View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Edinburgh Research Explorer

THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Extracting and verifying cryptographic models from C protocol code by symbolic execution

Citation for published version:
Aizatulin, M, Gordon, AD & Jürjens, J 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. ACM, New York, NY, USA, pp. 331-340. DOI: 10.1145/2046707.2046745

Digital Object Identifier (DOI):
10.1145/2046707.2046745

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the 18th ACM conference on Computer and communications security

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

OPEN ACCESS

Download date: 28. Apr. 2017

arXiv:1107.1017v1 [cs.CR] 6 Jul 2011

Extracting and Verifying Cryptographic Models from C Protocol Code by Symbolic Execution

Mihhail Aizatulin
The Open University

Andrew D. Gordon
Microsoft Research

Jan Jürjens
TU Dortmund & Fraunhofer ISST

ABSTRACT

Consider the problem of verifying security properties of a cryptographic protocol coded in C. We propose an automatic solution that needs neither a pre-existing protocol description nor manual annotation of source code. First, symbolically execute the C program to obtain symbolic descriptions for the network messages sent by the protocol. Second, apply algebraic rewriting to obtain a process calculus description. Third, run an existing protocol analyser (ProVerif) to prove security properties or find attacks. We formalise our algorithm and appeal to existing results for ProVerif to establish computational soundness under suitable circumstances. We analyse only a single execution path, so our results are limited to protocols with no significant branching. The results in this paper provide the first computationally sound verification of weak secrecy and authentication for (single execution paths of) C code.

1. INTRODUCTION

Recent years have seen great progress in formal verification of cryptographic protocols, as illustrated by powerful tools like ProVerif [13], CryptoVerif [12] or AVISPA [3]. There remains, however, a large gap between what we verify (formal descriptions of protocols, say, in the pi calculus) and what we rely on (protocol implementations, often in low-level languages like C). The need to start the verification from C code has been recognised before and implemented in tools like CSur [26] and ASPIER [18], but the methods proposed there are still rather limited. Consider, for example, the small piece of C code in fig. 1 that checks whether a message received from the network matches a message authentication code. Intuitively, if the key is honestly chosen and kept secret from the attacker then with overwhelming probability the event will be triggered only if another honest participant (with access to the key) generated the message. Unfortunately, previous approaches cannot prove this property: the analysis of CSur is too coarse to deal with authentication properties like this and ASPIER cannot directly deal with code manipulating memory through pointers. Furthermore the previous works do not offer a definition of security directly for C code, i.e. they do not formally state what it means for a C program to satisfy a security property, which makes it difficult to evaluate their overall soundness. The goal of our work is to improve upon this situation by giving a formal definition of security straight for C code and proposing a method that can verify secrecy and authentication for typical memory-manipulating implementations like the one in fig. 1 in a fully automatic and scalable manner, without relying on a pre-existing protocol specification.

```
void * key; size_t keylen;
readenv(”k”, &key, &keylen);
size_t len;
read(&len, sizeof(len));
if(len > 1000) exit();
void * buf = malloc(len + 2 * MAC_LEN);
read(buf, len);
mac(buf, len, key, keylen, buf + len);
read(buf + len + MAC_LEN, MAC_LEN);
if(memcmp(buf + len,
          buf + len + MAC_LEN,
          MAC_LEN) == 0)
  event(”accept”, buf, len);
```

---

in(x_1); **in**(x_2); **if** $x_2 = \mathit{mac}(k, x_1)$ **then event** $\mathit{accept}(x_1)$

Figure 1: An example C fragment together with the extracted model.

Our method proceeds by extracting a high-level model from the C code that can then be verified using existing tools (we use ProVerif in our work). Currently we restrict our analysis to code in which all network outputs happen on a single execution path, but otherwise we do not require use of any specific programming style, with the aim of applying our methods to legacy implementations. In particular, we do not assume memory safety, but instead explicitly verify it during model extraction. The method still assumes that the cryptographic primitives such as encryption or hashing are implemented correctly—verification of these is difficult even when done manually [2].

The two main contributions of our work are:

- formal definition of security properties for source code;
- an algorithm that computes a high-level model of the protocol implemented by a C program.

We implement and evaluate the algorithm as well as give a proof of its soundness with respect to our security definition. Our definition of security for source code is given by linking the semantics of a programming language, expressed as a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

transition system, to a computational security definition in the spirit of [15, 25, 41]. We allow an arbitrary number of sessions. We restrict our definition to trace properties (such as weak secrecy or authentication), but do not consider observational equivalence (for strong secrecy, say).

Due to the complexity of the C language we give the formal semantics for a simple assembler-like language into which C code can be easily compiled, as in other symbolic execution approaches such as [19]. The soundness of this step can be obtained by using well-known methods, as outlined in section 3.

Our model-extraction algorithm produces a model in an intermediate language without memory access or destructive updates, while still preserving our security definition. The algorithm is based on symbolic execution [30] of the C program, using symbolic expressions to over-approximate the sets of values that may be stored in memory during concrete execution. The main difference from existing symbolic execution algorithms (such as [17] or [24]) is that our variables represent bitstrings of potentially unknown length, whereas in previous algorithms a single variable corresponds to a single byte.

We show how the extracted models can be further simplified into the form understood by ProVerif. We apply the computational soundness result from [4] to obtain conditions where the symbolic security definition checked by ProVerif corresponds to our computational security definition. Combined with the security-preserving property of the model extraction algorithm this provides a computationally sound verification of weak secrecy and authentication for C.

Outline of our Method. The verification proceeds in several steps, as outlined in fig. 2. The method takes as input:

- the C implementations of the protocol participants, containing calls to a special function `event` as in fig. 1,
- an environment process (in the modelling language) which spawns the participants, distributes keys, etc.,
- symbolic models of cryptographic functions used by the implementation,
- a property that event traces in the execution are supposed to satisfy with overwhelming probability.

We start by compiling the program down to a simple stack-based instruction language (CVM) using CIL [34] to parse and simplify the C input. The syntax and semantics of CVM are presented in section 2 and the translation from C to CVM is informally described in section 3.

In the next step we symbolically execute CVM programs to eliminate memory accesses and destructive updates, thus obtaining an equivalent program in an intermediate model language (IML)—a version of the applied pi calculus extended with bitstring manipulation primitives. For each allocated memory area the symbolic execution stores an expression describing how the contents of the memory area have been computed. For instance a certain memory area might be associated with an expression $hmac(01|x, k)$, where x is known to originate from the network, k is known to be an environment variable, and $|$ denotes concatenation. The symbolic execution does not enter the functions that implement the cryptographic primitives, it uses the provided symbolic models instead. These models thus form the trusted base of the verification. An example of the symbolic execution output is shown at the bottom of fig. 1. We define the syntax and semantics of IML in section 4 and describe the symbolic execution in section 6.

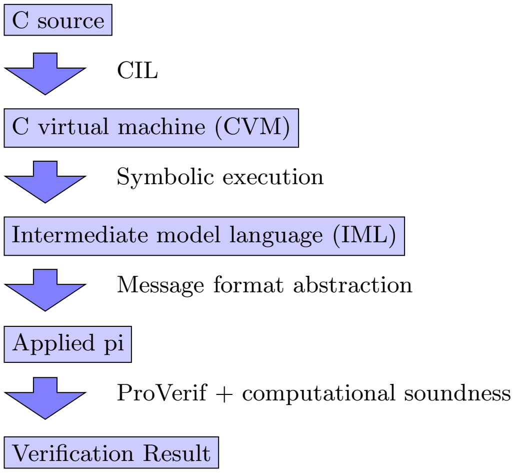
C source
CIL
C virtual machine (CVM)
Symbolic execution
Intermediate model language (IML)
Message format abstraction
Applied pi
ProVerif + computational soundness
Verification Result

Figure 2: An outline of the method

Our definition of security for source code is given in section 5. The definition is generic in that it does not assume a particular programming language. We simply require that the semantics of a language is given as a set of transitions of a certain form, and define a computational execution of the resulting transition system in the presence of an attacker and the corresponding notion of security. This allows one to apply the same security definition to protocols expressed both in the low-level implementation language and in the high-level model-description language, and to formulate a correspondence between the two.

Given that the transition systems generated by different languages are required to be of the same form, we can mix them in the same execution. This allows us to use CVM to specify a single executing participant, but at the same time use IML to describe an environment process that spawns multiple participants and allows them to interact. In particular, CVM need not be concerned with concurrency, thus making symbolic execution easier. Given an environment process P_E with n holes, we write $P_E[P_1, \dots, P_n]$ for a process where the ith hole is filled with P_i, which can be either a CVM or an IML process. The soundness result for symbolic execution (theorem 1) states that if $P_1, \dots, P_n$ are CVM processes and $\tilde{P}_1, \dots, \tilde{P}_n$ are IML models resulting from their symbolic execution then for any environment process P_E the security of $P_E[\tilde{P}_1, \dots, \tilde{P}_n]$ with respect to a trace property ρ relates to the security of $P_E[P_1, \dots, P_n]$ with respect to ρ.

To verify the security of an IML process, we replace its bitstring-manipulating expressions by applications of constructor and destructor functions, thus obtaining a process in the applied pi-calculus (the version proposed in [14] and augmented with events). We can then apply a computational soundness result, such as the one from [4], to specify conditions under which such a substitution is computationally sound: if the resulting pi calculus process is secure in a symbolic model (as can be checked by ProVerif) then it is asymptotically secure with respect to our computational notion of security. The correctness of translation from IML to pi is captured by theorem 2 and the computational soundness for resulting pi processes is captured by theorem 3. The verification of IML (and these two theorems in particular) is described in section 7.

Theoretical and Practical Evaluation. Theorems 1 to 3 establish the correctness of our approach. In a nutshell, their significance is as follows: given implementations $P_1, \dots, P_n$ of protocol participants in CVM, which are automatically obtained from the corresponding C code, and an IML process P_E that describes an execution environment, if $P_1, \dots, P_n$ are successfully symbolically executed with resulting models $\tilde{P}_1, \dots, \tilde{P}_n$, the IML process $P_E[\tilde{P}_1, \dots, \tilde{P}_n]$ is successfully translated to a pi process P_π, and ProVerif successfully verifies P_π against a trace property ρ then $P_1, \dots, P_n$ form a secure protocol implementation with respect to the environment P_E and property ρ.

We are aiming to apply our method to large legacy code bases like OpenSSL. As a step towards this goal we evaluated it on a range of protocol implementations, including recent code for smart electricity meters [37]. We were able to find bugs in preexisting implementations or to verify them without having to modify the code. Section 8 provides details.

The current restriction of analysis to a single execution path may seem prohibitive at first sight. In fact, a great majority of protocols (such as those in the extensive SPORE repository [36]) follow a fixed narration of messages between participants, where any deviation from the expected message leads to termination. For such protocols, our method allows us to capture and analyse the fixed narration directly from the C code. In the future we plan to extend the analysis to more sophisticated control flow.

Related Work. We mention particularly relevant works here and provide a broader survey in section 9. One of the first attempts at cryptographic verification of C code is contained in [26], where a C program is used to generate a set of Horn clauses that are then solved using a theorem prover. The method is implemented in the tool CSur. We improve upon CSur in two ways in particular.

First, we have an explicit attacker model with a standard computational attacker. The attacker in CSur is essentially symbolic—it is allowed to apply cryptographic operations, but cannot perform any arithmetic computations.

Second, we handle authentication properties in addition to secrecy properties. Adding authentication to CSur would be non-trivial, due to a rather coarse over-approximation of C code. For instance, the order of instructions in CSur is ignored, and writing a single byte into an array with unknown length is treated the same as overwriting the whole array. Authentication, however, crucially depends on the order of events in the execution trace as well as making sure that the authenticity of a whole message is preserved and not only of a single byte of it.

ASPIER [18] uses model checking to verify implementations of cryptographic protocols. The model checking operates on a protocol description language, which is rather more abstract than C; for instance, it does not contain pointers and cannot express variable message lengths. The translation from C to the protocol language is not described in the paper. Our method applies directly to C code with pointers, so that we expect it to provide much greater automation.

Corin and Manzano [19] report an extension of the KLEE test-generation tool [17] that allows KLEE to be applied to cryptographic protocol implementations (but not to extract models, as in our work). They do not extend the class of properties that KLEE is able to test for; in particular, testing for trace properties is not yet supported. Similarly to our work, KLEE is based on symbolic execution; the main difference is that [19] treats every byte in a memory buffer separately and thus only supports buffers of fixed length.

An appendix includes proofs for all the results stated in this paper.

2. C VIRTUAL MACHINE (CVM)

This section describes our low-level source language CVM (C Virtual Machine). The language is simple enough to formalise, while at the same time the operations of CVM are closely aligned with the operations performed by C programs, so that it is easy to translate from C to CVM. We shall describe such a translation informally in section 3.

The model of execution of CVM is a stack-based machine with random memory access. All operations with values are performed on the stack, and values can be loaded from memory and stored back to memory. The language contains primitive operations that are necessary for implementing security protocols: reading values from the network or the execution environment, choosing random values, writing values to the network and signalling events. The only kind of conditional that CVM supports is a testing operation that checks a boolean condition and aborts execution immediately if it is not satisfied.

The fact that CVM permits no looping or recursion in the program allows us to inline all function calls, so that we do not need to add a call operation to the language itself. For simplicity of presentation we omit some aspects of the C language that are not essential for describing the approach, such as global variable initialisation and structures. We also restrict program variables to all be of the same size: for the rest of the paper we choose a fixed but arbitrary $N \in \mathbb{N}$ and assume $\texttt{sizeof}(v) = N$ for all program variables v. Our implementation does not have these restrictions and deals with the full C language.

Let $BS = \{0,1\}^*$ be the set of finite bitstrings with the empty bitstring denoted by ε. For a bitstring b let $|b|$ be the length of b in bits. Let Var be a countably infinite set of variables. We write $f \colon X \rightharpoonup Y$ to denote a partial function and let $\mathrm{dom}(f) \subseteq X$ be the set of x for which $f(x)$ is defined. We write $f(x) = \bot$ when f is not defined on x and use the notation $f\{x \mapsto a\}$ to update functions.

Let **Ops** be a finite set of operation symbols such that each $op \in \mathbf{Ops}$ has an associated arity $\mathrm{ar}(op)$ and an efficiently computable partial function $A_{op} \colon BS^{\mathrm{ar}(op)} \rightharpoonup BS$. The set **Ops** is meant to contain both the primitive operations of the language (such as the arithmetic or comparison operators of C) and the cryptographic primitives that are used by the implementation. The security definitions of this paper (given later) assume an arbitrary security parameter. Since real-life cryptographic protocols are typically designed and implemented for a fixed value of the security parameter, for the rest of the paper we let $k_0 \in \mathbb{N}$ be the security parameter with respect to which the operations in **Ops** are chosen.

