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 [64].*

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, 15 pages. https://doi.org/10.1145/3576915.3623090

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. Nevertheless, despite the simplicity of such protocols, their design and implementation are notoriously error-prone, and there have been many attacks targeting either their design (e.g., man-in-the-middle attacks like the triple-handshake attack on TLS [15]) or 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 [74]. This is because enabling aggressive compiler optimizations can lead to missing source-level checks like code intended to detect integer overflows [70] or null pointers [74]. Compilers can invalidate code for secret scrubbing [67] or even turn constant-time code into a nonconstant-time binary [68]. 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

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11…$15.00
https://doi.org/10.1145/3576915.3623090

C Code

```
fun :
  char K[64]; // secret key
  if(getKey(K,sizeof(K)))
   {
     do something
   }
  memset(K,0,sizeof(K));
            ...
```

Simplified Assembly

```
fun :
  mov w1, 64
  mov x0, #key K address
  bl  getKey
  cbz w0, .L3
    do something
  .L3:
    ret
```

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 [69], 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, 71] and CryptoVerif [21]. Fig. 2 depicts the pipeline of the CryptoBap.

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 differences of the two approaches.) We adopt their process calculus *intermediate model language* (*IML*[1]) 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

[1]We 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.

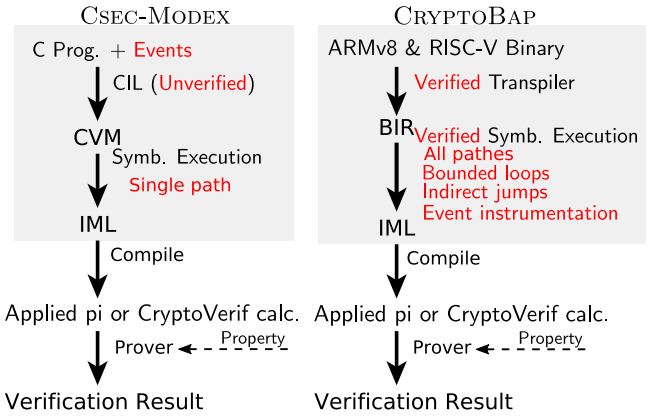

Figure 2: CryptoBap vs. CSec-modex

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 are 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 CryptoBap's approach. As in Fig. 2, CryptoBap (source code is available at https://github.com/FMSecure/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, CryptoBap 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

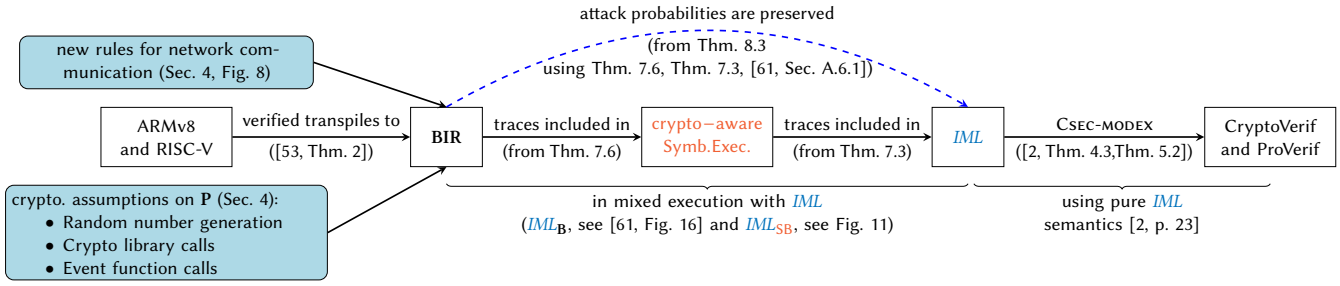

Figure 3: Organization of the CryptoBap approach.

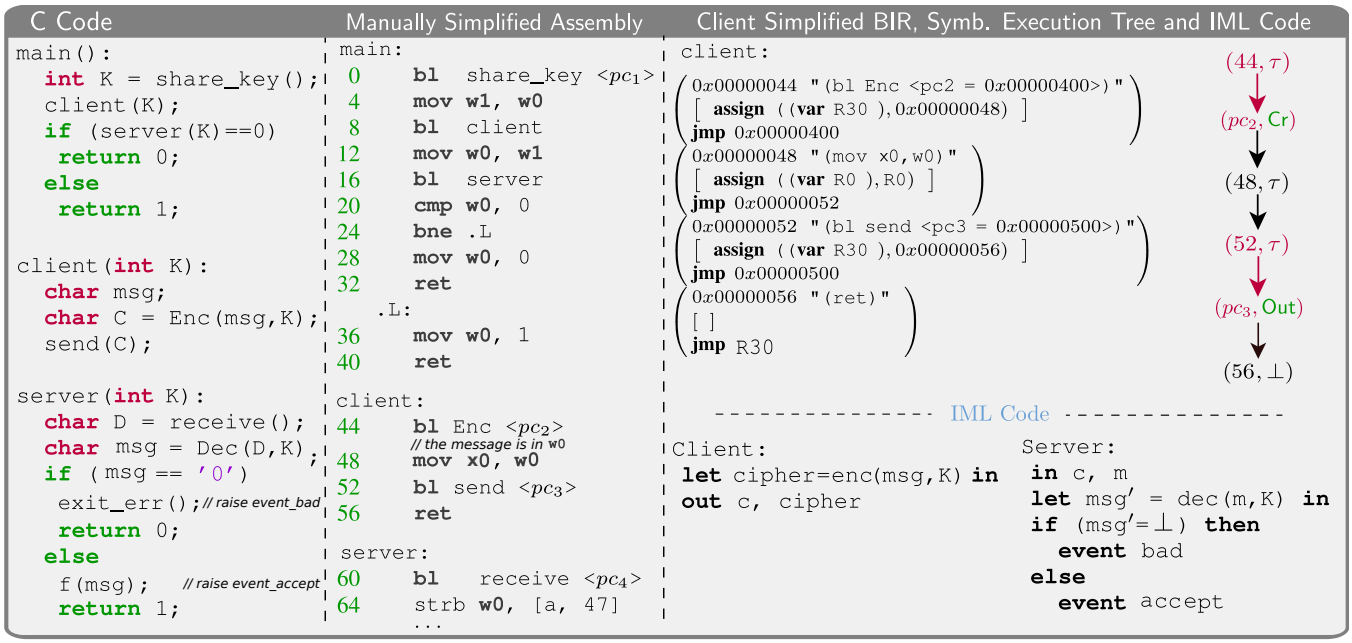

Figure 4: Running example. Function addresses are chosen randomly. Line numbers and pc_i are addresses of instructions and functions in the memory.

