

- $\text{pc} \in \mathcal{L}_C$

Based on the mixed semantics of IML and BIR , Fig. 16, we get that $I\{s_j\} = (\eta'', \text{pc}')$, Q^{xb} and $I\{s_i\} \xrightarrow{a^{\text{xb}}} I\{s_j\}$. Based on the definition of crypto calls in Sec. 4, we have $a^{\text{xb}} = \text{Cr}(v)$, $\text{pc}' = \text{ret}(\eta, \text{pc})$ and $\eta'' = \eta'[\text{r}_0 \mapsto a]$ where $(\eta', a) = \text{mstore}(\eta, \text{heap}_{\text{Op}}, \text{Mem}_{\text{Op}}, v, 128)$.

Since $I\{s_i\} \sim_H I\{s_j\}$, we get that $I\{s_i\}.\text{pc} \in \mathcal{L}_C$. Based on the semantics of IML_{SB} , Fig. 12, we find $I\{s_j\} = (\phi, \eta'', \text{pc}')$, Q^{xs} that is in the transition relation $I\{s_i\} \xrightarrow{a^{\text{xs}}} I\{s_j\}$ such that $a^{\text{xs}} = \text{Cr}(v)$ and $\eta'' = \eta'[\text{r}_0 \mapsto a]$ where $(\eta', a) = \text{mstore}(\eta, \text{heap}_{\text{Op}}, \text{Mem}_{\text{Op}}, v, 128)$.

Therefore, we choose H' by extending H according to the executed operation and such that $H'(v) = v$. Hence, we get that $I\{s_j\}.\eta =_{H'} I\{s_i\}.\eta$. Finally, we can conclude that $(\eta'', \text{pc}'), Q^{\text{xb}} \sim_{H'} (\phi, \eta'', \text{pc}'), Q^{\text{xs}}$ and $\text{Cr}(v) =_{H'} \text{Cr}(v)$.

- $\text{pc} \in \mathcal{L}_R$

Based on the mixed semantics of IML and BIR , Fig. 16, we get that $I\{s_j\} = (\eta'', \text{pc}')$, Q^{xb} and $I\{s_i\} \xrightarrow{a^{\text{xb}}} I\{s_j\}$. Based on the definition of RNG in Sec. 4, we have $a^{\text{xb}} = \text{Fr}(x_d)$, $d = \left\lfloor \frac{\eta[\text{r}_k]}{l} \right\rfloor + 1$, $\eta'' = \eta'[\text{r}_0 \mapsto a; \text{r}_k \mapsto \eta[\text{r}_k] + l]$ where $(\eta', a) = \text{mstore}(\eta, \text{heap}, \text{Mem}, x_d, 128)$, $x_d = \mathfrak{R}(\eta, n)$ and $\text{pc}' = \text{ret}(\eta, \text{pc})$.

Since $I\{s_i\} \sim_H I\{s_j\}$, we get that $I\{s_i\}.\text{pc} \in \mathcal{L}_R$. Based on the semantics of IML_{SB} , Fig. 12, we find $I\{s_j\} = (\phi, \eta'', \text{pc}')$, Q^{xs} that is in the relation $I\{s_i\} \xrightarrow{a^{\text{xs}}} I\{s_j\}$ such that $a^{\text{xs}} = \text{Fr}(x_{d'})$, $d' = \left\lfloor \frac{\eta[\text{r}_k]}{l} \right\rfloor + 1$ and $\eta'' = \eta'[\text{r}_0 \mapsto a; \text{r}_k \mapsto \eta[\text{r}_k] + l]$ where $(\eta', a) = \text{mstore}(\eta, \text{heap}, \text{Mem}, x_{d'}, 128)$ and $x_{d'} = \mathfrak{R}(\eta, n)$.

Since $I\{s_i\} \sim_H I\{s_j\}$ and $I\{s_i\}.\eta =_H I\{s_j\}.\eta$, we get that $d' = d$. Finally, we choose H' by extending H according to the executed operation and such that $H'(x_d) = x_d$. Hence, we get that $I\{s_j\}.\eta =_{H'} I\{s_i\}.\eta$. Therefore, we can conclude that $(\eta'', \text{pc}'), Q^{\text{xb}} \sim_{H'} (\phi, \eta'', \text{pc}'), Q^{\text{xs}}$ and $\text{Fr}(x_d) =_{H'} \text{Fr}(x_{d'})$.

- $\text{pc}_1 \in \mathcal{L}_{\mathcal{A}_s} \wedge \text{pc}_2 \in \mathcal{L}_{\mathcal{A}_r}$

In this case, $I\{s_i\} \xrightarrow{a^{\text{xb}}} I\{s_j\}$ amounts to 3 sub-transitions

as follows. First, using the $BtoI$ rule in Fig. 16, we have the transition relation $I\{s_i\} \xrightarrow{\text{Out}(b, (e_1, \dots, e_m))} I\{s_x\}$ such that $I\{s_i\}.\text{pc}_1 \in \mathcal{L}_{\mathcal{A}_s}$.

Then, based on the operational semantics of IML output processes [2, p. 23], the transition $I\{s_x\} \xrightarrow{p} I\{s_y\}$ takes place.

Finally, we have $I\{s_y\} \xrightarrow{\text{In}(b', (e'_1, \dots, e'_m))} I\{s_j\}$ using $ItoB$ rule in Fig. 16 such that there exist a BIR state $((\eta, \text{pc}_2), c[e'_1, \dots, e'_m])$ in the multiset of executing states $I\{s_y\}.Q^{\text{xb}}$ such that $\text{pc}_2 \in \mathcal{L}_{\mathcal{A}_r}$.

Since $I\{s_i\} \sim_H I\{s_j\}$, we need to find a mixed symbolic and IML state $I\{s_j\}$ such that $I\{s_j\} \sim_H I\{s_i\}$ and it is reachable from intermediate states $I\{s_x\}$ and $I\{s_y\}$ using $SBtoI$ and $ItoSB$ rules in Fig. 12, respectively.