A CVM program is simply a sequence of instructions, as shown in fig. 3. To define the semantics of CVM we choose two functions that relate bitstrings to integer values, $\mathrm{val} \colon BS \to \mathbb{N}$ and $\mathrm{bs} \colon \mathbb{N} \to BS$ and require that for $n < 2^N$ the value $\mathrm{bs}(n)$ is a bitstring of length N such that $\mathrm{val}(\mathrm{bs}(n)) = n$. We allow bs to have arbitrary behaviour for larger numbers. The functions val and bs encapsulate architecture-specific details of integer representation such as

$b \in BS$, $v \in Var$, $op \in \mathbf{Ops}$

$src ::=$ `read` | `rnd`	input source
$dest ::=$ `write` | `event`	output destination
$instr ::=$	instruction
`Const` b	constant value
`Ref` v	pointer to variable
`Malloc`	pointer to fresh memory
`Load`	load from memory
`In` v src	input
`Env` v	environment variable
`Apply` op	operation
`Out` $dest$	output
`Test`	test a condition
`Store`	write to memory
$P \in CVM ::= \{instr\,;\}^*$	program

Figure 3: The syntax of CVM.

the endianness. Even though these functions capture an unsigned interpretation of bitstrings, we only use them when accessing memory cells and otherwise place no restriction on how the bitstrings are interpreted by the program operations. For instance, the set **Ops** can contain both a signed and an unsigned arithmetic and comparison operators. Bitstring representations of integer constants shall be written as $i1$, $i20$, etc, for instance, $i10 = \mathrm{bs}(10)$.

We let $Addr = \{1, \ldots, 2^N - 1\}$ be the set of valid memory addresses. The reason we exclude 0 is to allow the length of the memory to be represented in N bits. The semantic configurations of CVM are of the form $(\mathcal{A}^c, \mathcal{M}^c, \mathcal{S}^c, P)$, where

- $\mathcal{M}^c \colon Addr \rightharpoonup \{0, 1\}$ is a partial function that represents concrete memory and is undefined for uninitialised cells,
- $\mathcal{A}^c \subseteq Addr$ is the set of allocated memory addresses,
- $\mathcal{S}^c$ is a list of bitstrings representing the execution stack,
- $P \in CVM$ is the executing program.

Semantic transitions are of the form $(\eta, s) \xrightarrow{l} (\eta', s')$, where s and s' are semantic configurations, η and η' are environments (mappings from variables to bitstrings) and l is a protocol action such as reading or writing values from the attacker or a random number generator, or raising events. The formal semantics of CVM is given in appendix C, in this section we give an informal overview. Before the program is executed, each referenced variable v is allocated an address $\mathrm{addr}(v)$ in $\mathcal{M}^c$ such that all allocations are non-overlapping. If the program contains too many variables to fit in memory, the execution does not proceed. Next, the instructions in the program are executed one by one as described below. For $a, b \in \mathbb{N}$ we define $\{a\}_b = \{a, \ldots, a + b - 1\}$.

- `Const` b places b on the stack.
- `Ref` v places $\mathrm{bs}(\mathrm{addr}(v))$ on the stack.
- `Malloc` takes a value s from the stack, reads a value p from the attacker, and if the range $\{\mathrm{val}(p)\}_{\mathrm{val}(s)}$ does not contain allocated cells, it becomes allocated and the value p is placed on the stack. Thus the attacker gets to choose the beginning of the allocated memory area.
- `Load` takes values l and p from the stack. In case $\{\mathrm{val}(p)\}_{\mathrm{val}(l)}$ is a completely initialised range in memory, the contents of that range are placed on the stack. In case some of the bits are not initialised, the value for those bits is read from the attacker.
- `In` v `read` or `In` v `rnd` takes a value l from the stack. `In` v `read` reads a value of length $\mathrm{val}(l)$ from the attacker and `In` v `rnd` requests a random value of length $\mathrm{val}(l)$. The resulting value b is then placed on the stack. The environment η is extended by the binding $v \mapsto b$.
- `Env` v places $\eta(v)$ and $\mathrm{bs}(|\eta(v)|)$ on the stack.
- `Apply` op with $\mathrm{ar}(op) = n$ applies A_{op} to n values on the stack, replacing them by the result.
- `Out write` sends the top of the stack to the attacker and `Out event` raises an event with the top of the stack as payload. Events with multiple arguments can be represented using a suitable bitstring pairing operation. Both commands remove the top of the stack.
- `Test` takes the top of the stack and checks whether it is $i1$. If yes, the execution proceeds, otherwise it stops.
- `Store` takes values p and b from the stack and writes b into memory at position starting with $\mathrm{val}(p)$.

The execution of a program can get stuck if rule conditions are violated, for instance, when the program runs out of memory or attempts to write to uninitialised memory. All these situations would likely result in a crash in a real system. Our work is not focused on preventing crashes, but rather on analysing the sequences of events that occur before the program terminates (either normally or abnormally). Thus we leave crashes implicit in the semantics. An exception is the instruction `Load`: reading uninitialised memory is unlikely to result in a crash in reality, instead it can silently return any value. We model this behaviour explicitly in the semantics.

3. FROM C TO CVM

We describe how to translate from C to CVM programs. We start with aspects of the translation that are particular to our approach, after which we illustrate the translation by applying it to the example program in fig. 1.

Proving correctness of C compilation is not the main focus of our work, so we trust compilation for now. To prove correctness formally one would need to show that a CVM translation simulates the original C program; an appropriate notion of simulation is defined in appendix B and is used to prove soundness of other verification steps. We believe that work on proving correctness of the CompCert compiler [31] can be reused in this context.

We require that the C program contains no form of looping or function call cycles and that all actions of the program (either network outputs or events) happen in the same path (called *main path* in the following). We then prune all other paths by replacing if-statements on the main path by test statements: a statement `if(cond) t_block else f_block` is replaced by `test(cond); t_block` in case the main path continues in the `t_block`, and by `test(!cond); f_block` otherwise. The test statements are then compiled to CVM `Test` instructions. The main path can be easily identified by static analysis; for now we simply identify the path to be compiled by observing an execution of the program.

As mentioned in the introduction, we do not verify the

```
void mac_proxy(void * buf, size_t buflen,
               void * key, size_t keylen,
               void * mac){
  load_buf(buf, buflen);
  load_buf(key, keylen);
  apply(”mac”, 2);
  store_buf(mac);
}

int memcmp_proxy(void * a, void * b,
                 size_t len){
  int ret;
  load_buf(a, len);
  load_buf(b, len);
  apply(”cmp”, 2);
  store_buf(&ret);
  return ret;
}
```

Figure 4: Examples of proxy functions.

source code of cryptographic functions, but instead trust that they implement the cryptographic algorithms correctly. Similarly, we would not be able to translate the source code of functions like `memcmp` into CVM directly, as these functions contain loops. Thus for the purpose of CVM translation we provide an abstraction for these functions. We do so by writing what we call a *proxy function* `f_proxy` for each function `f` that needs to be abstracted. Whenever a call to `f` is encountered during the translation, it is replaced by the call to `f_proxy`. The proxy functions form the trusted base of the verification.

Examples of proxy functions are shown in fig. 4. The functions `load_buf`, `apply` and `store_buf` are treated specially by the translation. For instance, assuming an architecture with $N = 32$, a call `load_buf(buf, len)` directly generates the sequence of instructions:

```
Ref buf; Const i32; Load;
Ref len; Const i32; Load; Load;
```

Similarly we provide proxies for all other special functions in the example program, such as `readenv`, `read`, `write` or `event`. The proxies essentially list the CVM instructions that need to be generated.

Appendix H.2 shows more examples of proxy functions. Appendix A shows the CVM translation of our example C program in fig. 1.

4. INTERMEDIATE MODEL LANGUAGE

This section presents the intermediate model language (IML) that we use both to express the models extracted from CVM programs and to describe the environment in which the protocol participants execute. IML borrows most of its structure from the pi calculus [1, 14]. In addition it has access both to the set **Ops** of operations used by CVM programs and to primitive operations on bitstrings: concatenation, substring extraction, and computing lengths of bitstrings. Unlike CVM, IML does not access memory or perform destructive updates.

The syntax of IML is presented in fig. 5. In contrast to the standard pi calculus we do not include channel names, but implicitly use a single public channel instead. This corresponds to our security model in which all communication happens through the attacker. The nonce sampling operation $(\nu x[e])$ takes an expression as a parameter that specifies the length of the nonce to be sampled—this is necessary in the computational setting in order to obtain a probability distribution. We introduce a special abbreviation for programs that choose randomness of length equal to the security parameter k_0 introduced in section 2: let $(\tilde{\nu} x)$; P stand for $(\nu \tilde{x}[k_0])$; **let** $x = \mathit{nonce}(\tilde{x})$ **in** P, where $\mathit{nonce} \in$ **Ops**. Using *nonce* allows us to have tagged nonces, which will be necessary to link to the pi calculus semantics from [4].

$b \in \mathit{BS}$, $x \in \mathit{Var}$, $\mathit{op} \in \mathbf{Ops}$

$e \in \mathit{IExp} ::=$		expression
	b	concrete bitstring
	x	variable
	$\mathit{op}(e_1, \ldots, e_n)$	computation
	$e_1 \vert e_2$	concatenation
	$e\{e_o, e_l\}$	substring extraction
	$\mathrm{len}(e)$	length
$P, Q \in \mathit{IML} ::=$		process
	0	nil
	$!P$	replication
	$P \vert Q$	parallel composition
	$(\nu x[e]);\ P$	randomness
	in(x); P	input
	out(e); P	output
	event(e); P	event
	if e **then** P [**else** Q]	conditional
	let $x = e$ **in** P [**else** Q]	evaluation

Figure 5: The syntax of IML.

$$
\begin{aligned}
&[\![b]\!] = b, \text{ for } b \in \mathit{BS},\\
&[\![x]\!] = \bot, \text{ for } x \in \mathit{Var},\\
&[\![\mathit{op}(e_1, \ldots, e_n)]\!] = A_{\mathit{op}}([\![e_1]\!], \ldots, [\![e_n]\!]),\\
&[\![e_1|e_2]\!] = [\![e_1]\!]|[\![e_2]\!],\\
&[\![e\{e_o, e_l\}]\!] = \mathrm{sub}([\![e]\!], \mathrm{val}([\![e_o]\!]), \mathrm{val}([\![e_l]\!])),\\
&[\![\mathrm{len}(e)]\!] = \mathrm{bs}(|[\![e]\!]|).
\end{aligned}
$$

Figure 6: The evaluation of IML expressions, whereby $\bot$ propagates.

For a bitstring b let $b[i]$ be the ith bit of b counting from 0. The concatenation of two bitstrings b_1 and b_2 is written as $b_1|b_2$.

Just as for CVM, the semantics of IML is parameterised by functions bs and val. The semantics of expressions is given by the partial function $[\![\cdot]\!] \colon \mathit{IExp} \rightharpoonup \mathit{BS}$ described in fig. 6. The partial function sub: $\mathit{BS} \times \mathbb{N} \times \mathbb{N} \rightharpoonup \mathit{BS}$ extracts a substring of a given bitstring such that $\mathrm{sub}(b, o, l)$ is the substring of b starting at offset o of length l:

$$
\mathrm{sub}(b, o, l) = \begin{cases} b[o] \ldots b[o + l - 1] & \text{if } o + l \le |b|,\\ \bot & \text{otherwise.} \end{cases}
$$

For a *valuation* $\eta \colon \mathit{Var} \rightharpoonup \mathit{BS}$ we denote with $[\![e]\!]_\eta$ the result of substituting all variables v in e by $\eta(v)$ (if defined) and then applying $[\![\cdot]\!]$.

The formal semantics of IML is mostly straightforward and is shown in detail in appendix D.

5. SECURITY OF PROTOCOLS

This section gives an informal overview of our security definition. The complete definition is given in appendix B.

To define security for protocols implemented by CVM and IML programs we need to specify what a protocol is and give a mapping from programs to protocols. The notion of a protocol is formally captured by a *protocol transition system (PTS)*, which describes how processes evolve and interact with the attacker. A PTS is a set of transitions of the form $(\eta, s) \xrightarrow{l} \{(\eta_1, s_1), \dots, (\eta_n, s_n)\}$, where η and η_i are environments (modelled as valuations), s and s_i are semantic configurations of the underlying programming language, and l is an action label. Actions can include reading values from the attacker, generating random values, sending values to the attacker, or raising events. We call a pair (η, s) an *executing process.* Multiple processes on the right hand side capture replication.

The semantics of CVM and IML are given in terms of the PTS that are implemented by programs. For a CVM program P we denote with $[\![P]\!]_C$ the PTS that is implemented by P. Similarly, for an IML process P the corresponding PTS is denoted by $[\![P]\!]_I$.

Given a PTS T and a probabilistic machine E (an attacker) we can execute T in the presence of E. The state of the executing protocol is essentially a multiset of executing processes. The attacker repeatedly chooses a process from the multiset which is then allowed to perform an action according to T. The result of the execution is a sequence of raised events. For a resource bound $t \in \mathbb{N}$ we denote with $\mathrm{Events}(T, E, t)$ the sequence of events raised during the first t steps of the execution. We shall be interested in the probability that this sequence of events belongs to a certain “safe” set. This is formally captured by the following definition:

Definition 1 (Protocol security) We define a *trace property* as a polynomially decidable prefix-closed set of event sequences. For a PTS T, a trace property ρ and a resource bound $t \in \mathbb{N}$ let $\mathrm{insec}(T, \rho, t)$ be the probability

$$\sup \{\Pr[\mathrm{Events}(T, E, t) \notin \rho] \mid E \text{ attacker}, |E| \leq t\},$$

where $|E|$ measures the size of the description of the attacker. □

Intuitively $\mathrm{insec}(T, \rho, t)$ measures the success probability of the most successful attack against T and property ρ when both the execution time of the attack and the size of the attacker code are bounded by t.

Since the semantics of CVM and IML are in the same formalism, we may combine the sets of semantic rules and obtain semantics $[\![\cdot]\!]_{CI}$ for mixed programs, where a CVM program can be a subprocess of a larger IML process. We add an additional syntactic form $[]_i$ (a hole) with $i \in \mathbb{N}$ and no reductions to IML. For an IML process P_E with n holes and CVM or IML processes $P_1, \dots, P_n$ we write $P_E[P_1, \dots, P_n]$ to denote process P_E where each hole $[]_i$ is replaced by P_i. The semantics of the resulting process P'_E, denoted with $[\![P'_E]\!]_{CI}$, is defined in appendix D.

Being able to embed a CVM program within an IML process is useful for modelling. As an example, let P_1 be the CVM program resulting from the translation of the C code in fig. 1 and let P_2 be a description of another participant of the protocol, in either CVM or IML. Then we might be interested in the security of the following process:

$$P_E[P_1, P_2] = !((\tilde{\nu}\ k);\ ((!P_1)|(!P_2))).$$

$v \in Var$, $i \in \mathbb{N}$		
$pb \in PBase ::=$		pointer base
	stack v	stack pointer to variable v
	heap i	heap pointer with id i
$e \in SExp ::=$		symbolic expression
	$\mathrm{ptr}(pb, e)$	pointer
	...	same as *IExp* in fig. 5

Figure 7: Symbolic expressions.

A trace property ρ of interest might be, for instance, “Each event of the form $accept(x)$ is preceded by an event of the form $request(x)$”, where $request$ is an event possibly raised in P_2. The goal is to obtain a statement about probability $\mathrm{insec}([\![P_E[P_1, P_2]]\!]_{CI}, \rho, t)$ for various t. The next section shows how we can relate the security of $P_E[P_1, P_2]$ to the security of $P_E[\tilde{P}_1, P_2]$, where IML process $\tilde{P}_1$ is a model of the CVM process P_1, extracted by symbolic execution.