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 HACL* library [76], CompCert in conjunction with FCF [11, 75] 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 [65], Verypto [5] and CryptHOL [55], which are easier to combine with other techniques (this was demonstrated [12, 65] 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 [66]. 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, 63]. 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, 72, 73]. 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 CryptoBap 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, CryptoBap handles **all execution paths** of protocols, including those that contain **conditionals**, **bounded loops**, and **indirect jumps**. More specifically, Aizatulin's approach restricts the input programs to programs that have no "else" branches and no loops. Conceptually, one may see this as irrelevant because one might speculate that all paths outside the "main" part are not useful, i.e., they do not produce network output, or at least none that reveals cryptographic information (e.g., error codes). But this is still restrictive. First, many protocols are specified to produce network output in error cases, for example, *decoy messages* in anonymity protocols or error messages that are encrypted. Moreover, seemingly regular protocols have "else" branches as part of their "main" message flow, once we look at them with the level of detail necessary to analyze implementations. For example, cypher suite negotiation in TLS must be formulated with multiple branches depending on the input message.

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, CryptoBap automates releasing events during symbolic execution and minimizes user involvement.

3 BACKGROUND

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

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.

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

Figure 5: The BIR syntax

3.1 HolBA framework & Vanilla symbolic exec.

CryptoBap 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 $\diamond_{\mathbf{b}}$ and $\diamond_{\mathbf{u}}$) for finite integer arithmetic, memory operations, and conditionals. Fig. 4 presents a **BIR** snippet for the running example.

A **BIR** state $(\eta, \mathbf{pc}) \in \mathbf{S}$ consists of an environment $\eta : \mathbf{Bvar} \mapsto \mathbf{Bval}$ which maps variables, i.e., registers $\mathbf{r_i}$ and memory locations *Mem*, to values and a program counter **pc** that holds the label of the executing **BIR** block. The relation $\rightarrow \subseteq \mathbf{S} \times \mathbf{S}$ 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 $H : \mathrm{SE} \rightarrow \mathbf{Bval}$. In addition to the symbolic environment $\eta : \mathbf{Bvar} \mapsto \mathrm{SE}$, the SBIR state $(\phi, \eta, \mathbf{pc}) \in \mathrm{S}$ also contains a path condition $\phi \in \mathrm{SE}$ and a **pc** that is kept concrete to obtain a concrete control flow.

Let $\twoheadrightarrow : \mathrm{S} \times \mathrm{S}$ be the single-step transition relation of SBIR and $\twoheadrightarrow^n$ (or $\twoheadrightarrow^+$) denote a multi-step symbolic transition. We write $\mathbf{s_i} \rightarrow^n_{\mathbf{L}} \mathbf{s_j}$ to restrict the transition from $\mathbf{s_i}$ to $\mathbf{s_j}$ 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* $\mathbf{s_i}, H, \mathbf{s_j}$, $\mathrm{s_i}$ *s.t.* $\mathbf{s_i} \sim_H \mathrm{s_i}$, *if* $\mathbf{s_i} \rightarrow \mathbf{s_j}$ *then there exist an* H' *and* $\mathrm{s_j}$ *s.t.* $H \subseteq H'$ *and* $\mathrm{s_i} \twoheadrightarrow \mathrm{s_j}$ *and* $\mathbf{s_j} \sim_{H'} \mathrm{s_j}$.

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.

3.2 Csec-modex toolchain & *IML*

Aizatulin 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 $op(e_1, \ldots, e_m)$ denotes the function application. *IML* expressions are evaluated with respect to an environment $\eta : Ivar \mapsto BS \cup \{\perp\}$ 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 $e_1, \ldots, e_m$ indicate the protocol participants' identifier. The construct $new\ x : t$ generates

$d, e \in Iexp$:= expression

$b \in BS,\ x \in Ivar$	bitstrings, variables
$op(e_1, \ldots, e_m)$	computation, $op \in Ops$

$P, Q \in IML$:= process

$0, P \mid Q$	nil, parallel composition
$!^{i \le m}\ P$	replicating P, m times
$new\ x{:}t; P$	randomness
$in(c[e_1, \ldots, e_m], x); P$	input
$out(c[e_1, \ldots, e_m], e); P$	output
$event(d_1, \ldots, d_m); P$	event
$if\ e\ then\ P\ [else\ Q]$	conditional
$let\ x = e\ in\ P$	assignment

Figure 6: A fragment of *IML* syntax

$$\frac{t = \texttt{fixed}_n \text{ for some } n \in \mathcal{N} \qquad |b| = n}{(\eta, new\ x{:}t; P), Q \xrightarrow{fr(b)}\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow_{\frac{1}{2^n}} (\eta[x \mapsto b], P), Q}$$

$$\frac{[\![e]\!]_\eta = b \neq \bot \quad b' = truncate(b, maxlen(c))}{\forall j \le m : [\![e_j]\!]_\eta = b_j \neq \bot \quad Q' = reduce(\{(\eta, Q)\})}$$
$$\frac{\exists!(\eta', Q') \in Q : Q' = in(c[e'_1, \ldots, e'_m], x'); P' \wedge \forall j \le m : [\![e_j']\!]_{\eta'} = b_j \neq \bot}{(\eta, out(c[e_1, \ldots, e_m], e); Q), Q \xrightarrow{}\!\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow_1 (\eta'[x' \mapsto b'], P'), Q \uplus Q' \setminus \{(\eta', Q')\}}$$

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

a uniform random number of type t and $event(d_1, \ldots, d_m)$ 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 $(\emptyset, out(c, \varepsilon); 0), reduce(\emptyset, Q)$ where *reduce* represents a function that executes a sequence of processes inside Q (e.g., $Q = P_1; P_2; \ldots$) until an input process waiting for a message from channel c is reached (see 2nd rule in Fig. 7). We use $\xrightarrow{a}\!\!\!\!\!\rightarrow_p \subseteq S \times S$ 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 $ev(b_1, \ldots, b_m)$, where ev is an event symbol and $b_1, \ldots, b_m$ are bitstrings. Moreover, an *IML* trace is defined as $R = s_1 \xrightarrow{a_0}\!\!\!\!\!\rightarrow_{p_1} \cdots \xrightarrow{a_n}\!\!\!\!\!\rightarrow_{p_n} s_n \subseteq \mathcal{R}^\iota(Q)$.