Therefore, since $\text{pc}_1 \in \mathcal{L}_{\mathcal{A}_s}$, based on the mixed semantics of IML and BIR , Fig. 16, we get that there exist an IML state

$I\{s_x\}$ such that $I\{s_i\} \xrightarrow{\text{Out}(b, (e_1, \dots, e_m))} I\{s_x\}$ and $I\{s_x\} = (\eta[x \mapsto b'], P), Q^{\text{xb}} \setminus Q^{\text{xb}'}$ such that $b' = \text{truncate}(b, \text{maxlen}(c))$.

Since $I\{s_i\} \sim_H I\{s_j\}$, we get that $I\{s_i\}.\text{pc}_1 \in \mathcal{L}_{\mathcal{A}_s}$. Based on the semantics of IML_{SB} , Fig. 12, we find $I\{s_x\} = (\eta[x \mapsto b'], P), Q^{\text{xs}} \setminus Q^{\text{xs}'}$ that is in the transition relation $I\{s_i\} \xrightarrow{a^{\text{xs}}} I\{s_x\}$ such that $a^{\text{xs}} = \text{Out}(e, (e_1, \dots, e_m))$.

Since $\text{pc}_2 \in \mathcal{L}_{\mathcal{A}_r}$, based on the mixed semantics of IML and BIR , Fig. 16, we get that there exist an IML state s_y such

that $I\{s_y\} \xrightarrow{\text{In}(b', (e'_1, \dots, e'_m))} I\{s_j\}$ and $I\{s_j\} = ((\eta'[\text{r}_0 \mapsto a], \text{pc}'), c[e'_1, \dots, e'_m]), Q^{\text{xb}} \setminus Q^{\text{xb}'} \setminus Q^{\text{xb}''}$ such that $(\eta', a) = \text{mstore}(\eta, \text{heap}_{\mathcal{A}}, \text{Mem}_{\mathcal{A}}, b', 128)$ and $\text{pc}' = \text{ret}(\eta, \text{pc}_2)$.

Since $I\{s_i\} \sim_H I\{s_j\}$, we get that $I\{s_i\}.\text{pc}_2 \in \mathcal{L}_{\mathcal{A}_r}$. Based on the semantics of IML_{SB} , Fig. 12, we find $I\{s_j\} = ((\phi, \eta'[\text{r}_0 \mapsto a], \text{pc}'), c[e'_1, \dots, e'_m]), Q^{\text{xs}} \setminus Q^{\text{xs}'} \setminus Q^{\text{xs}''}$ that is in the relation $I\{s_y\} \xrightarrow{a^{\text{xs}}} I\{s_j\}$ such that $\text{pc}' = \text{ret}(\phi, \eta, \text{pc}_2)$, $a^{\text{xs}} = \text{In}(e', (e'_1, \dots, e'_m))$ and $(\eta', a) = \text{mstore}(\eta, \text{heap}_{\mathcal{A}}, \text{Mem}_{\mathcal{A}}, e', 128)$.

Moreover, we choose H' by extending H according to the executed operation and such that $H'(e) = b$, $H'(e') = b'$, $H'(e_j) = e_j$ and $H'(e'_j) = e'_j$ for $1 \leq j \leq m$. Hence, we get that $I\{s_j\}.\eta =_{H'} I\{s_i\}.\eta$. Therefore, we can conclude that $((\eta'[\text{r}_0 \mapsto a], \text{pc}'), c[e'_1, \dots, e'_m]), Q^{\text{xb}} \setminus Q^{\text{xb}'} \setminus Q^{\text{xb}''} \sim_{H'} ((\phi, \eta'[\text{r}_0 \mapsto a], \text{pc}'), c[e'_1, \dots, e'_m]), Q^{\text{xs}} \setminus Q^{\text{xs}'} \setminus Q^{\text{xs}''}$, $\text{Out}(b, (e_1, \dots, e_m)) =_{H'} \text{Out}(e, (e_1, \dots, e_m))$, and $\text{In}(b', (e'_1, \dots, e'_m)) =_{H'} \text{In}(e', (e'_1, \dots, e'_m))$.

- $\text{pc} \in \mathcal{L}_{\mathcal{N}} \wedge \text{pc} \notin \mathcal{L}_{\mathcal{L}}$

Based on the mixed semantics of IML and BIR , Fig. 16, we

get that $I\{s_j\} = (\eta', \text{pc}')$, Q^{xb} and $I\{s_i\} \xrightarrow{a^{\text{xb}}} I\{s_j\}$. Since $I\{s_i\} \sim_H I\{s_j\}$, we get that $I\{s_i\}.\text{pc} \in \mathcal{L}_{\mathcal{N}}$. Based on the semantics of IML_{SB} , Fig. 12, we find $I\{s_j\} = (\phi', \eta', \text{pc}')$, Q^{xs} that is in the transition relation $I\{s_i\} \xrightarrow{a^{\text{xs}}} I\{s_j\}$.

Based on Lindner's work [53] (see Property 1), there exist an interpretation $H' \supseteq H$ such that $I\{s_j\}.\eta =_{H'} I\{s_i\}.\eta$. Therefore, we can conclude that $(\eta', \text{pc}'), Q^{\text{xb}} \sim_{H'} (\phi', \eta', \text{pc}')$,

$Q^{I \times S}$.

- $\text{pc} \in L_{\mathcal{L}}$

Based on the mixed semantics of *IML* and *BIR*, Fig. 16, we get that $I\{s_j\} = (\eta', \text{pc}')$, $Q^{I \times b}$ and $I\{s_i\} \xrightarrow{\text{loop}(t)}^+ I\{s_j\}$ and $\text{pc}' = \text{exit}(\text{pc})$. Since $I\{s_i\} \sim_H I\{s_j\}$, we get that $I\{s_i\}.\text{pc} \in L_{\mathcal{L}}$. Based on the semantics of *IML*_{SB}, Fig. 12 and the computed loop summary⁴, we find $I\{s_j\} = (\phi', \eta', \text{pc}')$, $Q^{I \times S}$ that is in the transition relation $I\{s_i\} \xrightarrow{\text{loop}(t)}^+ I\{s_j\}$ such that $\text{pc}' = \text{exit}(\text{pc})$. Therefore, we get that both loops in $I\{s_j\}$ and $I\{s_i\}$ terminate at the same position pc' .