6. CVM TO IML: SYMBOLIC EXECUTION

We describe how to automatically extract an IML model from a CVM program while preserving security properties. The key idea is to execute a CVM program in a symbolic semantics, where, instead of concrete bitstrings, memory locations contain IML expressions representing the set of all possible concrete values at a given execution point.

To track the values used as pointers during CVM execution, we extend IML expressions with an additional construct, resulting in the class of *symbolic expressions* shown in fig. 7. An expression of the form $\mathrm{ptr}(pb, e_o)$ represents a pointer into the memory location identified by the *pointer base* pb with an offset e_o relative to the beginning of the location. We require that $e_o \in IExp$, so that pointer offsets do not contain pointers themselves. Pointer bases are of two kinds: a base of the form stack v represents a pointer to the program variable v and a base of the form heap i represents the result of a `Malloc`.

Symbolic execution makes certain assumptions about the arithmetic operations that are available in **Ops**. We assume that programs use operators for bitwise addition and subtraction (with overflow) that we shall write as $+_b$ and $-_b$. We also make use of addition and subtraction without overflow—the addition operator (written as $+_{\mathbb{N}}$) is expected to widen its result as necessary and the negation operator (written as $-_{\mathbb{N}}$) returns $\bot$ instead of a negative result. We assume that **Ops** contains comparison operators $=$, $\leq$, and $<$ such that $A_=(a, b)$ returns $i1$ if $\mathrm{val}(a) = \mathrm{val}(b)$ and $i0$ otherwise, similarly for the other operators. This way $\leq$ and $<$ capture unsigned comparisons on bitstring values. We assume **Ops** contains logical connectives $\neg$ and $\vee$ that interpret $i0$ as false value and $i1$ as true value. These operators may or may not be the ones used by the program itself.

To evaluate symbolic expressions concretely, we need concrete values for pointer bases as well as concrete values for variables. Given an *extended valuation* $\eta\colon Var \cup PBase \rightharpoonup BS$, we extend the function $[\![\cdot]\!]_\eta$ from fig. 6 by the rule:

$$[\![\mathrm{ptr}(pb, e_o)]\!]_\eta = \eta(pb) +_b [\![e_o]\!]_\eta.$$

When applying arithmetic operations to pointers, we need to make sure that the operation is applied to the pointer offset and the base is kept intact. This behaviour is encoded

$$\frac{}{(\texttt{Init},\, P) \to (\Sigma_{op},\, \{\text{stack }\ v \mapsto \text{bs}(N) \mid v \in \text{var}(P)\},\, \{\text{stack }\ v \mapsto \varepsilon \mid v \in \text{var}(P)\},\, [],\, P)} \tag{S-Init}$$

$$\frac{}{(\Sigma,\, \mathcal{A}^s,\, \mathcal{M}^s,\, \mathcal{S}^s,\, \texttt{Const}\ b;\, P) \to (\Sigma,\, \mathcal{A}^s,\, \mathcal{M}^s,\, b :: \mathcal{S}^s,\, P)} \tag{S-Const}$$

$$\frac{}{(\Sigma,\, \mathcal{A}^s,\, \mathcal{M}^s,\, \mathcal{S}^s,\, \texttt{Ref}\ v;\, P) \to (\Sigma,\, \mathcal{A}^s,\, \mathcal{M}^s,\, \text{ptr}(\text{stack }\ v,\, i0) :: \mathcal{S}^s,\, P)} \tag{S-Ref}$$

$$\frac{e_l \in \mathit{IExp} \quad i \in \mathbb{N} \text{ minimal s.t. } pb = \text{heap }\ i \notin \text{dom}(\mathcal{M}^s)}{(\Sigma,\, \mathcal{A}^s,\, \mathcal{M}^s,\, e_l :: \mathcal{S}^s,\, \texttt{Malloc};\, P) \to (\Sigma,\, \mathcal{A}^s\{pb \mapsto e_l\},\, \mathcal{M}^s\{pb \mapsto \varepsilon\},\, \text{ptr}(pb, i0) :: \mathcal{S}^s,\, P)} \tag{S-Malloc}$$

$$\frac{pb \in \text{dom}(\mathcal{M}^s) \quad e = \text{simplify}_\Sigma(\mathcal{M}^s(pb)\{e_o, e_l\}) \quad \Sigma \vdash (e_o +_{\mathbb{N}} e_l \leq \text{getLen}(\mathcal{M}^s(pb)))}{(\Sigma,\, \mathcal{A}^s,\, \mathcal{M}^s,\, e_l :: \text{ptr}(pb, e_o) :: \mathcal{S}^s,\, \texttt{Load};\, P) \to (\Sigma,\, \mathcal{A}^s,\, \mathcal{M}^s,\, e :: \mathcal{S}^s,\, P)} \tag{S-Load}$$

$$\frac{e_l \in \mathit{IExp} \quad l = (\text{if } \mathit{src} = \texttt{read} \text{ then } \mathbf{in}(v);\ \text{else } (\nu v[e_l]);)}{(\Sigma,\, \mathcal{A}^s,\, \mathcal{M}^s,\, e_l :: \mathcal{S}^s,\, \texttt{In}\ v\ \mathit{src};\, P) \xrightarrow{l} (\Sigma \cup \{\text{len}(v) = e_l\},\, \mathcal{A}^s,\, \mathcal{M}^s,\, v :: \mathcal{S}^s,\, P)} \tag{S-In}$$

$$\frac{}{(\Sigma,\, \mathcal{A}^s,\, \mathcal{M}^s,\, \mathcal{S}^s,\, \texttt{Env}\ v;\, P) \to (\Sigma,\, \mathcal{A}^s,\, \mathcal{M}^s,\, \text{len}(v) :: v :: \mathcal{S}^s,\, P)} \tag{S-Env}$$

$$\frac{e = \text{apply}(op, e_1, \ldots, e_n) \neq \bot}{(\Sigma,\, \mathcal{A}^s,\, \mathcal{M}^s,\, e_1 :: \ldots :: e_n :: \mathcal{S}^s,\, \texttt{Apply}\ op;\, P) \to (\Sigma,\, \mathcal{A}^s,\, \mathcal{M}^s,\, \text{len}(e) :: e :: \mathcal{S}^s,\, P)} \tag{S-Apply}$$

$$\frac{e \in \mathit{IExp} \quad l = (\text{if } \mathit{dest} = \texttt{write} \text{ then } \mathbf{out}(e);\ \text{else } \mathbf{event}(e);)}{(\Sigma,\, \mathcal{A}^s,\, \mathcal{M}^s,\, e :: \mathcal{S}^s,\, \texttt{Out}\ \mathit{dest};\, P) \xrightarrow{l} (\Sigma,\, \mathcal{A}^s,\, \mathcal{M}^s,\, \mathcal{S}^s,\, P)} \tag{S-Out}$$

$$\frac{e \in \mathit{IExp}}{(\Sigma,\, \mathcal{A}^s,\, \mathcal{M}^s,\, e :: \mathcal{S}^s,\, \texttt{Test};\, P) \xrightarrow{\textbf{if } e \textbf{ then}} (\Sigma \cup \{e\},\, \mathcal{A}^s,\, \mathcal{M}^s,\, \mathcal{S}^s,\, P)} \tag{S-Test}$$

$$\frac{\begin{array}{c} e_h = \mathcal{M}^s(pb) \neq \bot \quad e_s = \mathcal{A}^s(pb) \neq \bot \quad e_{lh} = \text{getLen}(e_h) \quad e_l = \text{getLen}(e) \\ \text{either } \Sigma \vdash (e_o +_{\mathbb{N}} e_l < e_{lh}) \text{ and } e'_h = \text{simplify}_\Sigma(e_h\{i0, e_o\}|e|e_h\{e_o +_{\mathbb{N}} e_l, e_{lh} -_{\mathbb{N}} (e_o +_{\mathbb{N}} e_l)\}) \\ \text{or } \Sigma \vdash (e_o +_{\mathbb{N}} e_l \geq e_{lh}) \wedge (e_o \leq e_{lh}) \wedge (e_o +_{\mathbb{N}} e_l \leq e_s) \text{ and } e'_h = \text{simplify}_\Sigma(e_h\{i0, e_o\}|e) \end{array}}{(\Sigma,\, \mathcal{A}^s,\, \mathcal{M}^s,\, \text{ptr}(pb, e_o) :: e :: \mathcal{S}^s,\, \texttt{Store};\, P) \to (\Sigma,\, \mathcal{A}^s,\, \mathcal{M}^s\{pb \mapsto e'_h\},\, \mathcal{S}^s,\, P)} \tag{S-Store}$$

Figure 8: The symbolic execution of CVM.

by the function apply, defined as follows:

$$\begin{aligned}
&\text{apply}(+_b, \text{ptr}(pb, e_o), e) = \text{ptr}(pb, e_o +_b e), \\
&\qquad \text{for } e \in \mathit{IExp}, \\
&\text{apply}(-_b, \text{ptr}(pb, e_o), \text{ptr}(pb, e'_o)) = e_o -_b e'_o, \\
&\text{apply}(op, e_1, \ldots, e_n) = op(e_1, \ldots, e_n), \\
&\qquad \text{for } e_1, \ldots, e_n \in \mathit{IExp}, \\
&\text{apply}(...) = \bot, \text{ otherwise.}
\end{aligned}$$

As well as tracking the expressions stored in memory, we also track logical facts discovered during symbolic execution. To record these facts, we use symbolic expressions themselves, interpreted as logical formulas with $=$, $\leq$, and $<$ as relations and $\neg$ and $\vee$ as connectives. We allow quantifiers in formulas, with straightforward interpretation. Given a set Σ of formulas and a formula ϕ we write $\Sigma \vdash \phi$ iff for each Σ-consistent valuation η (that is, a valuation such that $[\![\psi]\!]_\eta = i1$ for all $\psi \in \Sigma$) we also have $[\![\phi]\!]_\eta = i1$.

To check the entailment relation, our implementation relies on the SMT solver *Yices* [21], by replacing unsupported operations, such as string concatenation or substring extraction, with uninterpreted functions. This works well for our purpose—the conditions that we need to check during the symbolic execution are purely arithmetic and are supported by Yices' theory.

The function getLen returns for each symbolic expression an expression representing its length:

$$\begin{aligned}
&\text{getLen}(\text{ptr}(\ldots)) = \text{bs}(N), \\
&\text{getLen}(\text{len}(\ldots)) = \text{bs}(N), \\
&\text{getLen}(b) = \text{bs}(|b|), \text{ for } b \in \mathit{BS}, \\
&\text{getLen}(x) = \text{len}(x), \text{ for } x \in \mathit{Var}, \\
&\text{getLen}(op(e_1, \ldots, e_n)) = \text{len}(op(e_1, \ldots, e_n)), \\
&\text{getLen}(e_1|e_2) = \text{getLen}(e_1) +_{\mathbb{N}} \text{getLen}(e_2), \\
&\text{getLen}(e\{e_o, e_l\}) = e_l.
\end{aligned}$$

We assume that the knowledge about the return lengths of operation applications is encoded in a fact set Σ_{op}. As an example, Σ_{op} might contain the facts:

$$\begin{aligned}
&\forall x, y, a\colon\ \text{len}(x) = a \wedge \text{len}(y) = a \Rightarrow \text{len}(x +_b y) = a, \\
&\forall x\colon\ \text{len}(\mathit{sha}1(x)) = i20.
\end{aligned}$$

We assume that Σ_{op} is consistent: $\emptyset \vdash \phi$ for all $\phi \in \Sigma_{op}$.

The transformations prescribed by the symbolic semantic rules would quickly lead to very large expressions. Thus the symbolic execution is parametrised by a simplification function simplify that is allowed to make use of the collected fact set Σ. We demand that the simplification function is sound in the following sense: for each fact set Σ, expression e and a Σ-consistent valuation η we have

$$[\![e]\!]_\eta \neq \bot \Longrightarrow [\![\text{simplify}_\Sigma(e)]\!]_\eta = [\![e]\!]_\eta.$$

The simplifications employed in our algorithm are described in appendix E.

Line no.	C line	symbolic memory updates	new facts	generated IML line
1.	`readenv("k", &key, &keylen);`	stack $key \Rightarrow \mathrm{ptr}(\mathrm{heap}\ 1,\ i0)$ heap $1 \Rightarrow k$ stack $keylen \Rightarrow \mathrm{len}(k)$		
2.	`read(&len, sizeof(len));`	stack $len \Rightarrow l$	$\mathrm{len}(l) = iN$	$\mathbf{in}(l)$
3.	`if(len > 1000) exit();`		$\neg(l > i1000)$	
4.	`void * buf = malloc(len + 2 * MAC_LEN);`	stack $buf \Rightarrow \mathrm{ptr}(\mathrm{heap}\ 2,\ i0)$ heap $2 \Rightarrow \varepsilon$		
5.	`read(buf, len);`	heap $2 \Rightarrow x_1$	$\mathrm{len}(x_1) = l$	$\mathbf{in}(x_1)$
6.	`mac(buf, len, key, keylen, buf + len);`	heap $2 \Rightarrow x_1 \mid mac(k, x_1)$		
7.	`read(buf + len + MAC_LEN, MAC_LEN);`	heap $2 \Rightarrow x_1 \mid mac(k, x_1) \mid x_2$	$\mathrm{len}(x_2) = i20$	$\mathbf{in}(x_2)$
8.	`if(memcmp(...) == 0)`			**if** $mac(k, x_1) = x_2$ **then**
9.	`event("accept", buf, len);`			**event** $accept(x_1)$

Figure 9: Symbolic execution of the example in fig. 1.

The algorithm for symbolic execution is determined by the set of semantic rules presented in fig. 8. The initial semantic configuration has the form $(\mathtt{Init},\ P)$ with the executing program $P \in \mathit{CVM}$. The other semantic configurations have the form $(\Sigma,\ \mathcal{A}^s,\ \mathcal{M}^s,\ \mathcal{S}^s,\ P)$, where

- $\Sigma \subseteq \mathit{SExp}$ is a set of formulas (the path condition),
- $\mathcal{A}^s \colon \mathit{PBase} \rightharpoonup \mathit{SExp}$ is the symbolic allocation table that for each memory location stores its allocated size,
- $\mathcal{M}^s \colon \mathit{PBase} \rightharpoonup \mathit{SExp}$ is the symbolic memory. We require that $\mathrm{dom}(\mathcal{M}^s) = \mathrm{dom}(\mathcal{A}^s)$,
- $\mathcal{S}^s$ is a list of symbolic expressions representing the execution stack,
- $P \in \mathit{CVM}$ is the executing program.

The symbolic execution rules essentially mimic the rules of the concrete execution. The crucial rules are (S-Load) and (S-Store) that reflect the effect of storing and loading memory values on the symbolic level. The rule (S-Load) is quite simple—it tries to deduce from Σ that the extraction is performed from a defined memory range, after which it represents the result of the extraction using an IML range expression. The rule (S-Store) distinguishes between two cases depending on how the expression e to be stored is aligned with the expression e_h that is already present in memory. If e needs to be stored completely within the bounds of e_h then we replace the contents of the memory location by $e_h\{\ldots\}|e|e_h\{\ldots\}$ where the first and the second range expression represent the pieces of e_h that are not covered by e. In case e needs to be stored past the end of e_h, the new expression is of the form $e_h\{\ldots\}|e$. The rule still requires that the beginning of e is positioned before the end of e_h, and hence it is impossible to write in the middle of an uninitialised memory location. This is for simplicity of presentation—the rule used in our implementation does not have this limitation (it creates an explicit "undefined" expression in these cases).