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 [61, 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: $\mathbf{L} = \mathbf{L}_\mathcal{N} \uplus \mathbf{L}_{\mathbf{Op}} \uplus \mathbf{L}_\mathcal{A} \uplus \mathbf{L}_\mathcal{R} \uplus \mathbf{L}_\mathcal{E}$. The sets $\mathbf{L}_{\mathbf{Op}} = \bigcup_{\mathbf{op} \in \mathbf{Op}} \mathbf{L}_\mathcal{A}, \mathbf{L}_\mathcal{R}, \mathbf{L}_\mathcal{E}$ 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 $\mathbf{L}_\mathcal{N}$. Moreover, a specific label set $\mathbf{L}_\mathcal{L} \subset \mathbf{L}_\mathcal{N}$ 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 $\ell \in \{\mathbf{L}_\mathcal{N}, \mathbf{L}_{\mathbf{Op}}, \mathbf{L}_\mathcal{A}, \mathbf{L}_\mathcal{R}, \mathbf{L}_\mathcal{E}\}$—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, $\xrightarrow{\mathbf{a}} \subseteq \mathbf{S} \times \mathbf{E} \times \mathbf{S}$, where $\mathbf{E}$ is the set of observable events plus the silent transition τ. Then, a multi-step **BIR** transition $\mathbf{s_0} \xrightarrow{(\mathbf{a_1}, \ldots, \mathbf{a_m})}{}^{+} \mathbf{s_n}$ exists if $\mathbf{s_0} \xrightarrow{\mathbf{a_1}}{}^{*} \mathbf{s_1} \ldots \mathbf{s_{n-1}} \xrightarrow{\mathbf{a_m}}{}^{*} \mathbf{s_n}$ where $\mathbf{s_i} \xrightarrow{\mathbf{a}}{}^{*} \mathbf{s_j}$ if $\mathbf{s_i} \xrightarrow{\tau}{}^{*} \xrightarrow{\mathbf{a}} \xrightarrow{\tau}{}^{*} \mathbf{s_j}$ is reminiscent to the big-step semantics. Also, we use $\mathcal{R}^{\mathbf{b}}(\mathbf{P}, \mathbf{s_0})$ to denote the set of execution traces of $\mathbf{P}$ starting from the initial state $\mathbf{s_0}$.

We define $ret : \mathbf{S} \to \mathbf{L}$ to obtain the next execution point immediately reachable after returning from a call. $mstore : (\mathbf{Bvar} \mapsto \mathbf{Bval}) \times \mathbf{Bvar} \times 2^{\mathbf{Bvar}} \times BS \times \mathcal{N} \to (\mathbf{Bvar} \mapsto \mathbf{Bval}) \times \mathbf{Bval}$ stores bitstrings into the memory in the **BIR** environment. Given $heap \in Mem$ and $l \in \{1, 8, 16, 32, 64, 128\}$ and $|b|$ a multiple of l that can be encoded in 128 bits, $mstore(\eta, heap, Mem, b, l)$ stores b in l-bit chunks, preceded by l (encoded as a word) in the memory $Mem \subseteq \mathbf{Bvar}$, 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 $\|$ be the byte concatenation operator. Then, $mload(\eta, \mathbf{a}) \stackrel{\text{def}}{=} \Big\|_{i=1\ldots\eta(\mathbf{a})} \eta(\mathbf{a} + i)$ for the given η and the address '$\mathbf{a}$'.

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 $\mathbf{s_0}$ and a random tape $rm_k \in \{0,1\}^{kl \times w}$ the state $\mathbf{s_0}.\eta[\mathbf{RM} \mapsto rm_k, \mathbf{r_k} \mapsto 0]$ is an instance of this initial state. To extract a random number

$$\frac{\mathbf{pc} \in \xi_{\mathcal{A}_s} \quad \mathbf{pc}' = ret(\eta, \mathbf{pc}) \quad \mathbf{a} = \eta[\mathbf{r_0}] \quad e = mload(\eta, \mathbf{a})}{\mathbf{P} \vdash (\eta, \mathbf{pc}), c[\mathbf{e_1}, \ldots, \mathbf{e_m}] \xrightarrow{\mathbf{Out}(e, (\mathbf{e_1}, \ldots, \mathbf{e_m}))} (\eta, \mathbf{pc}'), e :: c[\mathbf{e_1}, \ldots, \mathbf{e_m}]}\ \mathcal{A}_s$$

$$\frac{\mathbf{pc} \in \xi_{\mathcal{A}_r} \quad \mathbf{pc}' = ret(\eta, \mathbf{pc}) \quad (\eta', \mathbf{a}) = mstore(\eta, heap_{\mathcal{A}}, Mem_{\mathcal{A}}, e', 128)}{\mathbf{P} \vdash (\eta, \mathbf{pc}), e' :: c[\mathbf{e_1}, \ldots, \mathbf{e_m}] \xrightarrow{\mathbf{In}(e', (\mathbf{e_1}, \ldots, \mathbf{e_m}))} (\eta'[\mathbf{r_0} \mapsto \mathbf{a}], \mathbf{pc}'), c[\mathbf{e_1}, \ldots, \mathbf{e_m}]}\ \mathcal{A}_r$$

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

of size l from **RM**, we define $\mathfrak{R} : (\mathbf{Bvar} \to \mathbf{Bval}) \times \mathcal{N} \to BS$ which returns a value from **RM** yet unread: $\mathfrak{R}(\eta, n) \stackrel{\text{def}}{=}$ let $\mathbf{x} = \eta[\mathbf{r_k}]$ in $\|_{i=0\ldots l-1} \eta[\mathbf{RM} + \mathbf{x} + i]$. 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 $\mathbf{L}_{\mathcal{R}}$ with $\mathbf{l}_{\mathcal{R}} \in \mathbf{L}_{\mathcal{R}}$ one of its entry points an RNG function, if for any entering state $(\eta_0, \mathbf{l}_{\mathcal{R}})$ for which $\mathfrak{R}(\eta_0, \mathbf{n})$ is defined, and execution point $(\eta'', \mathbf{l})$ after returning from RNG, i.e., $\mathbf{l} = ret(\eta_0, \mathbf{l}_{\mathcal{R}})$, the output register holds the address of a copy of the random value and $\mathbf{r_k}$ is updated, i.e., $\eta'' = \eta'[\mathbf{r_0} \mapsto \mathbf{a}; \mathbf{r_k} \mapsto \eta_0[\mathbf{r_k}] + l]$ with $(\eta', \mathbf{a}) = mstore(\eta_0, heap, Mem, x_i, 128)$ s.t. $x_i = \mathfrak{R}(\eta_0, n)$. We denote RNG steps as $(\eta_0, \mathbf{l}_{\mathcal{R}}) \xrightarrow{\mathbf{Fr}(x_i)}{}^*_{\mathbf{L}_{\mathcal{R}}} (\eta'', \mathbf{l})$, 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.