Finally, for every *BIR* variable x and symbolic *BIR* variable x that $I\{s_j\}.\eta[x] \neq I\{s_i\}.\eta[x]$ and $I\{s_i\}.\eta[x] \neq I\{s_j\}.\eta[x]$, we choose H' by extending H according to the executed operation such that $H'(I\{s_j\}.\eta[x]) = I\{s_i\}.\eta[x]$. Moreover, for the symbolic counter t which represents the number of iterations of the loop in symbolic execution and the concrete counter t , we have $H'(t) = t$. Therefore, we can conclude that $(\eta', \text{pc}'), Q^{I \times b} \sim_{H'} (\phi', \eta', \text{pc}'), Q^{I \times S}$ and $\text{loop}(t) =_{H'} \text{loop}(t)$.

□

A.6 Security properties

Preliminaries. Trace property ψ is a polynomially decidable prefix-closed set of event traces.

$$\{t \in \psi \mid \forall i \in \mathcal{N} : t[i] \in \psi\} \implies \{(\exists j \in \mathcal{N} : j < i \wedge t[..j] \in \psi)\}$$

Here, we use the following notation. Given a trace $t = a_0 a_1 \dots$ and index i , we define $t[i] = a_i$ and $t[..i] = a_0 a_1 \dots a_i$.

THEOREM 8.3 (TRANSLATION PRESERVES ATTACKS). *Given a *BIR* program P , an *IML* process I , a security parameter $n \in \mathcal{N}$, a trace property ψ and an upper bound $k \in \mathcal{N}$ on the number of RNG steps in $T^{I \times b}(I\{P\}, \eta_0[\text{RM} \mapsto \text{rm}_k, \mathbf{r}_k \mapsto 0])$, we get that*

$$\text{insec}(I\{P\}, n, k, \psi) \leq \text{insec}(I\{\langle P \rangle\}, n, \psi).$$

PROOF. Throughout the following proof, we use the following property of sums: if $\zeta : A \rightarrow B$ is an injection, then $\sum_{a \in A} f(\zeta(a)) \leq \sum_{b \in B} f(b)$. Recall also that, if $\zeta : A \rightarrow B$ is injective, then $\zeta|_{A'} = \text{inj}$ for any A' . Also, for brevity, let $T^{I \times S} = T^{I \times S}(I\{P\}, \eta_0[\text{RM} \mapsto \text{rm}_k, \mathbf{r}_k \mapsto 0])$ and $T^{I \times b} = T^{I \times b}(I\{P\}, \eta_0[\text{RM} \mapsto \text{rm}_k, \mathbf{r}_k \mapsto 0])$ in the follow up.

$$\begin{aligned} \text{insec}(I\{\langle P \rangle\}, n, \psi) \\ \stackrel{\text{Def. 8.2}}{=} \sum_{t \in T(I\{\langle P \rangle\}, n) \cap \psi_{\neg}} \text{pr}(t) \end{aligned}$$

⁴Note that the concrete *BIR* uses unrolling for loops, while our symbolic execution, *SBIR*, uses the loop summary to handle bounded loops.

Note that $R^{I \times S} \sim_{H, k, \langle \cdot \rangle} R$ implies $H(t^{I \times S}) \in \psi_{\neg} \Leftrightarrow t \in \psi_{\neg}$

$$\begin{aligned} \stackrel{\text{Lem. A.4, Lem. A.2}}{\geq} \sum_{\substack{t^{I \times S} \in T^{I \times S} \\ H:\text{rm}_k \rightarrow BS_n^k \\ \text{s.t. } \exists H'. H'|_{\text{rm}_k} = H \\ \wedge H'(t^{I \times S}) \in \psi_{\neg} \\ \wedge \text{rng}(t^{I \times S}) = l}} 2^{-nl} \text{pr}(t^{I \times S}) \end{aligned}$$

Let k be the maximal l in the sum. We extend the range of H , which increases the range of the sum by $2^{n(k-l)}$:

$$\begin{aligned} &= \sum_{\substack{t^{I \times S} \in T^{I \times S} \\ H:\text{rm}_k \rightarrow BS_n^k \\ \text{s.t. } \exists H'. H'|_{\text{rm}_k} = H \\ \wedge H'(t^{I \times S}) \in \psi_{\neg} \\ \wedge \text{rng}(t^{I \times S}) = l}} \frac{2^{-nl}}{2^{n(k-l)}} \cdot \text{pr}(t^{I \times S}) \\ &\stackrel{\text{Lem. A.3}}{=} \sum_{\substack{t^{I \times S} \in T^{I \times S} \\ H:\text{rm}_k \rightarrow BS_n^k \\ \text{s.t. } \exists H'. H'|_{\text{rm}_k} = H \\ \wedge H'(t^{I \times S}) \in \psi_{\neg}}} 2^{-nk} \text{pr}(t^{I \times S}) \\ &= 2^{-nk} \cdot \sum_{\substack{t^{I \times S} \in T^{I \times S} \\ H:\text{rm}_k \rightarrow BS_n^k \\ \text{s.t. } \exists H'. H'|_{\text{rm}_k} = H \\ \wedge H'(t^{I \times S}) \in \psi_{\neg}}} \text{pr}(t^{I \times S}) \end{aligned}$$

Note that $R^{I \times b} \sim_{H, k} R^{I \times S}$ implies $H(t^{I \times S}) \in \psi_{\neg} \Leftrightarrow t^{I \times b} \in \psi_{\neg}$

$$\begin{aligned} \stackrel{\text{Lem. A.5}}{\geq} 2^{-nk} \cdot \sum_{\substack{\text{rm}_k \in BS_n^k \\ t^{I \times b} \in T^{I \times b} \cap \psi_{\neg}}} \text{pr}(t^{I \times b}) \\ \stackrel{\text{Def. 8.1}}{=} \text{insec}(I\{P\}, n, k, \psi) \end{aligned}$$

□

A.6.1 Lemmas required to prove Thm. 8.3.