Since all semantic rules are deterministic there is only one symbolic execution trace. Some semantic transition rules are labelled with parts of IML syntax. The sequence of these labels produces an IML process that simulates the behaviour of the original CVM program. Formally, for a CVM program P, let L be the symbolic execution trace starting from the state $(\mathtt{Init},\ P)$. If L ends in a state with an empty program, let $\lambda_1, \ldots, \lambda_n$ be the sequence of labels of L and set $[\![P]\!]_S = \lambda_1 \ldots \lambda_n 0 \in \mathit{IML}$, otherwise set $[\![P]\!]_S = \bot$.

We shall say that a polynomial is *fixed* iff it is independent of the arbitrary values assumed in this paper, such as N or the properties of the set **Ops**. Our main result relates the security of P to the security of $[\![P]\!]_S$.

Theorem 1 (Symbolic Execution is Sound) *There exists a fixed polynomial p such that if $P_1, \ldots, P_n$ are CVM processes and for each i $\tilde{P}_i := [\![P_i]\!]_S \neq \bot$ then for any IML process P_E, any trace property ρ, and resource bound $t \in \mathbb{N}$:*

$$\begin{aligned}&\mathrm{insec}([\![P_E[P_1, \ldots, P_n]]\!]_{CI}, \rho, t)\\ &\quad \leq \mathrm{insec}([\![P_E[\tilde{P}_1, \ldots, \tilde{P}_n]]\!]_I, \rho, p(t)).\end{aligned}$$

□

The condition that p is fixed is important—otherwise p could be large enough to give the attacker the time to enumerate all the $2^{2^N - 1}$ memory configurations. For practical use the actual shape of p can be recovered from the proof of the theorem given in appendix F.

Fig. 9 illustrates our method by showing how the symbolic execution proceeds for our example in fig. 1. For each line of the C program we show updates to the symbolic memory, the set of new facts, and the generated IML code if any. In our example `MAC_LEN` is assumed to be 20 and N is equal to `sizeof(size_t)`. The variables l, x_1, and x_2 are arbitrary fresh variables chosen during the translation from C to CVM (see appendix A). Below we mention details for some particularly interesting steps (numbers correspond to line numbers in fig. 9).

1. The call to `readenv` redirects to a proxy function that generates CVM instructions for retrieving the environment variable k and storing it in memory.
4. A new empty memory location is created and the pointer to it is stored in `buf`. We make an entry in the allocation table $\mathcal{A}^s$ with the length of the new memory location $(l +_b i2 * i20)$.
5. We check that the stored value fits within the allocated memory area, that is, $l \leq l +_b i2 * i20$. This is in general not true due to possibility of integer overflow, but in this case succeeds due to the condition $\neg(l > i1000)$ recorded before (assuming that the maximum integer value $2^N - 1$ is much larger than 1000). Similar checks are performed for all subsequent writes to memory.
7. The memory update is performed through an intermediate pointer value of the form $\mathrm{ptr}(\mathrm{heap}\ 2,\ l +_b i20)$. The set of collected facts is enough to deduce that this pointer points exactly at the end of $x_1|mac(k, x_1)$.
8. The proxy function for `memcmp` extracts values $e_1 = e\{l, i20\}$ and $e_2 = e\{l +_b i20, i20\}$, where e is the contents of memory at heap 2, and puts $cmp(e_1, e_2)$ on the stack. With the facts collected so far e_1 simplifies to $mac(k, x_1)$ and e_2 simplifies to x_2. With some special comprehension for the meaning of cmp we generate IML **if** $e_1 = e_2$ **then**.

7. VERIFICATION OF IML

The symbolic model extracted in fig. 9 does not contain any bitstring operations, so it can readily be given to ProVerif for verification. In general this is not the case and some further simplifications are required. In a nutshell, the simplifications are based on the observation that the bitstring expressions (concatenation and substring extraction) are meant to represent pairing and projection operations, so we can replace them by new symbolic operations that behave as pairing constructs in ProVerif. We then check that the expressions indeed satisfy the algebraic properties expected of such operations.

We outline the main results regarding the translation to ProVerif. Appendix G contains the details. The pi calculus used by ProVerif can be described as a subset of IML from which the bitstring operations have been removed. Unlike CVM and IML, the semantics of pi is given with respect to an arbitrary security parameter: we write $[\![P]\!]_\pi^k$ for the semantics of a pi process P with respect to the parameter $k \in \mathbb{N}$. In contrast, we consider IML as executing with respect to a fixed security parameter $k_0 \in \mathbb{N}$. For an IML process P we specify conditions under which it is translatable to a pi process $\tilde{P}$.

Theorem 2 (Soundness of the translation) *There exists a fixed polynomial p such that for any $P \in$ IML translatable to a pi process $\tilde{P}$, any trace property ρ and resource bound $t \in \mathbb{N}$:* $\mathrm{insec}([\![P]\!]_I, \rho, t) \leq \mathrm{insec}([\![\tilde{P}]\!]_\pi^{k_0}, \rho, p(t))$. □

Backes et al. [4] provide an example of a set of operations $\mathbf{Ops}^S$ and a set of *soundness conditions* restricting their implementations that are sufficient for establishing computational soundness. The set $\mathbf{Ops}^S$ contains a public key encryption operation that is required to be IND-CCA secure. The soundness result is established for the class of the so-called *key-safe* processes that always use fresh randomness for encryption and key generation, only use honestly generated decryption keys and never send decryption keys around.

Theorem 3 (Computational soundness) *Let P be a pi process using only operations in $\mathbf{Ops}^S$ such that the soundness conditions are satisfied. If P is key-safe and symbolically secure with respect to a trace property ρ (as checked by ProVerif) then for every polynomial p the following function is negligible in k:* $\mathrm{insec}([\![P]\!]_\pi^k, \rho, p(k))$. □

Overall, theorems 1 to 3 can be interpreted as follows: let $P_1, \ldots, P_n$ be implementations of protocol participants in CVM and let P_E be an IML process that describes an execution environment. Assume that $P_1, \ldots, P_n$ are successfully symbolically executed with resulting models $\tilde{P}_1, \ldots, \tilde{P}_n$, the IML process $P_E[\tilde{P}_1, \ldots, \tilde{P}_n]$ is successfully translated to a pi process P_π, and ProVerif successfully verifies P_π against a trace property ρ. Then we know by theorem 3 that P_π is a pi protocol model that is (asymptotically) secure with respect to ρ. By theorems 1 and 2 we know that $P_1, \ldots, P_n$ form a secure implementation of the protocol described by P_π for the security parameter k_0.

8. IMPLEMENTATION & EXPERIMENTS

We have implemented our approach and successfully tested it on several examples. Our implementation performs the conversion from C to CVM at runtime—the C program is instrumented using CIL so that it outputs its own CVM representation when run. This allows us to identify and compile the main path of the protocol easily. Apart from information about the path taken we do not use any runtime information and we plan to make the analysis fully static in future. The idea of instrumenting a program to emit a low-level set of instructions for symbolic execution at runtime as well as some initial implementation code were borrowed from the CREST symbolic execution tool [16].

	C LOC	IML LOC	outcome	result type	time
simple mac	∼ 250	12	verified	symbolic	4s
RPC	∼ 600	35	verified	symbolic	5s
NSL	∼ 450	40	verified	computat.	5s
CSur	∼ 600	20	flaw: fig. 11	—	5s
minexplib	∼ 1000	51	flaw: fig. 12	—	15s

Figure 10: Summary of analysed implementations.

```
read(conn_fd, temp, 128);
// BN_hex2bn expects zero-terminated string
temp[128] = 0;
BN_hex2bn(&cipher_2, temp);
// decrypt and parse cipher_2
// to obtain message fields
```

Figure 11: A flaw in the CSur example: input may be too short.

Currently we omit certain memory safety checks and assume that there are no integer overflows. This allows us to use the more efficient theory of mathematical integers in Yices, but we are planning to move to exact bitvector treatment in future.

The implementation comprises about 4600 lines of OCaml code. The symbolic proxies for over 80 of the cryptographic functions in the OpenSSL library comprise further 2000 lines of C code.

Fig. 10 shows a list of protocol implementations on which we tested our method. Some of the verified programs did not satisfy the conditions of computational soundness (mostly because they use cryptographic primitives other than public key encryption and signatures supported by the result that we rely on [4]), so we list the verification type as “symbolic”.

The “simple mac” is an implementation of a protocol similar to the example in fig. 1. RPC is an implementation of the remote procedure call protocol in [8] that authenticates a server response to a client using a message authentication code. It was written by a colleague without being intended for verification using our method, but we were still able to verify it without any further modifications to the code.

The NSL example is an implementation of the Needham-Schroeder-Lowe protocol written by us to obtain a fully computationally sound verification result. The implementation is designed to satisfy the soundness conditions listed in appendix G (modulo the assumption that the encryption used is indeed IND-CCA). Masking the second participant’s identity check triggers Lowe’s attack [32] as expected. Appendix H shows the source code and the extracted models.

The CSur example is the code analysed in a predecessor paper on C verification [26]. It is an implementation of a protocol similar to Needham-Schroeder-Lowe. During our verification attempt we discovered a flaw, shown in fig. 11: the received message in buffer `temp` is being converted to a `BIGNUM` structure `cipher_2` without checking that enough bytes were received. Later a `BIGNUM` structure derived from

```
unsigned char session_key[256 / 8];
...
// Use the 4 first bytes as a pad
// to encrypt the reading
encrypted_reading =
  ((unsigned int) *session_key) ^ *reading;
```

Figure 12: A flaw in the minexplib code: only one byte of the pad is used.

`cipher_2` is converted to a bitstring without checking that the length of the bitstring is sufficient to fill the message buffer. In both cases the code does not make sure that the information in memory actually comes from the network, which makes it impossible to prove authentication properties. The CSur example has been verified in [26], but only for secrecy, and secrecy is not affected by the flaw we discovered. The code reinterprets network messages as C structures (an unsafe practise due to architecture dependence), which is not yet supported by our analysis and so we were not able to verify a fixed version of it.

The minexplib example is an implementation of a privacy-friendly protocol for smart electricity meters [37] developed at Microsoft Research. The model that we obtained uncovered a flaw shown in fig. 12: incorrect use of pointer dereferencing results in three bytes of each four-byte reading being sent unencrypted. We found two further flaws: one could lead to contents of uninitialised memory being sent on the network, the other resulted in 0 being sent (and accepted) in place of the actual number of readings. All flaws have been acknowledged and fixed. An F# implementation of the protocol has been previously verified [38], which highlights the fact that C implementations can be tricky and can easily introduce new bugs, even for correctly specified and proven protocols. The protocol uses low-level cryptographic operations such as XOR and modular exponentiation. In general it is impossible to model XOR symbolically [40], so we could not use ProVerif to verify the protocol, but we are investigating the use of CryptoVerif for this purpose.

9. RELATED WORK

[26] presents the tool Csur for verifying C implementations of crypto-protocols by transforming them into a decidable subset of first-order logic. It only supports secrecy properties and relies on a Dolev-Yao attacker model. It was applied to a self-made implementation of the Needham-Schroeder protocol. [18] presents the verification framework ASPIER using predicate abstraction and model-checking which operates on a protocol description language where certain C concepts such as pointers and variable message lengths are manually abstracted away. In comparison, our method applies directly to C code including pointers and thus requires less manual effort. [28] presents the C API "DYC" which can be used to generate executable protocol implementations of Dolev-Yao type cryptographic protocol messages. By generating constraints from those messages, one can use a constraint solver to search for attacks. The approach presents significant limitations on the C code. [39] reports on the Pistachio approach which verifies the conformance of an implementation with a specification of the communication protocol. It does not directly support the verification of security properties. To prepare the ground for symbolic analysis of cryptographic protocol implementations, [19] reports an extension of the KLEE symbolic execution tool. Cryptographic primitives can be treated as symbolic functions whose execution analysis is avoided. A security analysis is not yet supported. The main difference from our work is that [19] treats every byte in a memory buffer separately and thus only supports buffers of fixed length. [20] shows how to adapt a general-purpose verifier to security verification of C code. This approach does not have our restriction to non-branching code, on the other hand, it requires the code to be annotated (with about one line of annotation per line of code) and works in the symbolic model, requiring the pairing and projection operations to be properly encapsulated.

There is also work on verifying implementations of security protocols in other high-level languages. These do not compare directly to the work presented here, since our aim is in particular to be able to deal with the intricacies of a low-level language like C. The tools FS2PV [10] and FS2CV translate F# to the process calculi which can be verified by the tools ProVerif [11] and CryptoVerif [12] versus symbolic and computational models, respectively. They have been applied to an implementation of TLS [9]. The refinement-type checker F7 [8] verifies security properties of F# programs versus a Dolev-Yao attacker. Under certain conditions, this has been shown to be provably computationally sound [6, 23]. [33] reports on a formal verification of a reference implementation of the TPM's authorization and encrypted transport session protocols in F#. It also provides a translator from programs into the functional fragment of F# into executable C code. [6] gives results on computational soundness of symbolic analysis of programs in the concurrent lambda calculus RCF. [5] reports on a type system for verifying crypto-protocol implementations in RCF. With respect to Java, [29] presents an approach which provides a Dolev-Yao formalization in FOL starting from the program's control-flow graph, which can then be verified for security properties with automated theorem provers for FOL (such as SPASS). [35] provides an approach for translating Java implementations into formal models in the LySa process calculus in order to perform a security verification. [27] presents an application of the ESC/Java2 static verifier to check conformance of JavaCard applications to protocol models. [22] describes verification of cryptographic primitives implemented in a functional language Cryptol. CertiCrypt [7] is a framework for writing machine-checked cryptographic proofs.

10. CONCLUSION

We presented methods and tools for the automated verification of cryptographic security properties of protocol implementations in C. More specifically, we provided a computationally sound verification of weak secrecy and authentication for (single execution paths of) C code. Despite the limitation of analysing single execution paths, the method often suffices to prove security of authentication protocols, many of which are non-branching. We plan to extend the analysis to more sophisticated control flow.

In future, we aim to provide better feedback in case verification fails. In our case this is rather easy to do as symbolic execution proceeds line by line. If a condition check fails for a certain symbolic expression, it is straightforward to print out a computation tree for the expression together with source code locations in which every node of the tree was computed. We plan to implement this feature in the future, although so far we found that manual inspection of the symbolic execution trace lets us identify problems easily.

Acknowledgements. Discussions with Bruno Blanchet, François Dupressoir, Bashar Nuseibeh, and Dominique Unruh were useful. We also thank George Danezis, François Dupressoir, and Jean Goubault-Larrecq for giving us access to the code of minexplib, RPC, and CSur, respectively. Ricardo Corin and François Dupressoir commented on a draft.

11. REFERENCES

[1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *ACM POPL*, pages 104–115, 2001.

[2] J. B. Almeida, M. Barbosa, J. S. Pinto, and B. Vieira. Deductive verification of cryptographic software. In *NASA Formal Methods Symposium 2009*, 2009.

[3] A. Armando, D. A. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuéllar, P. H. Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285. Springer, 2005.

[4] M. Backes, D. Hofheinz, and D. Unruh. CoSP: A general framework for computational soundness proofs. In *ACM CCS 2009*, pages 66–78, November 2009. Preprint on IACR ePrint 2009/080.