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 $c[\mathbf{e_1}, \ldots, \mathbf{e_m}]$, where $\mathbf{e_1}, \ldots, \mathbf{e_m}$ are expressions which identify communicating parties and their channel c. To send a message e ($\xi_{\mathcal{A}_s}$ in Fig. 8), we fetch the value of e from the memory address $\eta[\mathbf{r_0}]$, put it on the channel $e :: c[\mathbf{e_1}, \ldots, \mathbf{e_m}]$, and release the $\mathbf{Out}(e, (\mathbf{e_1}, \ldots, \mathbf{e_m}))$ event.

To receive a message $e' :: c[\mathbf{e_1}, \ldots, \mathbf{e_m}]$, 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 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., $Mem_{\mathbf{Op}}$. We denote library steps with $(\eta_0, \mathbf{l_{op}}) \xrightarrow{\mathbf{Cr}(v)}{}^*_{\mathbf{L_{Op}}} (\eta'', \mathbf{l})$, and expect *transitions using labels outside* $\mathbf{L_{Op}}$ *do not change the memory of library calls.*

We call $\mathbf{L_{op}} \subseteq \bigcup_{\mathbf{op} \in \mathbf{Op}}$ the library implementation of **op** (with arity m) and $\mathbf{l_{op}} \in \xi_{\mathbf{Op}}$ one of its entry points, if for any entering state $(\eta_0, \mathbf{l_{op}})$ and the return state $(\eta'', \mathbf{l})$, the function result $v = \mathbf{op}(b_1, \ldots, b_m)$ for $b_i = mload(\eta, \mathbf{r_i})$, $i \in \{1, \ldots, m\}$, is stored in a heap and its address is put into $\mathbf{r_0}$: $\eta'' = \eta'[\mathbf{r_0} \mapsto \mathbf{a}]$ where $(\eta', \mathbf{a}) = mstore(\eta_0, heap_{\mathbf{Op}}, Mem_{\mathbf{Op}}, v, 128)$.

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., `f(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 $(\eta_0, \mathbf{l}_{\mathcal{E}}) \xrightarrow{\mathbf{Ev}(b_1, \ldots, b_m)}{}^*_{\mathbf{L}_{\mathcal{E}}} (\eta_0, \mathbf{l})$ where $\mathbf{l}_{\mathcal{E}} \in \mathbf{L}_{\mathcal{E}}$ is the entry point, $b_i = mload(\eta, \mathbf{r_i})$ for $i \in \{1, \ldots, m\}$ are event parameters, and $\mathbf{l} \in \mathbf{L}_{\mathcal{N}}$.

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, $range(f)$ returns the image of f. For library calls, we define an *oracle* $L : \xi_{\mathbf{Op}} \times (\mathbf{Bvar} \mapsto \mathsf{SE}) \to \mathsf{SE}$ 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., $\stackrel{a}{\twoheadrightarrow} \subseteq \mathsf{S} \times \mathsf{E} \times \mathsf{S}$, and use $\mathcal{R}^{\mathsf{s}}(\mathbf{P}, \mathsf{s}_0)$ to denote the set of symbolic traces of **P** starting at s_0.

Bounded Loops. Loops can naïvely 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 [69]. 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 $processLoop : \mathsf{S} \to \mathsf{SE} \times (\mathbf{Bvar} \mapsto \mathsf{SE})$ represents our implementation to summarize loops' effect. It takes as input the symbolic state of the loop entry

$$\frac{\mathbf{pc} \in \xi_{\mathcal{A}_s} \quad \mathbf{pc}' = ret(\phi, \eta, \mathbf{pc}) \quad \mathbf{a} = \eta[\mathbf{r_0}] \quad \mathsf{e} = mload(\eta, \mathbf{a})}{\mathbf{P} \vdash (\phi, \eta, \mathbf{pc}), c[\mathsf{e}_1, \ldots, \mathsf{e}_m] \xrightarrow{\mathsf{Out}(\mathsf{e}, (\mathsf{e}_1, \ldots, \mathsf{e}_m))} (\phi, \eta, \mathbf{pc}'), \mathsf{e} :: c[\mathsf{e}_1, \ldots, \mathsf{e}_m]}\ \mathcal{A}_s$$

$$\frac{\mathbf{pc} \in \xi_{\mathcal{E}} \quad \mathbf{pc}' = ret(\phi, \eta, \mathbf{pc}) \quad (\mathsf{d}_1, \ldots, \mathsf{d}_m) \notin range(\eta)}{\mathbf{P} \vdash (\phi, \eta, \mathbf{pc}) \xrightarrow{\mathsf{Ev}(\mathsf{d}_1, \ldots, \mathsf{d}_m)} (\phi, \eta, \mathbf{pc}')}\ \text{event}$$

$$\frac{\mathbf{pc} \in \xi_{\mathcal{L}} \quad \mathbf{pc}' = exit(\mathbf{pc}) \quad \mathsf{t} \notin range(\eta) \quad (\phi', \eta') = processLoop(\phi, \eta, \mathbf{pc})}{\mathbf{P} \vdash (\phi, \eta, \mathbf{pc}) \xrightarrow{\mathsf{loop}(\mathsf{t})}{}^{+} (\phi', \eta', \mathbf{pc}')}\ \text{loop}$$