LEMMA A.1 (TRACE CONTAINS RANDOMNESS). *For a *BIR* program P , an *IML* process I , any upper bound $k \in \mathcal{N}$ on the number of RNG steps, all *IML* traces $R \in \mathcal{R}^I(I\{\langle P \rangle\})$, all interpretations H and H' , and all mixed *IML* and symbolic execution traces $R^{I \times S} \in \mathcal{R}^{I \times S}(I\{P\}, \eta_0[\text{RM} \mapsto \text{rm}_k, \mathbf{r}_k \mapsto 0])$ with d the number of RNG steps in $R^{I \times S}$ such that rm_d be the first d random symbolic values in RM and $d \leq k$, then,*

$$R^{I \times S} \sim_{H, k, \langle \cdot \rangle} R \wedge R^{I \times S} \sim_{H', k, \langle \cdot \rangle} R \implies H|_{\text{rm}_d} = H'|_{\text{rm}_d}$$

PROOF. Since the relation $R^{I \times S} \sim_{H, k, \langle \cdot \rangle} R$ holds, we know that

$$\begin{aligned} R^{I \times S} &= I\{s_0\} \xrightarrow{a_1^{I \times S}} p_{1,H} \dots \xrightarrow{a_i^{I \times S}} p_{i,H} I\{s_i\} \xrightarrow{\text{Fr}(x_1)} I\{s_{i+1}\} \\ &\xrightarrow{a_{i+2}^{I \times S}} p_{i+2,H} \dots \xrightarrow{a_j^{I \times S}} p_{j,H} I\{s_j\} \xrightarrow{\text{Fr}(x_d)} I\{s_{j+1}\} \xrightarrow{a_{j+2}^{I \times S}} p_{j+2,H} \dots \\ &\xrightarrow{a_m^{I \times S}} p_m, H I\{s_m\} \text{ where } a_i^{I \times S} \text{ does not come from a RNG}(n) \text{ call and} \\ &x_1, \dots, x_d = \text{rm}_d \text{ and } R = s_0 \xrightarrow{a_1} p_1 \dots \xrightarrow{a_i} p_i s_i \xrightarrow{\text{fr}(b_i)} \frac{1}{2^n} s_{i+1} \dots \\ &\xrightarrow{a_j} p_j s_j \xrightarrow{\text{fr}(b_d)} \frac{1}{2^n} s_{j+1} \dots \xrightarrow{a_m} p_m s_m. \text{ For all } 0 < i \leq d, \text{ then, } b_i = \\ &H(x_i) \text{ (1). Since } R^{I \times S} \sim_{H', k, \langle \cdot \rangle} R, \text{ for the random symbolic values} \\ &x_1, \dots, x_d, \text{ we get that } b_i = H'(x_i) \text{ for } 0 < i \leq d \text{ (2). From (1) and} \\ &\text{(2), we conclude that } H|_{\text{rm}_d} = H'|_{\text{rm}_d}. \end{aligned}$$

□

LEMMA A.2. For a BIR program P , an IML process I , any upper bound $k \in \mathcal{N}$ on the number of RNG steps, a security parameter $n \in \mathcal{N}$, all IML traces $R \in \mathcal{R}^I(I(\{P\}))$, all interpretations H , all mixed IML and symbolic execution traces $R^{t^{\times s}} \in \mathcal{R}^{t^{\times s}}(I\{P\}, \eta_0[\text{RM} \mapsto \text{rm}_k, \text{rk} \mapsto 0])$, if $R^{t^{\times s}} \sim_{H,k,\emptyset} R$, then, the number of INEW steps in R and the number of RNG plus the number of INEW steps in $R^{t^{\times s}}$ are equal. Moreover, $\text{pr}(\text{tr}(R)) = \text{pr}(\text{tr}(R^{t^{\times s}})) \cdot 2^{-n \cdot \text{rng}(R^{t^{\times s}})}$.

PROOF. Since $R^{t^{\times s}} \sim_{H,k,\emptyset} R$, we know that $\text{Fr}(x_i)$ or $\text{fr}(b)$ steps in $R^{t^{\times s}}$ maps to $\text{fr}(H(x_i))$ or $\text{fr}(b)$ steps in R , respectively, for $i \leq \text{rng}(R^{t^{\times s}})$. The first statement is as follows:

Based on the rule INEW in the operational semantics of IML output processes [2, p. 23], we get that the probability of generating the random number b based on the IML transition relation (i.e. $\rightarrow_{\frac{1}{2^n}}$ semantics) with respect to security parameter n is $\frac{1}{2^n}$. Therefore, the $\text{fr}(b)$ step has the probability $\frac{1}{2^n}$ in both $R^{t^{\times s}}$ and R . Hence, for $\text{fr}(b)$ steps, we have $\text{pr}(\text{tr}(R)) = \text{pr}(\text{tr}(R^{t^{\times s}}))$.

Based on the rule RNG(n) in the semantics of IML_{SB}, Fig. 12, we get that the probability of generating the symbolic random number x based on the mixed IML and symbolic transition relation (i.e., $\rightarrow_{1,H}$ semantics) with respect to security parameter n is 1 but we translate $\text{Fr}(x)$ to $\text{fr}(H(x))$ using interpretation H which have the probability $1 \cdot \frac{1}{2^n}$. The number of $\text{Fr}(x)$ steps in $R^{t^{\times s}}$ are $\text{rng}(R^{t^{\times s}})$, hence, the probability of $\text{Fr}(x)$ steps is $\text{pr}(\text{tr}(R^{t^{\times s}})) \cdot 2^{-n \cdot \text{rng}(R^{t^{\times s}})}$.

Therefore, we can conclude that $\text{pr}(\text{tr}(R)) = \text{pr}(\text{tr}(R^{t^{\times s}})) \cdot 2^{-n \cdot \text{rng}(R^{t^{\times s}})}$.

□

LEMMA A.3 (EXTRA RANDOMNESS). For a BIR program P , an IML process I , any upper bound $k \in \mathcal{N}$ on the number of RNG steps, a security parameter $n \in \mathcal{N}$, all IML traces $R \in \mathcal{R}^I(I(\{P\}))$, all interpretations H , all mixed IML and symbolic execution traces $R^{t^{\times s}} \in \mathcal{R}^{t^{\times s}}(I\{P\}, \eta_0[\text{RM} \mapsto \text{rm}_k, \text{rk} \mapsto 0])$ with $k = \text{rng}(R^{t^{\times s}})$, and for any $l \geq k$, if $R^{t^{\times s}} \sim_{H,k,\emptyset} R$, then, for all H' with $\text{dom}(H') \subseteq \text{rm}_l \setminus \text{rm}_k$, we have, $R^{t^{\times s}} \sim_{H',k,\emptyset} R$.