[5] M. Backes, C. Hritcu, and M. Maffei. Union and intersection types for secure protocol implementations. In *Theory of Security and Applications (TOSCA'11)*, 2011.

[6] M. Backes, M. Maffei, and D. Unruh. Computationally sound verification of source code. In *CCS*, 2010.

[7] G. Barthe, B. Grégoire, and S. Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 90–101, New York, NY, USA, 2009. ACM.

[8] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *CSF '08: Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, pages 17–32. IEEE Computer Society, 2008.

[9] K. Bhargavan, C. Fournet, R. Corin, and E. Zălinescu. Cryptographically verified implementations for TLS. Alexandria, VA, Oct. 2008. ACM.

[10] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, pages 139–152. IEEE Computer Society, 2006.

[11] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *CSFW*, pages 82–96. IEEE Computer Society, 2001.

[12] B. Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security and Privacy*, pages 140–154. IEEE Computer Society, 2006.

[13] B. Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, 2009.

[14] B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75(1):3–51, Feb.–Mar. 2008.

[15] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Comput.*, 13(4):850–864, 1984.

[16] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 443–446. IEEE Computer Society, 2008.

[17] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, San Diego, CA, Dec. 2008.

[18] S. Chaki and A. Datta. Aspier: An automated framework for verifying security protocol implementations. In *Computer Security Foundations Workshop*, pages 172–185, 2009.

[19] R. Corin and F. A. Manzano. Efficient symbolic execution for analysing cryptographic protocol implementations. In *International Symposium on Engineering Secure Software and Systems (ESSOS'11)*, LNCS. Springer, 2011.

[20] F. Dupressoir, A. D. Gordon, J. Jürjens, and D. A. Naumann. Guiding a general-purpose C verifier to prove cryptographic protocols. In *24th IEEE Computer Security Foundations Symposium*, 2011.

[21] B. Dutertre and L. D. Moura. The Yices SMT Solver. Technical report, 2006.

[22] L. Erkök, M. Carlsson, and A. Wick. Hardware/software co-verification of cryptographic algorithms using cryptol. In *FMCAD*, 2009.

[23] C. Fournet. Cryptographic soundness for program verification by typing. Unpublished draft, 2011.

[24] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. The Internet Society, 2008.

[25] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28:270–299, 1984.

[26] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 363–379. Springer, 2005.

[27] E. Hubbers, M. Oostdijk, and E. Poll. Implementing a formally verifiable security protocol in Java card. In *Security in Pervasive Computing, First International Conference, Revised Papers*, volume 2802 of *Lecture Notes in Computer Science*, pages 213–226. Springer, 2004.

[28] A. Jeffrey and R. Ley-Wild. Dynamic model checking

of C cryptographic protocol implementations. In *Proceedings of Workshop on Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis*, 2006.

[29] J. Jürjens. Security analysis of crypto-based Java programs using automated theorem provers. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 167–176. IEEE Computer Society, 2006.

[30] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[31] X. Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43:363–446, December 2009.

[32] G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Inf. Process. Lett.*, 56:131–133, November 1995.

[33] A. Mukhamedov, A. D. Gordon, and M. Ryan. Towards a verified reference implementation of the trusted platform module. In *17th International Workshop on Security Protocols (2009)*, LNCS. Springer, 2011. To appear.

[34] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, 2002. Springer-Verlag.

[35] N. O'Shea. Using Elyjah to analyse Java implementations of cryptographic protocols. In *FCS-ARSPA-WITS'08*, pages 211–223, 2008.

[36] Project EVA. Security protocols open repository, 2007. `http://www.lsv.ens-cachan.fr/spore/`.

[37] A. Rial and G. Danezis. Privacy-friendly smart metering. Technical Report MSR–TR–2010–150, 2010.

[38] N. Swamy, J. Chen, C. Fournet, K. Bharagavan, and J. Yang. Security programming with refinement types and mobile proofs. Technical Report MSR–TR–2010–149, 2010.

[39] O. Udrea, C. Lumezanu, and J. S. Foster. Rule-Based static analysis of network protocol implementations. *IN PROCEEDINGS OF THE 15TH USENIX SECURITY SYMPOSIUM*, pages 193–208, 2006.

[40] D. Unruh. The impossibility of computationally sound XOR, July 2010. Preprint on IACR ePrint 2010/389.

[41] A. C. Yao. Theory and application of trapdoor functions. In *SFCS '82: Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, pages 80–91. IEEE Computer Society, 1982.

APPENDIX

A. C TO CVM—EXAMPLE

Fig. 13 shows the CVM translation of the example program from fig. 1. We use abbreviations for some useful instruction sequences: we write `Clear` as an abbreviation for `Store dummy` that stores a value into an otherwise unused dummy variable. The effect of `Clear` is thus to remove one value from the stack. Often we do not need the length of the result that the instructions `Env` and `Apply` place on the stack, so we introduce the versions `Env'` and `Apply'` that discard the length: `Env'` v is an abbreviation for `Env` v`; Clear` and `Apply'` v is an abbreviation for `Apply` v`; Clear`. The abbreviation `Varsize` is supposed to load the variable width N onto the stack, for instance, on an architecture with $N = 32$ the meaning of `Varsize` would be `Const i32`. For convenience we write operation arguments of `Apply` together with their arities.

During the translation we arbitrarily choose fresh variables l, x_1, and x_2 for use in the `In` operations.

```
//void * key; size_t keylen;
//readenv(”k”, &key, &keylen);
Env k; Ref keylen; Store;
Ref keylen; Varsize; Load; Malloc;
Ref key; Store;
Ref key; Varsize; Load; Store;
//size_t len;
//read(&len, sizeof(len));
Varsize; In l read; Ref len; Store;
// if(len > 1000) exit();
Const i1000; Ref len; Varsize; Load;
Apply' >/2; Apply' ¬/1; Test;
//void * buf = malloc(len + 2 * 20);
Ref len; Varsize; Load;
Const i2; Const i20;
Apply' */2; Apply' +/2;
Malloc; Ref buf; Store;
//read(buf, len);
Ref len; Varsize; Load; In x1 read;
Ref buf; Varsize; Load; Store;
//mac(buf, len, key, keylen, buf + len);
Ref buf; Varsize; Load;
Ref len; Varsize; Load; Load;
Ref key; Varsize; Load;
Ref keylen; Varsize; Load; Load;
Apply' mac/2;
Ref buf; Varsize; Load;
Ref len; Varsize; Load;
Apply' +/2; Store;
//read(buf + len + 20, 20);
Const i20; In x2 read;
Ref buf; Varsize; Load;
Ref len; Varsize; Load;
Const i20;
Apply' +/2; Apply' +/2; Store;
//if(memcmp(buf + len,
//          buf + len + 20,
//          20) == 0)
Ref buf; Varsize; Load;
Ref len; Varsize; Load; Apply' +/2;
Const i20; Load;
Ref buf; Varsize; Load;
Ref len; Varsize; Load;
Const i20; Apply' +/2; Apply' +/2;
Const i20; Load;
Apply' cmp/2;
Const 0; Apply' ==/2; Test;
//  event(”accept”, buf, len);
Ref buf; Varsize; Load;
Ref len; Varsize; Load; Load;
Event;
```

Figure 13: Translation of the example C program (fig. 1) into CVM.

B. PROTOCOL TRANSITION SYSTEMS

This section establishes the definition of security that we use in the paper and gives some sufficient conditions under which a protocol transformation (as done, for instance, by translating from a description of a protocol in C to a description in a more abstract language) preserves security.

In order to define security for a program we first need to define the protocol that the program implements. The notion of a protocol is formally captured by a *protocol transition system (PTS)*, defined as follows: a PTS is a triple $(S, s_I, \rightarrow)$, where S is a set, $s_I \in S$ and $\rightarrow$ is a labelled transition relation with transitions of the form

$$(\eta, s) \xrightarrow{l} \{(\eta_1, s_1), \ldots, (\eta_n, s_n)\},$$

where η and η_i are valuations, $s, s_i \in S$, and the right hand side is a nonempty multiset. We call a pair (η, s) an *executing process* and think of η as an environment in which s executes. We require that each executing process is of one of the following types:

- a *reading process*, in which case all outgoing labels are of the form `read` b with $b \in BS$,
- a *control process*, in which case all outgoing labels are of the form `ctr` b with $b \in BS$,
- a *randomising process*, in which case all outgoing labels are of the form `rnd` b with $b \in BS$ and all b have the same length,
- a *writing process*, in which case there is a single outgoing transition with label of the form `write` b with $b \in BS$,
- an *event process*, in which case there is a single outgoing transition with label of the form `event` b with $b \in BS$.

We require that the transition relation is deterministically computable: there should exist a probabilistic algorithm that

- given a left hand side which is a reading or a control process and a label computes the right hand side (in particular, the right hand side is uniquely determined),
- given a left hand side which is a randomising process chooses one of the admissible outgoing labels uniformly at random and computes the right hand side,
- given a left hand side which is a writing or an event process computes the outgoing label and the right hand side,
- given inputs for which there is no transition, or malformed inputs, returns *"wrong"*.

The semantics of languages that we use (CVM and IML) will be given as a function from programs to PTS.

We now define protocol states and show how they evolve. Intuitively a protocol state is just a collection of executing processes. The attacker repeatedly chooses one of the processes, which is then allowed to perform a transition according to the PTS rules. The executing processes are assigned handles so that the attacker can refer to them. A handle is a sequence of all observable transitions that have been performed by the process so far—this way the handle contains all the information that the attacker has about a process.

Formally, an *observation* is either an integer or one of the reading, control, or writing labels. A *process history* is a sequence of observations. A *protocol state* over a PTS T is a partial map from process histories to executing processes over T. We extend the transition relation of T to a transition relation over protocol states as follows: Let $\mathcal{P}$ be a protocol state and $h \in \mathrm{dom}(\mathcal{P})$ a process history such that T contains a transition of the form

$$\mathcal{P}(h) \xrightarrow{l} \{(\eta_1, s_1), \ldots, (\eta_n, s_n)\}$$

Let

$$\mathcal{P}' = \mathcal{P}_{-h}\, \{hoi \mapsto (\eta_i, s_i) \mid 1 \leq i \leq n\}\,,$$

where $o = l$ if l is an observation and $o = \varepsilon$ otherwise, and we use an abbreviation $f_{-x} = f\{x \mapsto \bot\}$. Then there is a transition $\mathcal{P} \xrightarrow{c,\, a} \mathcal{P}'$ between protocol states $\mathcal{P}$ and $\mathcal{P}'$ with a *command* c and an *action* a, where

- $c = (h, l)$ and $a = \varepsilon$ if l is a control label, or a read label,
- $c = (h, \varepsilon)$ and $a = l$ if l is a a randomising label, a write label, or an event label.

Given an initial protocol state and a command, the action and the resulting state are computable by the assumption that the underlying PTS transitions are computable. We extend the definition to multiple transitions and write $\mathcal{P} \xrightarrow{c_1 \cdots c_n,\, a_1 \cdots a_m}{}^{*} \mathcal{P}'$, iff there is a sequence of transitions leading from $\mathcal{P}$ to $\mathcal{P}'$ with commands $c_1, \ldots, c_n$ and actions $a_1, \ldots, a_m$.

We shall be interested in the sequence of events raised by a protocol in the presence of an attacker. The execution of a protocol is defined as follows:

Definition 2 (Protocol execution) Given a PTS $T = (S, s_I, \rightarrow)$ and an interactive probabilistic machine E (an attacker) we define the execution of the protocol T as a probabilistic machine $\mathrm{Exec}(T, E)$ that proceeds as follows:

Maintain a protocol state $\mathcal{P}$. Initially $\mathcal{P} = \{\varepsilon \mapsto (\emptyset, s_I)\}$. Keep receiving commands from the attacker and for each command c

- compute a transition $\mathcal{P} \xrightarrow{c,\, a} \mathcal{P}'$ and set $\mathcal{P} := \mathcal{P}'$. If no such transition exists or if the command is malformed, terminate,
- if $a =$ `write` b, send b to the attacker,
- if $a =$ `event` b, raise event b. □

We shall assume that $\mathrm{Exec}(T, E)$ uses the most efficient algorithm to compute the PTS transitions. For a PTS T, an attacker E, and a resource bound $t \in \mathbb{N}$ let $\mathrm{Events}(T, E, t)$ be the sequence of events raised by the execution of $\mathrm{Exec}(T, E)$ during the first t elementary computation steps (each protocol transition will typically involve multiple steps). We define a *trace property* as a polynomially decidable prefix-closed set of event sequences. This leads us to the definition of security for protocols:

Restatement of definition 1 *For a PTS T, a trace property ρ and a resource bound $t \in \mathbb{N}$ let*

$$\begin{aligned} &\mathrm{insec}(T, \rho, t) \\ &\quad = \sup\{\Pr[\mathrm{Events}(T, E, t) \notin \rho] \mid E \text{ attacker},\ |E| \leq t\}\,, \end{aligned}$$

where $|E|$ is the size of the description of the attacker.

Intuitively $\mathrm{insec}(T, \rho, t)$ measures the success probability of the most successful attack against T and property ρ when both the execution time of the attack and the size of the attacker code are bounded by t.

In the following we define a simulation relation on PTS that preserves security. This relation will be used as a tool to relate the security of a protocol described by a low-level CVM program P to the security of a protocol described by a more abstract IML process $\tilde{P}$ that results from the symbolic execution of P.

In the definition of the simulation relation we shall refer to a slightly generalised notion of the protocol execution, parameterised by the initial environment: given a PTS T with an initial state s_I, an attacker E, and a valuation η, let $\mathrm{Exec}_\eta(T, E)$ be the machine that executes like $\mathrm{Exec}(T, E)$, but starts with $\{\varepsilon \mapsto (\eta, s_I)\}$ as the initial state.

We shall be interested in PTS in which the number of steps to reach a certain state is independent of how it is reached, as captured by the following definition:

Definition 3 (History-independent PTS) A PTS T is called *history-independent* iff, whenever for any valuation η and attackers E and $\tilde{E}$ the machine $\mathrm{Exec}_\eta(T, E)$ reaches a protocol state $\mathcal{P}$ in t non-attacker steps and the machine $\mathrm{Exec}_\eta(T, \tilde{E})$ reaches $\mathcal{P}$ in $\tilde{t}$ non-attacker steps, $\tilde{t} = t$.

Given a history-independent PTS T, a protocol state $\mathcal{P}$ over T and a valuation η we say that T *reaches* $\mathcal{P}$ *from* η *in* t *steps* iff t is the number of non-attacker steps in which $\mathrm{Exec}_\eta(T, E)$ reaches $\mathcal{P}$ for some attacker E.

For the PTS defined in this paper we shall ensure history-independence by recording enough information in the state to be able to reconstruct the set of transitions that lead into that state.