$$\frac{\begin{array}{ccc}\mathbf{pc} \in \xi_{\mathcal{R}} & i = \left\lfloor \frac{\eta[\mathbf{r_k}]}{l} \right\rfloor + 1 & \mathsf{x}_i = \Re(\eta, n) \\ \mathbf{pc}' = ret(\phi, \eta, \mathbf{pc}) & \mathsf{x}_i \notin range(\eta) & (\eta', \mathbf{a}) = mstore(\eta, heap, Mem, \mathsf{x}_i, 128)\end{array} \quad \eta'' = \eta'[\mathbf{r_0} \mapsto \mathbf{a}; \mathbf{r_k} \mapsto \eta[\mathbf{r_k}] + l]}{\mathbf{P} \vdash (\phi, \eta, \mathbf{pc}) \xrightarrow{\mathsf{Fr}(\mathsf{x}_i)} (\phi, \eta'', \mathbf{pc}')}\ \text{RNG}(n)$$

$$\frac{\mathbf{pc} \in \xi_{\mathcal{A}_r} \quad \mathbf{pc}' = ret(\phi, \eta, \mathbf{pc}) \quad \mathsf{e} \notin range(\eta) \quad (\eta', \mathbf{a}) = mstore(\eta, heap_{\mathcal{A}}, Mem_{\mathcal{A}}, \mathsf{e}, 128)}{\mathbf{P} \vdash (\phi, \eta, \mathbf{pc}), \mathsf{e} :: c[\mathsf{e}_1, \ldots, \mathsf{e}_m] \xrightarrow{\mathsf{In}(\mathsf{e}, (\mathsf{e}_1, \ldots, \mathsf{e}_m))} (\phi, \eta'[\mathbf{r_0} \mapsto \mathbf{a}], \mathbf{pc}'), c[\mathsf{e}_1, \ldots, \mathsf{e}_m]}\ \mathcal{A}_r$$

$$\frac{\mathbf{pc} \in \xi_{\mathrm{Op}} \quad \mathbf{pc}' = ret(\phi, \eta, \mathbf{pc}) \quad \mathsf{v} \notin range(\eta) \quad \mathsf{v} = L(\mathbf{pc}, \eta) \quad (\eta', \mathbf{a}) = mstore(\eta, heap_{\mathrm{Op}}, Mem_{\mathrm{Op}}, \mathsf{v}, 128)}{\mathbf{P} \vdash (\phi, \eta, \mathbf{pc}) \xrightarrow{\mathsf{Cr}(\mathsf{v})} (\phi, \eta'[\mathbf{r_0} \mapsto \mathbf{a}], \mathbf{pc}')}\ \text{library}$$

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

point and reflects the effect of the loop body in its exit state (computed by $exit : \xi_{\mathcal{L}} \rightarrow \mathbf{L}$). The rule also raises the event $\mathsf{loop}(\mathsf{t})$ with 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** e, we evaluate e w.r.t. the current state to get an expression e'; we then query the SMT solver for a satisfiable assignment to $\mathsf{tgt} = \mathsf{e}' \wedge \phi$, assuming that tgt does not occur in e' and ϕ. The solver returns one possible target, say 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 $\mathsf{Branch}(\gamma, \mathsf{T}_1, \mathsf{T}_2)$ with the condition γ and sub-trees T_i, or an event node $\mathsf{node}(\mathbf{pc}, \mathsf{ev})$ 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., $\mathsf{T}[\mathbf{pc}]$ 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.

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 $(\!|\mathsf{T}|\!)$ represents the compiled process, i.e. $Q^{full} = (\!|\mathsf{T}|\!)$. 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*; $LoopProc : \xi_{\mathcal{L}} \rightarrow P$ 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.

$\mathsf{T} = \mathsf{node} :: \mathsf{T}'$ Event tree

$(\!|\mathsf{Leaf} :: \mathsf{T}'|\!) \mapsto 0; (\!|\mathsf{T}'|\!)$

$(\!|\mathsf{node} :: \mathsf{T}'|\!) :=$ Events nodes

$(\!|(\mathbf{pc}, \tau) :: \mathsf{T}'|\!)$	$\mapsto$	$(\!|\mathsf{T}'|\!)$
$(\!|(\mathbf{pc}, \mathsf{Ev}(\mathsf{d}_1, \ldots, \mathsf{d}_m)) :: \mathsf{T}'|\!)$	$\mapsto$	$event(\mathsf{d}_1, \ldots, \mathsf{d}_m); (\!|\mathsf{T}'|\!)$
$(\!|(\mathbf{pc}, \mathsf{In}(\mathsf{v}, (\mathsf{e}_1, \ldots, \mathsf{e}_m))) :: \mathsf{T}'|\!)$	$\mapsto$	$in(c[\mathsf{e}_1, \ldots, \mathsf{e}_m], \mathsf{v}); (\!|\mathsf{T}'|\!)$
$(\!|(\mathbf{pc}, \mathsf{Out}(\mathsf{e}, (\mathsf{e}_1, \ldots, \mathsf{e}_m))) :: \mathsf{T}'|\!)$	$\mapsto$	$out(c[\mathsf{e}_1, \ldots, \mathsf{e}_m], \mathsf{e}); (\!|\mathsf{T}'|\!)$
$(\!|(\mathbf{pc}, \mathsf{Cr}(\mathsf{v})) :: \mathsf{T}'|\!)$	$\mapsto$	$let\ x = \mathsf{v}\ in\ (\!|\mathsf{T}'|\!)$ where x is fresh
$(\!|(\mathbf{pc}, \mathsf{Fr}(\mathsf{x}_i)) :: \mathsf{T}'|\!)$	$\mapsto$	$new\ \mathsf{x}_i;\ (\!|\mathsf{T}'|\!)$
$(\!|(\mathbf{pc}, \mathsf{loop}(\mathsf{t})) :: \mathsf{T}'|\!)$	$\mapsto$	$!^{\mathsf{t} \leq m} LoopProc(\mathbf{pc});\ (\!|\mathsf{T}'|\!)$

$(\!|(\mathbf{pc}, \mathsf{Branch}(\gamma, \mathsf{T}_1, \mathsf{T}_2))|\!) \mapsto if\ (\!|\gamma|\!)\ then\ (\!|\mathsf{T}_1|\!)\ else\ (\!|\mathsf{T}_2|\!)$