PROOF. Since $R^{t^{\times s}} \sim_{H,k,\emptyset} R$, we know that the number of RNG steps in $R^{t^{\times s}}$ is $\text{rng}(R^{t^{\times s}}) \leq k$. Based on the semantics of IML_{SB}, Fig. 12, we know that random symbolic values $x_i \in \text{rm}_k$ for $0 < i \leq k$ are used in RNG(n) rule by order of i . The random symbolic values $x_j \in \text{rm}_l$ for $l \geq j \geq k$ are not generated by RNG(n) rule in Fig. 12. Therefore, the random symbolic values $x_j \in \text{rm}_l$ for $l \geq j \geq k$ are not part of the mixed IML and symbolic state after RNG(n) call. Hence, we can conclude that modification of H does not affect trace equivalence and we have $R^{t^{\times s}} \sim_{H',k,\emptyset} R$ such that $H \subseteq H'$ and $\text{dom}(H') \subseteq \text{rm}_l \setminus \text{rm}_k$.

□

LEMMA A.4 (INJECTIVE EVENT TRACE INCLUSION (IML)). For a BIR program P , an IML process I , any security parameter $n \in \mathcal{N}$, an upper bound on the number of nonces $k \in \mathcal{N}$, and for $T^{t^{\times s}} = T^{t^{\times s}}(I\{P\}, \eta_0[\text{RM} \mapsto \text{rm}_k, \text{rk} \mapsto 0])$, there is an injective function ζ from

$$\left\{ \left(\begin{array}{c} t^{\times s} \in T^{t^{\times s}}, \\ H : \text{rm}_k \rightarrow \text{BS}_n^k \end{array} \right) \mid \text{rng}(t^{\times s}) = k \wedge \exists H'. H'|_{\text{rm}_k} = H \right\}$$

to $T(I(\{P\}), n)$ such that

$$\zeta(t^{\times s}, H) = t \implies \left(\begin{array}{c} \exists H', R^{t^{\times s}}, R. \\ R^{t^{\times s}} \sim_{H',k,\emptyset} R \wedge \\ \text{tr}(R^{t^{\times s}}) = t^{\times s} \wedge \text{tr}(R) = t \end{array} \right)$$

PROOF. From the skolemization of Thm. 7.3, we define $\zeta' : \mathcal{R}^{t^{\times s}} \times \mathcal{H} \rightarrow \mathcal{R}'$ such that $\zeta'(R^{t^{\times s}}, H') = R$ implies $R^{t^{\times s}} \sim_{H',k,\emptyset} R$. We define $\zeta : T^{t^{\times s}} \times \mathcal{H} \rightarrow \mathcal{T}$ such that $\zeta(t^{\times s}, H) = t$. Hence, we choose arbitrary $R^{t^{\times s}}$ and H' such that $\text{tr}(R^{t^{\times s}}) = t^{\times s}$ and $\text{tr}(\zeta'(R^{t^{\times s}}, H')) = \text{tr}(R) = t$.

Then, we prove the function ζ is injective by contradiction and assume for arbitrary $t^{\times s}, t^{\times s'}, H$, and H' such that $t^{\times s} \neq t^{\times s'} \vee H \neq H'$, we have $\zeta(t^{\times s}, H) = t = \zeta(t^{\times s'}, H')$. Therefore, $H(t^{\times s}) = H'(t^{\times s'})$ and for any event $a^{t^{\times s}}$ where $t^{\times s}$ and $t^{\times s'}$ are different, we have $H(a^{t^{\times s}}) = H'(a^{t^{\times s'}})$. Hence, there exist two cases:

- (1) $a^{t^{\times s}} \neq a^{t^{\times s'}}$
 - (a) $H = H'$: We assume that there exists a symbolic value d such that $H(\text{Ev}(d)) \neq H(\text{Fr}(d))$. Based on $R^{t^{\times s}} \sim_{H,k,\emptyset} R$, we have $\text{ev}(H(d)) \in t$ at the position j and $\text{Ev}(d) \in t^{\times s}$ at the same position j . Similarly, we have $\text{fr}(H(d)) \in t$ at the position j and $\text{Fr}(d) \in t^{\times s'}$ at the same position j . Hence, we get that we have two different t (i.e., at the position j). Since $a^{t^{\times s}} \neq a^{t^{\times s'}}$ gives a contradiction, we deduce that $a^{t^{\times s}} = a^{t^{\times s'}}$.
 - (b) $H \neq H'$: We assume that there exists a symbolic value d such that $H(\text{Ev}(d)) \neq H'(\text{Fr}(d))$. Based on $R^{t^{\times s}} \sim_{H,k,\emptyset} R$, we have $\text{ev}(H(d)) \in t$ at the position j and $\text{Ev}(d) \in t^{\times s}$ at the same position j . Based on $R^{t^{\times s'}} \sim_{H',k,\emptyset} R$, we have $\text{fr}(H'(d)) \in t$ at the position j and $\text{Fr}(d) \in t^{\times s'}$ at the same position j . Hence, we get that we have two different t (i.e., at the position j). Since $a^{t^{\times s}} \neq a^{t^{\times s'}}$ gives a contradiction, we deduce that $a^{t^{\times s}} = a^{t^{\times s'}}$.
- (2) $H \neq H' \wedge a^{t^{\times s}} = a^{t^{\times s'}}$
 - (a) $a^{t^{\times s}}$ is a $\text{Fr}(\cdot)$ event: We assume that there exists a random symbolic value $x_i \in \text{rm}_k$ for $0 < i \leq k$ such that $H(\text{Fr}(x_i)) \neq H'(\text{Fr}(x_i))$. Based on $R^{t^{\times s}} \sim_{H,k,\emptyset} R$, we have $\text{fr}(H(x_i)) \in t$ at the position j and $\text{Fr}(x_i) \in t^{\times s}$ at the same position j . Similarly, for $t^{\times s'}$ and H' and the same position j . Therefore, $H(x_i) = H'(x_i)$. Hence, from $x_i \in \text{rm}_k$, we get that $H(\text{Fr}(x_i)) = H'(\text{Fr}(x_i))$. Since $H \neq H'$ gives a contradiction, we deduce that $H = H'$.
 - (b) $a^{t^{\times s}}$ is a $\text{Ev}(\cdot)$ event: We assume that there exists a symbolic value d such that $H(\text{Ev}(d)) \neq H'(\text{Ev}(d))$. Based on $R^{t^{\times s}} \sim_{H,k,\emptyset} R$, we have $\text{ev}(H(d)) \in t$ at the position j and $\text{Ev}(d) \in t^{\times s}$ at the same position j . Similarly, for $t^{\times s'}$ and H' and the same position j . Because we generate symbolic values in a canonical form, $H(d) = H'(d)$. Therefore, we get that $H(\text{Ev}(d)) = H'(\text{Ev}(d))$. Since $H \neq H'$ gives a contradiction, we deduce that $H = H'$.