Intuitively we shall say that a PTS $\tilde{T}$ simulates a PTS T when an attacker has a way of playing against $\tilde{T}$ in such a way that it solicits the same sequence of actions as when playing against T. In other words, given an execution trace of T, it should be feasible to reconstruct an execution trace of $\tilde{T}$ with the same sequence of actions. The only additional complication is that the reconstruction should happen online, that is, the translation of a prefix of a trace should not depend on what follows the prefix. This corresponds to the fact that the attacker cannot see into the future. We achieve the on-line property by demanding that there is an equivalence relation between protocol states of T and $\tilde{T}$ such that for each transition from $\mathcal{P}$ to $\mathcal{P}'$ in T and a state $\tilde{\mathcal{P}}$ equivalent to $\mathcal{P}$ there is a transition from $\tilde{\mathcal{P}}$ to $\tilde{\mathcal{P}}'$ in $\tilde{T}$ such that $\tilde{\mathcal{P}}'$ is equivalent to $\mathcal{P}'$. Most of the technicalities of the definition deal with placing restrictions on the computability of these transitions.

Definition 4 (Simulation relation on PTS) For a polynomial p we say that PTS $\tilde{T}$ with initial state $\tilde{s}_I$ *p-simulates* a PTS T with initial state s_I, writing $T \lesssim_p \tilde{T}$ iff both T and $\tilde{T}$ are history-independent and there exists a relation $\lesssim$ between protocol states of T and protocol states of $\tilde{T}$ and a partial map τ from commands to sequences of commands such that

1. for all valuations η
$$\{\varepsilon \mapsto (\eta, s_I)\} \lesssim \{\varepsilon \mapsto (\eta, \tilde{s}_I)\},$$
2. if $\mathcal{P} \lesssim \tilde{\mathcal{P}}$ and there exists a transition $\mathcal{P} \xrightarrow{c,\, a} \mathcal{P}'$ with a command c and an action a then there exists a protocol state $\tilde{\mathcal{P}}'$ of $\tilde{T}$ such that $\mathcal{P}' \lesssim \tilde{\mathcal{P}}'$ and $\tilde{\mathcal{P}} \xrightarrow{\tau(c),\, a}{}^{*}\ \tilde{\mathcal{P}}'$,
3. $\tau(c)$ is computable in $p(|c| + |s_I|)$ steps,
4. if $\mathcal{P} \lesssim \tilde{\mathcal{P}}$ and for some valuation η T reaches $\mathcal{P}$ from η in t steps and $\tilde{T}$ reaches $\tilde{\mathcal{P}}$ from η in $\tilde{t}$ steps then $\tilde{t} \le p(t)$. □

Theorem 4 (Preservation of security by simulation) *For every polynomial* p *there exists a polynomial* p' *such that whenever* $T \lesssim_p \tilde{T}$ *for PTS* T *and* $\tilde{T}$*, for any trace property* ρ *and resource bound* $t \in \mathbb{N}$
$$\mathrm{insec}(T, \rho, t) \le \mathrm{insec}(\tilde{T}, \rho, p'(t)).$$
□

Proof Let $T \lesssim_p \tilde{T}$ for PTS T and $\tilde{T}$ and a polynomial p. Given an attacker E we shall construct an attacker $\tilde{E}$ such that whenever the machine $\mathrm{Exec}(T, E)$ produces a sequence of events es within the first t steps when running with random tape R, the machine $\mathrm{Exec}(\tilde{T}, \tilde{E})$ produces the sequence es within at most $p'(t)$ steps when running with R, where p' is a polynomial depending on p. Thus, given that ρ is defined to be prefix-closed, any violation of ρ happening in T will happen in $\tilde{T}$ with at least the same probability.

The attacker $\tilde{E}$ shall run an instance of E and iterate as follows:

- Receive a sequence $c_1 \dots c_m$ of commands from E and output $\tau(c_1) \dots \tau(c_m)$,
- Forward any input to E.

Let M be the state of the machine $\mathrm{Exec}(T, E)$ running with random tape R after having processed commands $c_1 \dots c_n$ and $\tilde{M}$ the state of the machine $\mathrm{Exec}(\tilde{T}, \tilde{E})$ running with R after having processed commands $\tau(c_1) \dots \tau(c_n)$. By induction using (1)–(2) in definition 4 we can show:

- if $\mathcal{P}$ is the protocol state contained in M and $\tilde{\mathcal{P}}$ is the protocol state contained in $\tilde{M}$ then $\mathcal{P} \lesssim \tilde{\mathcal{P}}$,
- the instance of E run by $\tilde{E}$ in $\tilde{M}$ has executed the same computations as the instance of E in M, the same portion of R has been consumed, and the same sequence of events has been raised.

To bound the execution time of $\tilde{M}$ assume that M has executed t steps and $\tilde{M}$ has executed $\tilde{t}$ steps. Let $t = t_E + t_T$, where t_E is the number of steps executed by the attacker and t_T is the number of non-attacker steps. Similarly split $\tilde{t} = \tilde{t}_E + \tilde{t}_T$. The attacker $\tilde{E}$ runs an instance of E which takes time $O(t_E)$ and additionally issues n queries to τ. According to (3) in definition 4 the runtime of each query is bounded by $p(|c_{max}| + |s_I|)$, where c_{max} is the longest command received from E. Both n and $|c_{max}|$ are bounded by t_E and $|s_I|$ is bounded by t_T as $\mathrm{Exec}(T, E)$ needs to construct the initial state. Overall
$$\begin{aligned} \tilde{t}_E &\le O(t_E) + n \cdot p(|c_{max}| + |s_I|) \\ &\le O(t_E) + t_E \cdot p(t_E + t_T). \end{aligned}$$
■

According to (4) in definition 4 $\tilde{t}_T \le p(t_T)$. We conclude that $\tilde{t} \le t \cdot p(t) + p(t) + O(t)$.

We shall be interested in executing a PTS in the context of another PTS. This is useful for modelling: we shall specify the threat model for a CVM program by embedding it as a subprocess within an IML process. This way we can formally define a setting with multiple threads and shared key creation and distribution without having to add process control primitives to CVM itself. An important property of embedding that we define is that it preserves the simulation relation. In order to define the embedding we start by adding holes to PTS:

Definition 5 (PTS with a hole) Given a polynomial p we define a *PTS with a hole identifiable in p-time* as a history-independent PTS with initial state s_I that contains a special state [] such that there are no transitions from [] and such that there exists an algorithm that, given a process history h, runs in time $p(|h|+|s_I|)$ and decides whether h is a *history of a hole*, that is, whether for all protocol states $\mathcal{P}$ reachable from some environment η and such that $h \in \mathrm{dom}(\mathcal{P})$ the process $\mathcal{P}(h)$ is of the form $(\eta', [])$ with some environment η'. □

The definition intuitively states that the attacker must have an efficient means to decide whether a process is a hole given the observable history of the process. We can now proceed to defining the embedding:

Definition 6 (Embedding of PTS) Given a PTS with a hole $T_E = (S_E, s_{IE}, \to_E)$ and a PTS $T = (S, s_I, \to)$ we define the *embedding* $T_E[T]$ of T within T_E by

$$T_E[T] = (((S_E \setminus []) \times \{s_I\}) \cup S,\ (s_{IE}, s_I),\ \to'_E \cup \to),$$

where $\to'_E$ is obtained from $\to_E$ by replacing each occurrence of $s \in S_E \setminus []$ by (s, s_I) and by replacing each occurrence of [] with s_I. □

Theorem 5 (Simulation and embedding) *For each two polynomials p and p' there exists a polynomial p'' such that if T and $\tilde{T}$ are PTS with $T \lesssim_p \tilde{T}$ and T_E is a PTS with a hole identifiable in p'-time then $T_E[T] \lesssim_{p''} T_E[\tilde{T}]$.* □

Proof We start by giving a definition of embedding for protocol states. Given a protocol state $\mathcal{P}$ that contains holes with histories $h_1, \dots, h_n$ and protocol states $\mathcal{P}_1, \dots, \mathcal{P}_n$ we define the *embedding*

$$\begin{aligned}&\mathcal{P}[\mathcal{P}_1, \dots, \mathcal{P}_n]\\ &\quad = \mathcal{P}_{-h_1,\dots,h_n} \{ h_i h_j \mapsto \mathcal{P}_i(h_j) \mid 1 \le i \le n,\ h_j \in \mathrm{dom}(\mathcal{P}_i) \}.\end{aligned}$$

Let $T_E = (S_E, s_{IE}, \to_E)$, $T = (S, s_I, \to)$, and $\tilde{T} = (\tilde{S}, \tilde{s}_I, \tilde{\to})$ be defined as in the theorem. We show how to extend the relation $\lesssim$ on protocol states and the function τ given by the definition of simulation relation of T and $\tilde{T}$ to a corresponding relation $\lesssim_E$ and a function τ_E for $T_E[T]$ and $T_E[\tilde{T}]$. For a protocol state $\mathcal{P}$ over $T_E[T]$ and $\tilde{\mathcal{P}}$ over $T_E[\tilde{T}]$ we set $\mathcal{P} \lesssim_E \tilde{\mathcal{P}}$ iff there exist protocol states $\mathcal{P}_1, \dots, \mathcal{P}_n$ over T, $\tilde{\mathcal{P}}_1, \dots, \tilde{\mathcal{P}}_n$ over $\tilde{T}$ and a protocol state $\mathcal{P}_E$ over T_E such that $\mathcal{P}_i \lesssim \tilde{\mathcal{P}}_i$ for all i and

$$\mathcal{P} = \mathcal{P}_E[\mathcal{P}_1, \dots, \mathcal{P}_n] \text{ and } \tilde{\mathcal{P}} = \mathcal{P}_E[\tilde{\mathcal{P}}_1, \dots, \tilde{\mathcal{P}}_n].$$

Let a command $c = (h, d)$ be given. We compute $\tau_E(c)$ as follows: first check whether h contains a prefix h_E such that h_E is a history of a hole. If it doesn't, set $\tau_E(c) = c$, otherwise let h' be a process history such that $h = h_E h'$ and let

$$\begin{aligned}&(\tilde{h}_1, d_1), \dots, (\tilde{h}_m, d_m) = \tau((h', d)),\\ &\tau_E(c) = (h_E \tilde{h}_1, d_1), \dots, (h_E \tilde{h}_m, d_m).\end{aligned}$$

It is straightforward to check that (1)–(2) in definition 4 are satisfied for $T_E[T]$ and $T_E[\tilde{T}]$ with τ_E and $\lesssim_E$.

To prove (3) we need to bound the evaluation time of $\tau_E(c)$ for a command $c = (h, d)$ in terms of $|c|$ and $|s'_{IE}|$ where $s'_{IE} = (s_{IE}, s_I)$ is the initial state of $T_E[T]$. To evaluate $\tau_E(c)$ the following operations are performed:

- Run the hole-detection algorithm for each prefix of h. According to the assumption on T_E this can be done in $|h| \cdot p'(|h| + |s'_{IE}|)$ steps,
- if $h = h_E h'$, where h_E is a history of a hole, evaluate $\tau(c')$ for $c' = (h', d)$. According to the assumption that $T \lesssim_p \tilde{T}$ this takes $p(|c'| + |s_I|)$ steps.

Given that $|h| \le |c|$, $|c'| \le |c|$, and $|s_I| \le |s'_{IE}|$, the overall evaluation time of τ_E is bounded by

$$O(|c| \cdot p'(|c| + |s'_{IE}|) + p(|c| + |s'_{IE}|)).$$

To prove (4) choose a valuation η and assume that $T_E[T]$ reaches a state $\mathcal{P}$ from η in t steps, $T_E[\tilde{T}]$ reaches a state $\tilde{\mathcal{P}}$ from η in $\tilde{t}$ steps, and $\mathcal{P} \lesssim \tilde{\mathcal{P}}$. By definition the states are of the form

$$\mathcal{P} = \mathcal{P}_E[\mathcal{P}_1, \dots, \mathcal{P}_n] \text{ and } \tilde{\mathcal{P}} = \mathcal{P}_E[\tilde{\mathcal{P}}_1, \dots, \tilde{\mathcal{P}}_n],$$

where $\mathcal{P}_E$ is a state of T_E and $\mathcal{P}_i \lesssim \tilde{\mathcal{P}}_i$ for all i. For each i let η_i be the environment of the ith hole in $\mathcal{P}_E$. It is easy to see that $t = O(t_E + t_1 + \ldots + t_n)$, where t_E is the time in which T_E reaches $\mathcal{P}_E$ from η and t_i is the time in which T reaches $\mathcal{P}_i$ from η_i. Similarly $\tilde{t} = O(t_E + \tilde{t}_1 + \ldots + \tilde{t}_n)$, where $\tilde{t}_i$ is the time in which $\tilde{T}$ reaches $\tilde{\mathcal{P}}_i$ from η_i. From the assumption $T \lesssim_p \tilde{T}$ we know that $\tilde{t}_i \le p(t_i)$ for each i. Assuming w.l.o.g. that p is at least linear and monotonic, we conclude $\tilde{t} \le p(t)$. ■

The definition and the theorem can easily be extended to the setting with multiple holes $[]_1, \dots, []_n$. We shall write $T_E[T_1, \dots, T_n]$ to denote the corresponding embedding.

C. SEMANTICS OF CVM

This section presents the formal semantics of the CVM language, the syntax of which is introduced in fig. 3. In the following, let N, val, and bs be chosen as in section 2. In order to define the semantics, we associate to each CVM program the protocol transition system that is generated by it. Let a program $P \in \mathit{CVM}$ be given. Let $\mathrm{var}(P)$ be the set of variables used in `Ref` instructions within P and choose an allocation function $\mathrm{addr}\colon \mathrm{var}(P) \to \mathbb{N}$. We require that the allocated memory ranges do not overlap, that is

$$\{\mathrm{addr}(v)\}_N \cap \{\mathrm{addr}(v')\}_N = \emptyset \text{ for all } v \ne v'.$$

We let $[\![P]\!]_C$ be the PTS with the initial state $(\mathtt{Init}, P)$ and all other states of the form $(\mathcal{A}^c, \mathcal{M}^c, \mathcal{S}^c, P)$, as described in section 2. The transition rules of $[\![P]\!]_C$ are presented in fig. 14. The right hand side of each transition always contains a single process, so we omit the multiset bracket.

The rule (C-In) stores the input value in the environment in addition to placing it on the stack. This way the resulting PTS is history-independent—the state contains the information about all inputs so that there is only one trace leading to each state.

D. SEMANTICS OF IML

Just as for CVM, the semantics of IML is given as a protocol transition system. We choose the functions bs and val as in section 2 and let the function $[\![\cdot]\!]_\eta$ be defined as in section 4. For an IML process P we let $[\![P]\!]_I$ be the PTS with IML processes as states, with starting state P and transitions described in fig. 15.