$(\!|\gamma|\!) :=$ Expressions

$(\!|\mathbf{b} \in \mathbf{Bval}|\!)$	$\mapsto$	$(\!|\mathbf{b}|\!) \in BS$
$(\!|\mathbf{x} \in \mathbf{Bvar}|\!)$	$\mapsto$	$(\!|\mathbf{x}|\!) \in Ivar$
$(\!|\gamma_1 \diamond_\mathbf{b} \gamma_2|\!)$	$\mapsto$	$(\!|\gamma_1|\!)(\!|\diamond_\mathbf{b}|\!)(\!|\gamma_2|\!)$ Binary Ops.
$(\!|\diamond_\mathbf{b}|\!)$	$\mapsto$	$\wedge$ **AND**; $\vee$ **OR**; $=$ **Equal**; $+$ **Plus**; ...
$(\!|\diamond_\mathbf{u} \gamma'|\!)$	$\mapsto$	$(\!|\diamond_\mathbf{u}|\!)(\!|\gamma'|\!)$ Unary Ops.
$(\!|\diamond_\mathbf{u}|\!)$	$\mapsto$	$\neg$ **Not**; $\bot$ otherwise
$(\!|\mathsf{f}(\mathsf{e}_1, \ldots, \mathsf{e}_m)|\!)$	$\mapsto$	$(\!|\mathsf{f}|\!)((\!|\mathsf{e}_1|\!), \ldots, (\!|\mathsf{e}_m|\!))$

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

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 $read(\mathsf{x}_1, l)$, 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).

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 in 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 $run(\mathbf{pc}, (y_1, \ldots, y_m))$ 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 $\mathsf{y}_1, \ldots, \mathsf{y}_m$ and store them in the environment η of the **BIR** program.

In the following, we use $I\{\mathbf{P}\}$ 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, $I\{\mathsf{s}_i\}$ also denotes states of the mixed semantics.

Fig. 11 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 $\mathbf{r_0}$, 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 $\mathbf{r_0}$.

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 $\sim_{H,(\!|\cdot|\!)} \subseteq S'^{\times \mathsf{s}} \times S$ between states/events of these two abstraction layers and show that it is preserved by the single-step executions. The simulation relation, $I\{\mathsf{s}_i\} \sim_{H,(\!|\cdot|\!)} s_i$, 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., $s_i.P = (\!|\mathsf{T}[\mathsf{s}_i.\mathbf{pc}]|\!)$, and (ii) the environments of the two abstractions are related through the interpretation H, i.e., for all $\mathsf{x} \in dom(I\{\mathsf{s}_i\}.\eta)$ there are $x \in dom(s_i.\eta)$ and an H s.t. $H(I\{\mathsf{s}_i\}.\eta[\mathsf{x}]) = s_i.\eta[x]$. 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 symbolic 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* $I\{\mathsf{s}_0\} = (True, \eta_0[\mathbf{RM} \mapsto \mathsf{rm}_k, \mathbf{r_k} \mapsto 0], \mathbf{pc_0})$ *be an initial symbolic*

[2] A 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.

$$\frac{\mathbf{pc} \in \mathbf{L}_{\mathcal{N}} \uplus \xi_{\mathrm{Op}} \uplus \xi_{\mathcal{R}} \uplus \xi_{\mathcal{E}} \qquad (\phi, \eta, \mathbf{pc}) \xrightarrow{a} (\phi', \eta', \mathbf{pc}')}{\mathbf{P} \vdash (\phi, \eta, \mathbf{pc}), Q^{\iota \times \mathrm{s}} \xrightarrow{a}\!\!\!\!\rightarrow_{1,H} (\phi', \eta', \mathbf{pc}'), Q^{\iota \times \mathrm{s}}} \text{ normal}$$

$$\frac{\begin{array}{c}[\![e]\!]_\eta = b \neq \bot \quad b' = truncate(b, maxlen(c)) \quad \forall j \leq m : [\![e_j]\!]_\eta = b_j \neq \bot \quad Q^{\iota \times \mathrm{s}\prime} = reduce(\{(\eta, Q)\}) \\ \exists!((\phi,\eta,\mathbf{pc}), c[\mathrm{e}_1, \ldots, \mathrm{e}_m]) \in Q^{\iota \times \mathrm{s}} \ : \ \mathbf{pc} \in \xi_{\mathcal{A}_\mathrm{r}} \wedge \forall j \leq m \ : \ H(\eta[\mathrm{e}_j]) = b_j \neq \bot \quad \mathbf{pc}' = ret(\phi, \eta, \mathbf{pc}) \\ \mathrm{e} \notin range(\eta) \quad H(\mathrm{e}) = b' \quad (\eta', \mathbf{a}) = mstore(\eta, heap_{\mathcal{A}}, Mem_{\mathcal{A}}, \mathrm{e}, 128) \quad Q^{\iota \times \mathrm{s}\prime\prime} = \{((\phi, \eta, \mathbf{pc}), c[\mathrm{e}_1, \ldots, \mathrm{e}_m])\}\end{array}}{\mathbf{P} \vdash (\eta, out(c[e_1, \ldots, e_m], e); Q), Q^{\iota \times \mathrm{s}} \xrightarrow{\mathrm{In}(\mathrm{e}, (\mathrm{e}_1, \ldots, \mathrm{e}_m))}\!\!\!\!\rightarrow_{1,H} ((\phi, \eta'[\mathbf{r_0} \mapsto \mathbf{a}], \mathbf{pc}'), c[\mathrm{e}_1, \ldots, \mathrm{e}_m]), Q^{\iota \times \mathrm{s}} \uplus Q^{\iota \times \mathrm{s}\prime} \setminus Q^{\iota \times \mathrm{s}\prime\prime}} \ ItoSB$$

$$\frac{\begin{array}{c}\mathbf{pc} \in \xi_{\mathcal{A}_\mathrm{s}} \quad \mathbf{pc}' = ret(\phi, \eta, \mathbf{pc}) \quad Q^{\iota \times \mathrm{s}\prime} = \{(\phi, \eta, \mathbf{pc}')\} \quad \mathbf{a} = \eta[\mathbf{r_0}] \quad \mathrm{e} = mload(\eta, \mathbf{a}) \quad H(\mathrm{e}) = b \neq \bot \quad b' = truncate(b, maxlen(c)) \\ \forall j \leq m : H(\eta[\mathrm{e}_j]) = b_j \neq \bot \quad \exists!(\eta, Q) \in Q^{\iota \times \mathrm{s}} : Q = in(c[e_1, \ldots, e_m], x); P \ \wedge \ \forall j \leq m : [\![e_j]\!]_\eta = b_j \neq \bot \quad Q^{\iota \times \mathrm{s}\prime\prime} = \{(\eta, Q)\}\end{array}}{\mathbf{P} \vdash ((\phi, \eta, \mathbf{pc}), c[\mathrm{e}_1, \ldots, \mathrm{e}_m]), Q^{\iota \times \mathrm{s}} \xrightarrow{\mathrm{Out}(\mathrm{e}, (\mathrm{e}_1, \ldots, \mathrm{e}_m))}\!\!\!\!\rightarrow_{1,H} (\eta[x \mapsto b'], P), Q^{\iota \times \mathrm{s}} \uplus Q^{\iota \times \mathrm{s}\prime} \setminus Q^{\iota \times \mathrm{s}\prime\prime}} \ SBtoI$$