Therefore, we can conclude that the function ζ is injective.

□

LEMMA A.5 (INJECTIVE EVENT TRACE INCLUSION (SYMBOLIC EXECUTION)). For a BIR program P , an IML process I , any security parameter $n \in \mathcal{N}$, an upper bound on the number of nonces $k \in \mathcal{N}$, and for $T^{\text{ixb}} = T^{\text{ixb}}(I\{P\}, \eta_0[\text{RM} \mapsto \text{rm}_k, \mathbf{r}_k \mapsto 0])$, there is an injective function ζ from

$$\{(t^{\text{ixb}}, \text{rm}_k) \mid \text{rm}_k \in \text{BS}_n^k \wedge t^{\text{ixb}} \in T^{\text{ixb}}(I\{P\}, \eta_0[\text{RM} \mapsto \text{rm}_k, \mathbf{r}_k \mapsto 0])\}$$

to

$$\left\{ \left(\begin{array}{c} t^{\text{ixs}} \in T^{\text{ixs}} \\ H : \text{rm}_k \rightarrow \text{BS}_n^k \end{array} \right) \mid \text{s.t. } \exists H'. H' \upharpoonright_{\text{rm}_k} = H \right\}$$

such that

$$\zeta(t^{\text{ixb}}, \text{rm}_k) = (t^{\text{ixs}}, H)$$

$$\implies \left(\begin{array}{c} \text{pr}(t^{\text{ixb}}) \cdot 2^{-n \cdot k} = \text{pr}(t^{\text{ixs}}) \cdot 2^{-n \cdot k} \wedge \\ \text{tr}(R^{\text{ixb}}) = t^{\text{ixb}} \wedge \text{tr}(R^{\text{ixs}}) = t^{\text{ixs}} \end{array} \right)$$

PROOF. From the skolemization of Thm. 7.6, we define $\zeta' : \mathcal{R}^{\text{ixb}} \times \text{BS}_n^k \rightarrow \mathcal{R}^{\text{ixs}} \times \mathcal{H}$ such that $\zeta'(R^{\text{ixb}}, \text{rm}_k) = (R^{\text{ixs}}, H)$ implies $R^{\text{ixb}} \sim_{H,k} R^{\text{ixs}}$. We define $\zeta : T^{\text{ixb}} \times \text{BS}_n^k \rightarrow T^{\text{ixs}} \times \mathcal{H}$ such that $\zeta(t^{\text{ixb}}, \text{rm}_k) = (t^{\text{ixs}}, H)$. Hence, we choose arbitrary R^{ixb} and rm_k such that $\text{tr}(R^{\text{ixb}}) = t^{\text{ixb}}$ and $\text{tr}(\zeta'(R^{\text{ixb}}, \text{rm}_k)) = \text{tr}(R^{\text{ixs}}, H) = t^{\text{ixs}}$.

Then, we prove the function ζ is injective by contradiction and assume for arbitrary $t^{\text{ixb}}, t^{\text{ixb}'}, \text{rm}_k$, and rm_k' such that $t^{\text{ixb}} \neq t^{\text{ixb}'} \vee \text{rm}_k \neq \text{rm}_k'$, we have $\zeta(t^{\text{ixb}}, \text{rm}_k) = (t^{\text{ixs}}, H) = \zeta(t^{\text{ixb}'}, \text{rm}_k')$. Therefore, we get that $(t^{\text{ixb}}, \text{rm}_k) = H(t^{\text{ixs}}) = (t^{\text{ixb}'}, \text{rm}_k')$. Hence, there exist two cases:

- (1) $\text{rm}_k \neq \text{rm}_k'$

We assume that there exists a random value $x_i \in \text{rm}_k$ and a random value $x_i' \in \text{rm}_k'$ for $0 < i \leq k$ such that $x_i \neq x_i'$. Based on $R^{\text{ixb}} \sim_{H,k} R^{\text{ixs}}$, we have $\text{Fr}(x_i) \in t^{\text{ixs}}$ at the position j and $\text{Fr}(x_i) \in t^{\text{ixb}}$ at the same position j such that $H(x_i) = x_i$. Based on $R^{\text{ixb}'} \sim_{H,k} R^{\text{ixs}}$, we have $\text{Fr}(x_i') \in t^{\text{ixs}}$ at the position j and $\text{Fr}(x_i') \in t^{\text{ixb}'}$ at the same position j such that $H(x_i') = x_i'$. Therefore, we get that $x_i = H(x_i) = x_i'$. Since $\text{rm}_k \neq \text{rm}_k'$ gives a contradiction, we deduce that $\text{rm}_k = \text{rm}_k'$.