The rules (I-Repl) and (I-Par) are standard replication and parallel composition rules from the pi calculus. The rule (I-Nonce) is interesting in that it restricts the generated nonce to be of a given length. The input rule (I-In)

$$\frac{\forall v \in \mathrm{var}(P)\colon\ \{\mathrm{addr}(v)\}_N \subseteq \mathit{Addr}}{\eta,\ (\mathtt{Init},\ P) \xrightarrow{\mathtt{ctr}\ \varepsilon} \eta,\ (\bigcup_{v\in\mathrm{var}(P)} \{\mathrm{addr}(v)\}_N,\ \emptyset,\ [],\ P)} \qquad \text{(C-Init)}$$

$$\frac{}{\eta,\ (\mathcal{A}^c,\ \mathcal{M}^c,\ \mathcal{S}^c,\ \mathtt{Const}\ b;\ P) \xrightarrow{\mathtt{ctr}\ \varepsilon} \eta,\ (\mathcal{A}^c,\ \mathcal{M}^c,\ b :: \mathcal{S}^c,\ P)} \qquad \text{(C-Const)}$$

$$\frac{}{\eta,\ (\mathcal{A}^c,\ \mathcal{M}^c,\ \mathcal{S}^c,\ \mathtt{Ref}\ v;\ P) \xrightarrow{\mathtt{ctr}\ \varepsilon} \eta,\ (\mathcal{A}^c,\ \mathcal{M}^c,\ \mathrm{bs}(\mathrm{addr}(v)) :: \mathcal{S}^c,\ P)} \qquad \text{(C-Ref)}$$

$$\frac{p \in \mathit{BS} \quad |p| = N \quad \{\mathrm{val}(p)\}_{\mathrm{val}(l)} \subseteq \mathit{Addr} \setminus \mathcal{A}^c}{\eta,\ (\mathcal{A}^c,\ \mathcal{M}^c,\ l :: \mathcal{S}^c,\ \mathtt{Malloc};\ P) \xrightarrow{\mathtt{ctr}\ p} \eta,\ (\mathcal{A}^c \cup \{\mathrm{val}(p)\}_{\mathrm{val}(l)},\ \mathcal{M}^c,\ p :: \mathcal{S}^c,\ P)} \qquad \text{(C-Malloc)}$$

$$\frac{b, b_E \in \mathit{BS} \quad |b| = \mathrm{val}(l) \le |b_E| \quad \forall i \in |b| : b[i] = \text{if } \mathrm{val}(p) + i \in \mathrm{dom}(\mathcal{M}^c) \text{ then } \mathcal{M}^c(\mathrm{val}(p) + i) \text{ else } b_E[i]}{\eta,\ (\mathcal{A}^c,\ \mathcal{M}^c,\ l :: p :: \mathcal{S}^c,\ \mathtt{Load};\ P) \xrightarrow{\mathtt{ctr}\ b_E} \eta,\ (\mathcal{A}^c,\ \mathcal{M}^c,\ b :: \mathcal{S}^c,\ P)} \qquad \text{(C-Load)}$$

$$\frac{b \in \mathit{BS} \quad |b| = \mathrm{val}(l) < 2^N}{\eta,\ (\mathcal{A}^c,\ \mathcal{M}^c,\ l :: \mathcal{S}^c,\ \mathtt{In}\ v\ \mathit{src};\ P) \xrightarrow{\mathit{src}\ b} \eta\{v \mapsto b\},\ (\mathcal{A}^c,\ \mathcal{M}^c,\ b :: \mathcal{S}^c,\ P)} \qquad \text{(C-In)}$$

$$\frac{v \in \mathrm{dom}(\eta) \quad |\eta(v)| < 2^N}{\eta,\ (\mathcal{A}^c,\ \mathcal{M}^c,\ \mathcal{S}^c,\ \mathtt{Env}\ v;\ P) \xrightarrow{\mathtt{ctr}\ \varepsilon} \eta,\ (\mathcal{A}^c,\ \mathcal{M}^c,\ \mathrm{bs}(|\eta(v)|) :: \eta(v) :: \mathcal{S}^c,\ P)} \qquad \text{(C-Env)}$$

$$\frac{\mathrm{ar}(\mathit{op}) = n \quad b = A_{\mathit{op}}(b_1, \ldots, b_n) \ne \bot \quad |b| < 2^N}{\eta,\ (\mathcal{A}^c,\ \mathcal{M}^c,\ b_1 :: \ldots :: b_n :: \mathcal{S}^c,\ \mathtt{Apply}\ \mathit{op};\ P) \xrightarrow{\mathtt{ctr}\ \varepsilon} \eta,\ (\mathcal{A}^c,\ \mathcal{M}^c,\ \mathrm{bs}(|b|) :: b :: \mathcal{S}^c,\ P)}, \qquad \text{(C-Apply)}$$

$$\frac{}{\eta,\ (\mathcal{A}^c,\ \mathcal{M}^c,\ b :: \mathcal{S}^c,\ \mathtt{Out}\ \mathit{dest};\ P) \xrightarrow{\mathit{dest}\ b} \eta,\ (\mathcal{A}^c,\ \mathcal{M}^c,\ \mathcal{S}^c,\ P)} \qquad \text{(C-Out)}$$

$$\frac{b = i1}{\eta,\ (\mathcal{A}^c,\ \mathcal{M}^c,\ b :: \mathcal{S}^c,\ \mathtt{Test};\ P) \xrightarrow{\mathtt{ctr}\ 1} \eta,\ (\mathcal{A}^c,\ \mathcal{M}^c,\ \mathcal{S}^c,\ P)} \qquad \text{(C-Test)}$$

$$\frac{\{\mathrm{val}(p)\}_{|b|} \subseteq \mathcal{A}^c}{\eta,\ (\mathcal{A}^c,\ \mathcal{M}^c,\ p :: b :: \mathcal{S}^c,\ \mathtt{Store};\ P) \xrightarrow{\mathtt{ctr}\ \varepsilon} \eta,\ (\mathcal{A}^c,\ \mathcal{M}^c\{\mathrm{val}(p) + i \mapsto b[i] \mid i \in |b|\},\ \mathcal{S}^c,\ P)} \qquad \text{(C-Store)}$$

Figure 14: The concrete semantics of CVM.

does not place such a restriction and allows the input to be of any length (this is more permissive than the CVM input rule). The rules (I-Out) and (I-Event) generate an output and an event transition respectively. The conditional rules (I-Cond-True) and (I-Cond-False) have different control labels, so that the attacker can distinguish the branch that has been taken by the process. Unlike CVM there is no explicit rule for reading environment variables, because IML operates on the environment η directly.

Consider IML enriched with an additional syntactic form $[]_i$ (a hole) with $i \in \mathbb{N}$ and without any reductions. For an IML process P with holes the semantics $[\![P]\!]_I$ is a PTS with holes (definition 5). The history of a process uniquely determines its state, for instance, given the process $P = !(\textbf{if } e \textbf{ then } [] \textbf{ else } 0)$ and history $h = (\mathtt{ctr}\ \varepsilon)\,1\,(\mathtt{ctr}\ 1)\,1$, it is easy to see that h is a history of a hole in $[\![P]\!]_I$. Here it is important that the true and the false branches in fig. 15 have different control labels. In general, whether h is a history of a hole in $[\![P]\!]_I$, is computable in time linear in $|P| + |h|$. Just like CVM IML is history-independent because it records all the inputs in the environment. Thus the following holds:

Lemma 1 (IML with holes) *For an IML process P with holes the semantics $[\![P]\!]_I$ is a PTS with holes identifiable in p-time for some fixed linear polynomial p.* □

The semantics of mixed IML and CVM processes is defined by using a PTS embedding as follows:

Definition 7 (Mixed semantics) For a process $P_E \in \mathit{IML}$ with n holes and processes $P_1, \ldots, P_n \in \mathit{CVM}$ let

$$[\![P_E[P_1, \ldots, P_n]]\!]_{CI} = [\![P_E]\!]_I[[\![P_1]\!]_C, \ldots, [\![P_n]\!]_C]. \qquad □$$

$$\frac{}{(\eta,\ !P) \xrightarrow{\texttt{ctr}\ \varepsilon} \{(\eta,\ P),\ (\eta,\ !P)\}} \qquad \text{(I-Repl)}$$

$$\frac{}{(\eta,\ P|Q) \xrightarrow{\texttt{ctr}\ \varepsilon} \{(\eta,\ P),\ (\eta,\ Q)\}} \qquad \text{(I-Par)}$$

$$\frac{b \in BS, \quad |b| = \mathrm{val}(\llbracket e \rrbracket_\eta)}{(\eta,\ (\nu x[e]); P) \xrightarrow{\texttt{rnd}\ b} \{(\eta\{x \mapsto b\},\ P)\}} \qquad \text{(I-Nonce)}$$

$$\frac{b \in BS}{(\eta,\ \mathbf{in}(x); P) \xrightarrow{\texttt{read}\ b} \{(\eta\{x \mapsto b\},\ P)\}} \qquad \text{(I-In)}$$

$$\frac{b = \llbracket e \rrbracket_\eta \neq \bot}{(\eta,\ \mathbf{out}(e); P) \xrightarrow{\texttt{write}\ b} \{(\eta,\ P)\}} \qquad \text{(I-Out)}$$

$$\frac{b = \llbracket e \rrbracket_\eta \neq \bot}{(\eta,\ \mathbf{event}(e); P) \xrightarrow{\texttt{event}\ b} \{(\eta,\ P)\}} \qquad \text{(I-Event)}$$

$$\frac{\llbracket e \rrbracket_\eta = i1}{(\eta,\ \mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q) \xrightarrow{\texttt{ctr}\ 1} \{(\eta,\ P)\}} \qquad \text{(I-Cond-True)}$$

$$\frac{\llbracket e \rrbracket_\eta = i0}{(\eta,\ \mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q) \xrightarrow{\texttt{ctr}\ 0} \{(\eta,\ Q)\}} \qquad \text{(I-Cond-False)}$$

$$\frac{b = \llbracket e \rrbracket_\eta \neq \bot}{(\eta,\ \mathbf{let}\ x = e\ \mathbf{in}\ P\ \mathbf{else}\ Q) \xrightarrow{\texttt{ctr}\ 1} \{(\eta\{x \mapsto b\},\ P)\}} \qquad \text{(I-Let-True)}$$

$$\frac{\llbracket e \rrbracket_\eta = \bot}{(\eta,\ \mathbf{let}\ x = e\ \mathbf{in}\ P\ \mathbf{else}\ Q) \xrightarrow{\texttt{ctr}\ 0} \{(\eta,\ Q)\}} \qquad \text{(I-Let-False)}$$

Figure 15: The semantics of IML.

E. SIMPLIFICATIONS

Fig. 16 presents the simplification rules used in our symbolic execution algorithm. The simplification function is concerned with simplifying range expressions when possible, for instance, an expression of the form $(a|b)\{x, y\}$, where $\Sigma \vdash (x = \mathrm{getLen}(a))$ and $\Sigma \vdash (y = \mathrm{getLen}(b))$ will simplify to b. The main work is done by two recursive functions cutL, cutR: $SExp \times SExp \to SExp$ that given a length expression l and a concatenation expression e attempt to split e at the position given by l. If this succeeds, cutL returns the part of e to the left of the split position and cutR returns the part to the right.

In order to simplify an expression of the form $e\{e_o, e_l\}$ the function simplify first checks two special cases: if e_o is equal to zero and e_l is equal to the length of e then the range can be removed and the expression can be simplified to just e. On the other hand if e_l is equal to zero then the range expression can be simplified to ε. If e is itself a range expression of the form $e'\{e'_o, e'_l\}$ then the two ranges are merged giving the result $e'\{e_o +_{\mathbb{N}} e'_o, e_l\}$. If e is a concatenation then the functions cutR and cutL are applied. Finally, if all of the above fails, the original expression is returned without simplification.

$$\mathrm{cutL}_\Sigma(l,\ e_1|\ldots|e_n) = \begin{cases} e_1|\ldots|e_{i-1}|\,\mathrm{simplify}_\Sigma(\mathrm{cutL}_\Sigma(l -_{\mathbb{N}} l', e_i)) \\ \quad \text{if } \Sigma \vdash (l \geq l') \wedge (l \leq l' +_{\mathbb{N}} \mathrm{getLen}(e_i)), \\ \quad \text{where } l' = \Sigma_{j=1}^{i-1} \mathrm{getLen}(e_j), \\ (e_1|\ldots|e_n)\{i0, l\} \ \text{ otherwise,} \end{cases}$$

$$\mathrm{cutR}_\Sigma(l,\ e_1|\ldots|e_n) = \begin{cases} \mathrm{simplify}_\Sigma(\mathrm{cutR}_\Sigma(l -_{\mathbb{N}} l', e_i))|e_{i+1}|\ldots|e_n \\ \quad \text{if } \Sigma \vdash (l \geq l') \wedge (l \leq l' +_{\mathbb{N}} \mathrm{getLen}(e_i)), \\ \quad \text{where } l' = \Sigma_{j=1}^{i-1} \mathrm{getLen}(e_j), \\ (e_1|\ldots|e_n)\{l, \mathrm{getLen}(e_1|\ldots|e_n) -_{\mathbb{N}} l\} \ \text{ otherwise,} \end{cases}$$

$$\mathrm{simplify}_\Sigma(e\{e_o, e_l\}) = \begin{cases} e & \text{if } \Sigma \vdash (e_o = i0) \\ & \qquad \wedge\ (e_l = \mathrm{getLen}(e)) \\ \varepsilon & \text{if } \Sigma \vdash (e_l = i0) \\ e'\{e_o +_{\mathbb{N}} e'_o, e_l\} & \text{if } e = e'\{e'_o, e'_l\} \\ \mathrm{cutL}_\Sigma(e_l, \mathrm{cutR}_\Sigma(e_o, e)) & \text{if } e \text{ is a concatenation} \\ e\{e_o, e_l\} & \text{otherwise.} \end{cases}$$

Figure 16: Simplification rules.

We omit the soundness proof for our simplification function.

F. SYMBOLIC EXECUTION SOUNDNESS

We prove our main result (theorem 1). We shall do so by showing that the PTS $\llbracket \llbracket P \rrbracket_S \rrbracket_I$ resulting from the symbolic execution of a program $P \in CVM$ simulates (in the sense of definition 4) the PTS $\llbracket P \rrbracket_C$ resulting from running P directly. This result is captured by lemma 4. Theorem 1 then follows by combined application of theorems 4 and 5 and lemma 1 together with definition 7.

For compactness we shall write $b^{\mathbb{N}}$ instead of $\mathrm{val}(b)$ for $b \in BS$. When referring to valuations we shall mean extended valuations of the form $\eta\colon Var \cup PBase \to BS_\bot$. For an extended valuation η let $\mathrm{var}(\eta)$ be the restriction of η to Var.

We shall make use of the soundness of the function getLen introduced in section 6 that we state here without proof: for any $e \in SExp$ and valuation η

$$(\llbracket e \rrbracket_\eta \neq \bot) \wedge (|\llbracket e \rrbracket_\eta| < 2^N) \Rightarrow \llbracket \mathrm{getLen}(e) \rrbracket_\eta = \mathrm{bs}(|\llbracket e \rrbracket_\eta|).$$

The main tool in the proof of lemma 4 is a concretisation function that, given a valuation, maps symbolic execution states to concrete execution states. Given a symbolic state $s = (\Sigma, \mathcal{A}^s, \mathcal{M}^s, \mathcal{S}^s, P)$ and a valuation η we say that s is *η-consistent* when all expressions in s are well-defined with respect to η, when η maps all symbolic memory locations to disjoint ranges that are within allocated memory bounds, all conditions in Σ hold with respect to η, and η agrees with the addr function for stack variables. Formally, we say that s is η-consistent, iff

1. for all $pb \in \mathrm{dom}(\mathcal{M}^s)$:

$$\llbracket \mathcal{M}^s(pb) \rrbracket_\eta \neq \bot, \quad \llbracket \mathcal{A}^s(pb) \rrbracket_\eta \neq \bot, \quad \eta(pb) \neq \bot,$$
$$|\llbracket \mathcal{M}^s(pb) \rrbracket_\eta| \leq \llbracket \mathcal{A}^s(pb) \rrbracket_\eta^{\mathbb{N}},$$

2. for all pb, $pb' \in \mathrm{dom}(\mathcal{A}^s)$ with $pb \neq pb'$:

   $$\left\{\eta(pb)^{\mathbb{N}}\right\}_{\llbracket\mathcal{A}^s(pb)\rrbracket^{\mathbb{N}}_{\eta}} \cap \left\{\eta(pb')^{\mathbb{N}}\right\}_{\llbracket\mathcal{A}^s(pb')\rrbracket^{\mathbb{N}}_{\eta}} = \emptyset,$$

3. for all $\psi \in \Sigma$: $\llbracket\psi\rrbracket_\eta = i1$,
4. for all $e \in \mathcal{S}^s$: $\llbracket e\rrbracket_\eta \neq \bot$,
5. for all $v \in \mathrm{var}(P)$: $\eta(\mathrm{stack}\ v) = \mathrm{bs}(\mathrm{addr}(v))$.