$$\frac{\forall j \leq m : [\![y_j]\!]_\eta = b_j \neq \bot \quad (\mathrm{y}_1, \ldots, \mathrm{y}_m) \notin range(\eta_0) \quad \forall j \leq m : H(\mathrm{y}_j) = b_j \neq \bot \quad \forall j \leq m : (\eta_{j+1}, \mathbf{a_i}) = mstore(\eta_j, heap, Mem, \mathrm{y}_j, 128)}{\mathbf{P} \vdash (\eta, run(\mathbf{pc}, (y_1, \ldots, y_m))), Q^{\iota \times \mathrm{s}} \xrightarrow{}\!\!\!\!\rightarrow_{1,H} (\phi, \eta_{m+1}[\mathbf{r_0} \mapsto \mathbf{a_1}, \ldots, \mathbf{r_m} \mapsto \mathbf{a_m}], \mathbf{pc}), Q^{\iota \times \mathrm{s}}} \text{ run}$$

Figure 11: The mixed semantics of SBIR and IML shown by IML_{SB}.

state in IML_{SB} and $s_0 = (\eta_0, Q^{full})$ the corresponding initial IML state. Then, for all H: $I\{\mathrm{s}_0\} \sim_{H,\{\!|\cdot|\!\}} s_0$.

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

Lemma 7.2 (State/Event Equivalence). *Let $\mathbf{P}$ be a* **BIR** *program and I be an IML process, then, for all s_i, $I\{\mathrm{s}_i\}$, $I\{\mathrm{s}_j\}$ and H s.t. $I\{\mathrm{s}_i\} \sim_{H,\{\!|\cdot|\!\}} s_i$ and $I\{\mathrm{s}_i\} \xrightarrow{a^{\iota \times \mathrm{s}}}\!\!\!\!\rightarrow^{+}_{p,H} I\{\mathrm{s}_j\}$, there exist an H' and s_j s.t. $H \subseteq H'$, $s_i \xrightarrow{a}\!\!\!\!\rightarrow_p s_j$ and $I\{\mathrm{s}_j\} \sim_{H',\{\!|\cdot|\!\}} s_j$ and if $a \neq \bot$ then $a^{\iota \times \mathrm{s}} =_{H'} a$.*

We then show the translation's soundness by extending the simulation relation to execution traces, i.e., $\sim_{H,\{\!|\cdot|\!\},k} \subseteq \mathcal{R}^{\iota \times \mathrm{s}} \times \mathcal{R}^{\iota}$, w.r.t an upper bound[3] $k \in \mathcal{N}$ on the number of RNG steps of the execution $rng : R \to \mathcal{N}$. That is, $R^{\iota \times \mathrm{s}} \sim_{H,\{\!|\cdot|\!\},k} R$ holds, iff, $rng(R^{\iota \times \mathrm{s}}) \leq k$ and for all $I\{\mathrm{s}\}$ and $a^{\iota \times \mathrm{s}} \in R^{\iota \times \mathrm{s}}$ there exist $s, a \in R$, and H s.t. $I\{\mathrm{s}\} \sim_{H,\{\!|\cdot|\!\}} s$ and $a^{\iota \times \mathrm{s}} =_H a$.

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. The results can be extended to multiple **BIR** programs, as presented in [61, Sec. A.7].

Theorem 7.3 (Trace Inclusion). *Let $\mathbf{P}$ 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 symbolic execution traces $R^{\iota \times \mathrm{s}} \in \mathcal{R}^{\iota \times \mathrm{s}}(I\{\mathbf{P}\}, \eta_0[\mathbf{RM} \mapsto \mathrm{rm}_\mathrm{k}, \mathbf{r_k} \mapsto 0])$ s.t. $rng(R^{\iota \times \mathrm{s}}) \leq k$, there are an IML trace $R \in \mathcal{R}^{\iota}(I\{(\!|\mathbf{P}|\!)\})$ and an H s.t. $R^{\iota \times \mathrm{s}} \sim_{H,\{\!|\cdot|\!\},k} R$.*

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.

The reader may find the proof of lemmas 7.1 and 7.2 in [61, Sec. A.4]. □

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 showed 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 $IML_{\mathbf{B}}$, that allows the **BIR** program to communicate with the same IML attacker at the IML_{SB} level. The $IML_{\mathbf{B}}$ execution semantics is presented in [61, Sec. A.3]—the rules are similar and have the same meaning as those defined for IML_{SB}.

Our proof strategy to show the behavioral equivalence of $IML_{\mathbf{B}}$ 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 $IML_{\mathbf{B}}$ in IML_{SB}.

We show the state/event equivalence by extending the simulation relation of Property 1 to a relation $\sim_H \subseteq S^{\iota \times \mathbf{b}} \times S^{\iota \times \mathrm{s}}$ between $IML_{\mathbf{B}}$ and IML_{SB}. The relation $I\{\mathbf{s_i}\} \sim_H I\{\mathrm{s}_i\}$ checks that $I\{\mathrm{s}_i\}.\mathbf{pc} = I\{\mathbf{s_i}\}.\mathbf{pc}$, and for all $\mathbf{x} \in dom(I\{\mathbf{s_i}\}.\eta)$ there exist $\mathrm{x} \in dom(I\{\mathrm{s}_i\}.\eta)$ and an interpretation H s.t. $H(I\{\mathrm{s}_i\}.\eta[\mathrm{x}]) = I\{\mathbf{s_i}\}.\eta[\mathbf{x}]$. The first