- (2) $t^{\text{ixb}} \neq t^{\text{ixb}'} \wedge \text{rm}_k = \text{rm}_k'$

Let a^{ixb} and $a^{\text{ixb}'}$ be the earliest mixed IML and BIR events where t^{ixb} and $t^{\text{ixb}'}$ are different. Based on $R^{\text{ixb}} \sim_{H,k} R^{\text{ixs}}$, we have $a^{\text{ixs}} \in t^{\text{ixs}}$ at the position j and $a^{\text{ixb}} \in t^{\text{ixb}}$ at the same position j such that $H(a^{\text{ixs}}) = a^{\text{ixb}}$. Based on $R^{\text{ixb}'} \sim_{H,k} R^{\text{ixs}}$, we have $a^{\text{ixs}} \in t^{\text{ixs}}$ at the position j and $a^{\text{ixb}'} \in t^{\text{ixb}'}$ at the same position j such that $H(a^{\text{ixs}}) = a^{\text{ixb}'}$. Therefore, we get that $a^{\text{ixb}} = H(a^{\text{ixs}}) = a^{\text{ixb}'}$. Since $t^{\text{ixb}} \neq t^{\text{ixb}'}$ gives a contradiction, we deduce that $t^{\text{ixb}} = t^{\text{ixb}'}$.

Therefore, we can conclude that the function ζ is injective. Since $R^{\text{ixb}} \sim_{H,k} R^{\text{ixs}}$, we know that $\text{Fr}(x_i)$ or $\text{fr}(b)$ steps in R^{ixs} maps to $\text{Fr}(H(x_i))$ or $\text{fr}(b)$ steps in R^{ixb} , respectively, for $i \leq k$. The first statement is as follows:

Based on the rule INew in the operational semantics of IML output processes [2, p. 23], we get that the probability of generating the random number b based on the IML transition relation (i.e. $\longrightarrow_{\frac{1}{2^n}}$ semantics) with respect to security parameter n is $\frac{1}{2^n}$. Therefore, the $\text{fr}(b)$ step has the probability $\frac{1}{2^n}$ in both t^{ixb} and t^{ixs} . Hence, for $\text{fr}(\cdot)$ steps, we have $\text{pr}(t^{\text{ixb}}) = \text{pr}(t^{\text{ixs}})$.

Based on the rule RNG(n) in the semantics of IML_{SB}, Fig. 12, we get that the probability of generating the symbolic random number x based on the mixed IML and symbolic transition relation (i.e., $\longrightarrow_{I,H}$ semantics) with respect to security parameter n is 1 but we map $\text{Fr}(x)$ to $\text{Fr}(H(x))$ using interpretation H which have the probability $1 \cdot \frac{1}{2^n}$. The number of $\text{Fr}(x)$ steps in t^{ixs} are k , hence, the probability of $\text{Fr}(x)$ steps is $\text{pr}(t^{\text{ixs}}) \cdot 2^{-n \cdot k}$.

Based on the rule normal in the mixed semantics of IML and BIR Fig. 16 and Sec. 4, we get that the probability of generating the random number x based on the mixed IML and BIR transition relation (i.e., \longrightarrow_I semantics) with respect to security parameter n is 1 but we extracting x from the random memory RM with length n which have the probability $1 \cdot \frac{1}{2^n}$. The number of $\text{Fr}(x)$ steps in t^{ixb} are k , hence, the probability of $\text{Fr}(x)$ steps is $\text{pr}(t^{\text{ixb}}) \cdot 2^{-n \cdot k}$.

Therefore, we can conclude that $\text{pr}(t^{\text{ixb}}) \cdot 2^{-n \cdot k} = \text{pr}(t^{\text{ixs}}) \cdot 2^{-n \cdot k}$. \square

A.7 Extension for Multi-Programs

In this section, we extend our results for multiple programs by establishing theorems A.6 to A.8. Given an IML process I and implementations P_1, \dots, P_m of protocol participants in BIR, we denote $I\{P_1, \dots, P_m\}$ which parties running in parallel with an IML attacker and communicate through a channel. For P_1, \dots, P_m which are symbolically executed and translated into IML processes $\langle P_1 \rangle, \dots, \langle P_m \rangle$, the IML process $I\{\langle P_1 \rangle, \dots, \langle P_m \rangle\}$ describes the parallel composition of m parties in the presence of an attacker. The following theorem indicates that IML_{SB} and IML preserve the simulation relation for m programs.

THEOREM A.6 (IML_{SB}-IML TRACE INCLUSION*). Let P_1, \dots, P_m be BIR programs, I be an IML process and $k \in \mathcal{N}$ is any upper bound on RNG steps, then, for all mixed IML and symb. execution traces $R^{\text{ixs}} \in \mathcal{R}^{\text{ixs}}(I\{P_1, \dots, P_m\}, \eta_0[\text{RM} \mapsto \text{rm}_k, \mathbf{r}_k \mapsto 0])$ such that $\text{rng}(R^{\text{ixs}}) \leq k$, there exist an IML trace $R \in \mathcal{R}^I(I\{\langle P_1 \rangle, \dots, \langle P_m \rangle\})$ and an interpretation H such that $R^{\text{ixs}} \sim_{H, \emptyset, \emptyset, k} R$.

PROOF. By Thm. 7.3, for each $0 < i \leq m$, we have that for all mixed IML and symb. execution traces $R_i^{\text{ixs}} \in \mathcal{R}_i^{\text{ixs}}(I\{P_i\}, \eta_0[\text{RM} \mapsto \text{rm}_k, \mathbf{r}_k \mapsto 0])$, exist an IML trace $R_i \in \mathcal{R}_i^I(I\{\langle P_i \rangle\})$ and an interpretation H_i s.t. $R_i^{\text{ixs}} \sim_{H_i, \emptyset, \emptyset, k} R_i$. Then, we can conclude that for all mixed IML and symb. execution traces $R^{\text{ixs}} \in \mathcal{R}^{\text{ixs}}(I\{P_1, \dots, P_m\}, \eta_0[\text{RM} \mapsto \text{rm}_k, \mathbf{r}_k \mapsto 0])$, there exist an IML trace $R \in \mathcal{R}^I(I\{\langle P_1 \rangle, \dots, \langle P_m \rangle\})$ and a H such that $R^{\text{ixs}} \sim_{H, \emptyset, \emptyset, k} R$ and $H_i \subseteq H$ for $0 < i \leq m$. \square

Thm. A.6 proves that the IML model resulting from the translation of m programs covers all behaviors in the mixed IML and symbolic execution semantics. To ensure that the extracted IML model for m protocol parties preserves the semantics of their implementations in binary, we have to show Thm. A.7.