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

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* $\mathbf{P}$*, an IML process* I *and any upper bound* $k \in \mathcal{N}$ *on the number of RNG steps, let* $I\{\mathbf{s_0}\} = (\eta_0[\mathbf{RM} \mapsto rm_k, \mathbf{r_k} \mapsto 0], \mathbf{pc_0})$ *be an initial* **BIR** *state in* $IML_{\mathbf{B}}$ *and* $I\{\mathsf{s}_0\} = (True, \eta_0[\mathbf{RM} \mapsto \mathsf{rm}_k, \mathbf{r_k} \mapsto 0], \mathbf{pc_0})$ *be the corresponding initial state in* IML_{SB}*. Then,* $I\{\mathbf{s_0}\} \sim_H I\{\mathsf{s}_0\}$ *for all* H*.*

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

Lemma 7.5 (State/Event Equivalence). *Let* $\mathbf{P}$ *be a* **BIR** *program and* I *be an IML process, then, for all* $I\{\mathsf{s}_i\}$, $I\{\mathbf{s_i}\}$, $I\{\mathbf{s_j}\}$ *and* H *s.t.* $I\{\mathbf{s_i}\} \sim_H I\{\mathsf{s}_i\}$ *and* $I\{\mathbf{s_i}\} \xrightarrow{a^{\iota\times\mathbf{b}}} {}^{+}_{p} I\{\mathbf{s_j}\}$*, there exist an* H' *and* $I\{\mathsf{s}_j\}$ *s.t.* $H \subseteq H'$, $I\{\mathsf{s}_i\} \xrightarrow{a^{\iota\times\mathsf{s}}} {}^{+}_{p,H'} I\{\mathsf{s}_j\}$, $I\{\mathbf{s_j}\} \sim_{H'} I\{\mathsf{s}_j\}$ *and* $a^{\iota\times\mathbf{b}} =_{H'} a^{\iota\times\mathsf{s}}$*.*

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

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

Proof. Thm. 7.6 shows that for all $IML_{\mathbf{B}}$ 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.

The interested reader can find the proof of lemmas 7.4 and 7.5 in [61, Sec. A.5]. □

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.

$$\mathbf{Auth} = \{t \in \psi \mid \forall i \in \mathcal{N} : t[i] = Accept_S(PK_S, PK_C) \implies (\exists j \in \mathcal{N} : j < i \ \wedge \ t[j] = Accept_C(PK_S, PK_C)) \ \}$$

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 $\mathcal{R}^\alpha \downarrow_{events}$, 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 $\mathbf{P}$, 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 $I\{\mathbf{P}\}$ w.r.t. ψ is $\mathbf{insec}(I\{\mathbf{P}\}, n, k, \psi) = 2^{-n\cdot k} \cdot \sum_{rm_k \in BS_n^k} \sum_{t^{\iota\times\mathbf{b}} \in T^{\iota\times\mathbf{b}}_{rm_k} \cap \psi_\neg} pr(t^{\iota\times\mathbf{b}})$ where $T^{\iota\times\mathbf{b}}_{rm_k} = T^{\iota\times\mathbf{b}}(I\{\mathbf{P}\}, \eta_0[\mathbf{RM} \mapsto rm_k, \mathbf{r_k} \mapsto 0])$.

After translating $\mathbf{P}$ into $(\!|\mathbf{P}|\!)$, 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 $\mathbf{insec}(I, n, \psi) = \sum_{t \in T(I,n) \cap \psi_\neg} pr(t)$ where $pr(s_0 \xrightarrow{\ } p_1 s_1 \cdots s_{n-1} \xrightarrow{\ } p_n s_n) = \prod_{1 \leq i \leq n} p_i$.

Note that definitions 8.1 and 8.2 coincide with $IML_{\mathbf{B}}$ 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 $IML_{\mathbf{B}}$ 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* $\mathbf{P}$*, 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* $T^{\iota\times\mathbf{b}}(I\{\mathbf{P}\}, \eta_0[\mathbf{RM} \mapsto rm_k, \mathbf{r_k} \mapsto 0])$*, we get that*

$$\mathbf{insec}(I\{\mathbf{P}\}, n, k, \psi) \leq \mathbf{insec}(I\{(\!|\mathbf{P}|\!)\}, n, \psi).$$

Via [2, Thm. 4.3,Thm. 5.2] we obtain a bound for $\mathbf{insec}(I\{(\!|\mathbf{P}|\!)\}, n, \psi)$ 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 $\mathbf{insec}(I\{\mathbf{P}\}, n, k, \psi)$ is negligible.

We present the proof of Thm. 8.3 in [61, Sec. A.6]. Based on definitions 8.1 and 8.2, we calculate the probability distribution in both IML and $IML_{\mathbf{B}}$. 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 [61, 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. To evaluate CryptoBap we first verified the case studies of [3, 4]. 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 [62]. 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. Simple XOR implements a protocol in which the one-time pad includes both protocol parties. 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 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 CryptoBap—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 CryptoBap to analyze the binary of cryptographic protocols. CryptoBap enables sound verification of authentication and weak secrecy for protocols' machine code using the 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,

Protocol		ARM Loc	Verified Code Size	Feasible Path	Infeasible Path	IML Loc	CryptoVerif (CV) Loc	ProVerif (PV) Loc	Time (Second) IML	CV	PV	Verified in	Primitives
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.

we have evaluated the automation and completeness of CryptoBap via case studies while ensuring soundness through proofs.

The current implementation of CryptoBap 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 CryptoBap'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 CryptoBap 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] Mihhail 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] Mihhail 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 Maffeis. 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) *(SFCS '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 Mojzis. 2018. The SSH Library. https://github.com/janmojzis/tinyssh.

[61] Faezeh Nasrabadi, Robert Künnemann, and Hamed Nemati. 2023. CryptoBap: A Binary Analysis Platform for Cryptographic Protocols. *arXiv preprint arXiv* (2023). http://arxiv.org/abs/2308.14450

[62] 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.

[63] 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).

[64] Marco Patrignani. 2020. Why should anyone use colours? or, syntax highlighting beyond code snippets. *arXiv preprint arXiv:2001.11334* (2020).

[65] 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.

[66] 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.

[67] Huzaifa Sidhpurwala. 2019. Security flaws caused by compiler optimizations. https://www.redhat.com/en/blog/security-flaws-caused-compiler-optimizations

[68] 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.

[69] Jan Strejcek and Marek Trtík. 2012. Abstracting path conditions. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*. 155–165.

[70] Xi Wang, Nickolai 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* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 260–275.

[71] 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.

[72] 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*.

[73] 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.

[74] 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

[75] 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.

[76] 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.