THEOREM A.7 (IML_B-IML_{SB} TRACE INCLUSION*). Let P_1, \dots, P_m be BIR programs, I be an IML process and $k \in \mathcal{N}$ is any upper bound on RNG steps, then, for all mixed IML and BIR traces $R^{\text{ixb}} \in \mathcal{R}^{\text{ixb}}(I\{P_1, \dots, P_m\}, \eta_0[\text{RM} \mapsto \text{rm}_k, \mathbf{r}_k \mapsto 0])$ such that $\text{rng}(R^{\text{ixb}}) \leq k$, there exist a mixed IML and SBIR trace $R^{\text{ixs}} \in \mathcal{R}^{\text{ixs}}(I\{P_1, \dots, P_m\}, \eta_0[\text{RM} \mapsto \text{rm}_k, \mathbf{r}_k \mapsto 0])$ and an H such that $R^{\text{ixb}} \sim_{H,k} R^{\text{ixs}}$.

PROOF. By Thm. 7.6, for each $0 < i \leq m$, we have for all mixed *IML* and *BIR* traces $R_i^{\text{xb}} \in \mathcal{R}_i^{\text{xb}}(I\{P_1\}, \eta_0[\text{RM} \mapsto \text{rm}_k, \mathbf{r}_k \mapsto 0])$, there is a mixed *IML* and symbolic execution trace $R_i^{\text{xs}} \in \mathcal{R}_i^{\text{xs}}(I\{P_1\}, \eta_0[\text{RM} \mapsto \text{rm}_k, \mathbf{r}_k \mapsto 0])$, and an interpretation H_i such that $R_i^{\text{xb}} \sim_{H_i, k} R_i^{\text{xs}}$. Then, we can conclude that for all mixed *IML* and *BIR* traces $R^{\text{xb}} \in \mathcal{R}^{\text{xb}}(I\{P_1, \dots, P_m\}, \eta_0[\text{RM} \mapsto \text{rm}_k, \mathbf{r}_k \mapsto 0])$, there exist a mixed *IML* and symbolic execution trace $R^{\text{xs}} \in \mathcal{R}^{\text{xs}}(I\{P_1, \dots, P_m\}, \eta_0[\text{RM} \mapsto \text{rm}_k, \mathbf{r}_k \mapsto 0])$ and an interpretation H such that $R^{\text{xb}} \sim_{H, k} R^{\text{xs}}$ and $H_i \subseteq H$ for $0 < i \leq m$. \square

Thm. A.7 states that symbolically executed *BIR* programs P_1, \dots, P_m preserve all behaviors of the same programs in the concrete execution for an appropriately chosen interpretation and random memory. In the following, we measure the success probability of the attacker for *BIR* programs in *IML*_B execution semantics $I\{P_1, \dots, P_m\}$ and extracted *IML* process $I\{\langle P_1 \rangle, \dots, \langle P_m \rangle\}$. Then we show by Thm. A.8 that $I\{P_1, \dots, P_m\}$ is at least as secure as $I\{\langle P_1 \rangle, \dots, \langle P_m \rangle\}$ with respect to any trace property ψ , security parameter n and upper bound k on the number of RNG steps.

THEOREM A.8 (TRANSLATION PRESERVES ATTACKS*). *Given an *IML* process I , *BIR* programs P_1, \dots, P_m , a security parameter $n \in \mathcal{N}$, a trace property ψ and an upper bound $k \in \mathcal{N}$ on the number of RNG steps in $T^{\text{xb}}(I\{P_1, \dots, P_m\}, \eta_0[\text{RM} \mapsto \text{rm}_k, \mathbf{r}_k \mapsto 0])$, we get that $\text{insec}(I\{P_1, \dots, P_m\}, n, k, \psi) \leq \text{insec}(I\{\langle P_1 \rangle, \dots, \langle P_m \rangle\}, n, \psi)$*

PROOF. The insecurity of $I\{\langle P_1 \rangle, \dots, \langle P_m \rangle\}$ w.r.t. ψ is as follows:

$$\text{insec}(I\{\langle P_1 \rangle, \dots, \langle P_m \rangle\}, n, \psi) = \sum_{t \in T(I\{\langle P_1 \rangle, \dots, \langle P_m \rangle\}, n) \cap \psi_{\neg}} \text{pr}(t)$$

Note that by Thm. A.6, we have $R^{\text{xs}} \sim_{H, k, \langle \cdot \rangle} R$ which implies $H(t^{\text{xs}}) \in \psi_{\neg} \Leftrightarrow t \in \psi_{\neg}$

$$= \sum_{0 < i \leq m} \sum_{\substack{t_i \in \\ T_i(I\{\langle P_i \rangle\}, n) \cap \psi_{\neg}}} \text{pr}(t_i)$$

$$\stackrel{\text{Def. 8.2}}{=} \sum_{0 < i \leq m} \text{insec}(I\{\langle P_i \rangle\}, n, \psi)$$

$$\stackrel{\text{Thm. 8.3}}{\geq} \sum_{0 < i \leq m} \text{insec}(I\{P_i\}, n, k, \psi)$$

$$\stackrel{\text{Def. 8.1}}{=} \sum_{0 < i \leq m} 2^{-n \cdot k} \cdot \sum_{\substack{\text{rm}_k \in BS_n^k \\ t_i^{\text{xb}} \in \psi_{\neg} \cap \\ T_i^{\text{xb}}(I\{P_i\}, \eta_0[\text{RM} \mapsto \text{rm}_k, \mathbf{r}_k \mapsto 0])}} \text{pr}(t_i^{\text{xb}})$$

Note that by Thm. A.7, we have $R^{\text{xb}} \sim_{H, k} R^{\text{xs}}$ which implies $H(t^{\text{xs}}) \in \psi_{\neg} \Leftrightarrow t^{\text{xb}} \in \psi_{\neg}$

$$= 2^{-nk} \cdot \sum_{\substack{\text{rm}_k \in BS_n^k \\ t^{\text{xb}} \in T^{\text{xb}} \cap \psi_{\neg}}} \text{pr}(t^{\text{xb}})$$

$$= \text{insec}(I\{P_1, \dots, P_m\}, n, k, \psi)$$

\square