For an η-consistent state s let $\mathrm{conc}_\eta(s) = (\mathcal{A}^{sc}, \mathcal{M}^{sc}, \mathcal{S}^{sc}, P)$ be the concrete state where $\mathcal{S}^{sc}$ is obtained from $\mathcal{S}^s$ by applying $\llbracket\cdot\rrbracket_\eta$ to each element and

$$\mathcal{A}^{sc} = \bigcup \left\{ \left\{\eta(pb)^{\mathbb{N}}\right\}_{\llbracket\mathcal{A}^s(pb')\rrbracket^{\mathbb{N}}_{\eta}} \,\middle|\, pb \in \mathrm{dom}(\mathcal{A}^s) \right\},$$

$$\mathcal{M}^{sc} = \left\{ \eta(pb)^{\mathbb{N}} + i \mapsto \llbracket\mathcal{M}^s(pb)\rrbracket_\eta[i] \,\middle|\, \begin{array}{l} pb \in \mathrm{dom}(\mathcal{M}^s), \\ i < |\llbracket\mathcal{M}^s(pb)\rrbracket_\eta| \end{array} \right\}.$$

The conditions of η-consistency guarantee that $\mathcal{M}^{sc}$ is well-defined: symbolic expressions will map onto concrete memory without overlapping, that is, for each $p \in \mathbb{N}$ there is only one pair pb, i such that $\eta(pb)^{\mathbb{N}} + i = p$.

The special state $(\mathtt{Init}, P)$ is defined to be η-consistent for any η with $\mathrm{dom}(\eta) \subsetneq \mathit{Var}$ and we let $\mathrm{conc}_\eta(\mathtt{Init}, P) = (\mathtt{Init}, P)$.

We start by proving two lemmas relating the symbolic and the concrete execution of a program. Lemma 2 shows that if a symbolic state s maps to a concrete state c then the state following s in the symbolic execution can be mapped to the state following c in the concrete execution. Lemma 3 shows that if in a symbolic and a concrete execution the states can be mapped to each other then the IML program generated by the symbolic execution performs the same actions as the concrete execution.

Lemma 2 *Let* $(\eta_c, c) \xrightarrow{l} (\eta'_c, c')$ *be a concrete transition (fig. 14),* $s \xrightarrow{\lambda} s'$ *a symbolic transition (fig. 8), and* η *an extension of* η_c *such that* s *is* η*-consistent and* $\mathrm{conc}_\eta(s) = c$*. Then there exists an extension* η' *of both* η *and* η'_c *such that* s' *is* η'*-consistent and* $\mathrm{conc}_{\eta'}(s') = c'$*.* □

PROOF By definition of the concretisation function both the concrete and the symbolic step are executed with the same instruction or both perform the initialisation. We prove the lemma by enumerating the pairs of rules that generate the transitions. For the purpose of this proof we are not interested in the values of transition labels l and λ.

In the following $\mathcal{A}^c, \ldots$ and $\mathcal{A}^{c\prime}, \ldots$ refer to components of c and c' respectively, $\mathcal{A}^s, \ldots$ and $\mathcal{A}^{s\prime}, \ldots$ refer to components of s and s', and $\mathcal{A}^{sc}, \ldots$ and $\mathcal{A}^{sc\prime}, \ldots$ refer to components of $\mathrm{conc}_\eta(s)$ and $\mathrm{conc}_{\eta'}(s')$.

1. (C-Init) and (S-Init)

   By definition of η-consistency for the initial state we know that $\mathrm{stack}\ v \notin \mathrm{dom}(\eta)$ for all $v \in \mathrm{var}(P)$. We show that the lemma holds with

   $$\eta' = \eta\left\{\mathrm{stack}\ v \mapsto \mathrm{bs}(\mathrm{addr}(v)) \mid v \in \mathrm{var}(P)\right\}.$$

   The second condition of η'-consistency of s' follows by the choice of addr function (appendix C), the other conditions are straightforward to check. In s' each location in the symbolic memory is initialised to ε, so applying the definition of conc_η we see that $\mathcal{M}^{sc\prime} = \emptyset = \mathcal{M}^{c\prime}$. Finally

   $$\begin{aligned} \mathcal{A}^{sc\prime} &= \bigcup_{v \in \mathrm{var}(P)} \left\{ \left\{\eta'(\mathrm{stack}\ v)^{\mathbb{N}}\right\}_{\llbracket\mathcal{A}^{s\prime}(\mathrm{stack}\ v)\rrbracket^{\mathbb{N}}_{\eta'}} \right\} \\ &= \bigcup_{v \in \mathrm{var}(P)} \left\{ \left\{\mathrm{bs}(\mathrm{addr}(v))^{\mathbb{N}}\right\}_{\mathrm{bs}(N)^{\mathbb{N}}} \right\} \\ &= \bigcup_{v \in \mathrm{var}(P)} \left\{ \{\mathrm{addr}(v)\}_N \right\} = \mathcal{A}^{c\prime}. \end{aligned}$$

2. (C-Const) and (S-Const) with `Const` b

   Both the concrete and the symbolic transition have the effect of putting the same bitstring b onto the stack. Thus both the η-consistency and the state correspondence are preserved and the lemma holds with $\eta' = \eta$.

3. (C-Ref) and (S-Ref) with `Ref` v

   The concrete transition puts $\mathrm{bs}(\mathrm{addr}(v))$ on the stack and the symbolic transition puts $\mathrm{ptr}(\mathrm{stack}\ v,\ i0)$ on the stack. By η-consistency $\eta(\mathrm{stack}, v) = \mathrm{bs}(\mathrm{addr}(v))$, thus

   $$\llbracket\mathrm{ptr}(\mathrm{stack}\ v,\ i0)\rrbracket_\eta = \eta(\mathrm{stack}, v) +_b i0 = \mathrm{bs}(\mathrm{addr}(v))$$

   and the lemma holds with $\eta' = \eta$.

4. (C-Malloc) and (S-Malloc)

   Let p and l be defined as in rule (C-Malloc) and pb and e_l be defined as in rule (S-Malloc). We show that the lemma holds with $\eta' = \eta\{pb \mapsto p\}$. It is straightforward to check that the first condition of η'-consistency of s' holds, taking into consideration that $\llbracket\mathcal{M}^{s\prime}(pb)\rrbracket_\eta = \varepsilon$. To prove the second condition, let $pb' \in \mathrm{dom}(\mathcal{A}^{s\prime})$ such that $pb' \neq pb$. In that case $pb' \in \mathrm{dom}(\mathcal{A}^s)$ and by definition of conc_η and the state correspondence of c and s

   $$\begin{aligned} \left\{\eta'(pb')^{\mathbb{N}}\right\}_{\llbracket\mathcal{A}^{s\prime}(pb')\rrbracket^{\mathbb{N}}_{\eta'}} &= \left\{\eta(pb')^{\mathbb{N}}\right\}_{\llbracket\mathcal{A}^{s}(pb')\rrbracket^{\mathbb{N}}_{\eta}} \\ &\subseteq \mathcal{A}^{sc} = \mathcal{A}^c. \end{aligned}$$

   By initial state correspondence and the definition of η'

   $$\begin{aligned} \left\{\eta'(pb)^{\mathbb{N}}\right\}_{\llbracket\mathcal{A}^{s\prime}(pb)\rrbracket^{\mathbb{N}}_{\eta'}} &= \left\{\eta'(pb)^{\mathbb{N}}\right\}_{\llbracket e_l\rrbracket^{\mathbb{N}}_{\eta'}} \\ &= \left\{p^{\mathbb{N}}\right\}_{\llbracket e_l\rrbracket^{\mathbb{N}}_{\eta}} = \left\{p^{\mathbb{N}}\right\}_{l^{\mathbb{N}}} \subseteq \mathit{Addr} \setminus \mathcal{A}^c. \end{aligned}$$

   Thus the allocation ranges of pb and pb' are disjoint and the condition (2) holds. Conditions (3) to (5) are straightforward to check. To prove that $\mathrm{conc}_{\eta'}(s') = c'$ observe that

   $$\mathcal{A}^{sc\prime} = \mathcal{A}^{sc} \cup \left\{\eta'(pb)^{\mathbb{N}}\right\}_{\llbracket\mathcal{A}^{s\prime}(pb)\rrbracket^{\mathbb{N}}_{\eta'}} = \mathcal{A}^c \cup \left\{p^{\mathbb{N}}\right\}_{l^{\mathbb{N}}} = \mathcal{A}^{c\prime},$$

   $$\mathcal{M}^{sc\prime} = \mathcal{M}^{sc} = \mathcal{M}^c = \mathcal{M}^{c\prime},$$

   $$\llbracket\mathrm{ptr}(pb,\ i0)\rrbracket_{\eta'} = \eta'(pb) +_b i0 = p.$$

5. (C-Load) and (S-Load)

   Both the concrete and the symbolic rule have the effect of replacing two values on the stack with a new value. In the concrete transition the new value is $b \in BS$ such that $b[i] = \mathcal{M}^c(p+i)$ whenever $\mathcal{M}^c(p+i)$ is initialised and p is defined as in rule (C-Load). In the symbolic transition the new value is $e = \mathrm{simplify}_\Sigma(\mathcal{M}^s(pb)\{e_o, e_l\})$, where pb, e_o, and e_l are defined as in rule (S-Load). We shall prove that $[\![e]\!]_\eta = b$ so that the lemma holds with $\eta' = \eta$.

Let $b_h = [\![\mathcal{M}^s(pb)]\!]_\eta$. By definition of conc_η and initial state correspondence

$$b_h = \mathcal{M}^{sc}\left(\left\{b_{pb}^{\mathbb{N}}\right\}_{|b_h|}\right) = \mathcal{M}^{c}\left(\left\{b_{pb}^{\mathbb{N}}\right\}_{|b_h|}\right),$$

where we use the notation $\mathcal{M}^c(I)$ for $I \subseteq \mathit{Addr}$ to denote the sequence of bits of $\mathcal{M}^c$ with addresses in I. Thus $\mathcal{M}^c$ is defined in the range $\{b_{pb}^{\mathbb{N}}\}_{|b_h|}$, in particular

$$b_{pb}^{\mathbb{N}} + |b_h| < 2^N. \qquad (*)$$

Let $b_o = [\![e_o]\!]_\eta$ and $b_l = [\![e_l]\!]_\eta$. Evaluating the conditions of the rule (S-Load) and using the assumption of η-consistency we obtain $b_o +_{\mathbb{N}} b_l \le [\![\mathrm{getLen}(e_h)]\!]_\eta$. Because $|b_h| < 2^N$ we can apply soundness of getLen which together with the definitions of bitstring operations $+_{\mathbb{N}}$ and $\le$ gives

$$b_o^{\mathbb{N}} + b_l^{\mathbb{N}} \le |b_h|. \qquad (**)$$

Using the definition of the function sub

$$\begin{aligned}[\![\mathcal{M}^s(pb)\{e_o, e_l\}]\!]_\eta &= \mathrm{sub}(b_h, b_o^{\mathbb{N}}, b_l^{\mathbb{N}}) \\ &= \mathrm{sub}\left(\mathcal{M}^c\left(\left\{b_{pb}^{\mathbb{N}}\right\}_{|b_h|}\right), b_o^{\mathbb{N}}, b_l^{\mathbb{N}}\right) \\ &= \mathcal{M}^c\left(\left\{b_{pb}^{\mathbb{N}} + b_o^{\mathbb{N}}\right\}_{b_l^{\mathbb{N}}}\right).\end{aligned}$$

This allows us to apply soundness of simplify:

$$\begin{aligned}[\![e]\!]_\eta &= [\![\mathrm{simplify}_\Sigma(\mathcal{M}^s(pb)\{e_o, e_l\})]\!]_\eta \\ &= [\![\mathcal{M}^s(pb)\{e_o, e_l\}]\!]_\eta = \mathcal{M}^c\left(\left\{b_{pb}^{\mathbb{N}} + b_o^{\mathbb{N}}\right\}_{b_l^{\mathbb{N}}}\right).\end{aligned}$$

By the state correspondence of c and s we obtain

$$\begin{aligned}&p = [\![\mathrm{ptr}(pb, e_o)]\!]_\eta = \eta(pb) +_b [\![e_o]\!]_\eta = b_{pb} +_b b_o, \\ &|b| = [\![e_l]\!]^{\mathbb{N}} = b_l^{\mathbb{N}}.\end{aligned}$$

By (*) and (**) $b_{pb}^{\mathbb{N}} + b_o^{\mathbb{N}} < 2^N$, thus

$$b_{pb}^{\mathbb{N}} + b_o^{\mathbb{N}} = (b_{pb} +_b b_o)^{\mathbb{N}} = p^{\mathbb{N}}.$$

Substituting this into the above we get

$$[\![e]\!]_\eta = \mathcal{M}^c\left(\left\{b_{pb}^{\mathbb{N}} + b_o^{\mathbb{N}}\right\}_{b_l^{\mathbb{N}}}\right) = \mathcal{M}^c\left(\left\{p^{\mathbb{N}}\right\}_{|b|}\right) = b$$

The final equality holds as the referenced memory cells lie within the initialised range $\{b_{pb}^{\mathbb{N}}\}_{|b_h|}$.

6. (C-In) and (S-In) with `In` v src

   The rule (C-In) takes a value l from the stack and places a value b of length $l^{\mathbb{N}}$ on the stack. Additionally it updates $\eta_c' = \eta_c\{v \mapsto b\}$. The rule (S-In) takes an expression e_l from the stack, places v on the stack, and adds the fact $\mathrm{len}(v) = e_l$ to Σ. We show that the lemma holds with $\eta' = \eta\{v \mapsto b\}$. Due to initial state correspondence $[\![e_l]\!]_\eta = l$ and due to the condition of the rule (C-In) $|b| < 2^N$, thus

   $$[\![\mathrm{len}(v)]\!]_{\eta'}^{\mathbb{N}} = \mathrm{bs}(|b|)^{\mathbb{N}} = |b| = l^{\mathbb{N}} = [\![e_l]\!]_{\eta'}^{\mathbb{N}},$$

   so that the new fact is indeed valid.

7. (C-Env) and (S-Env) with `Env` v

   The rule (C-Env) places $\eta_e(v)$ together with $\mathrm{bs}(|\eta_e(v)|)$ on the stack (the valuation η in fig. 14 corresponds to η_e in the lemma). The rule (S-Env) places v and $\mathrm{len}(v)$ on the stack. By assumption of the lemma $\eta(v) = \eta_e(v)$, so it is straightforward to check that the lemma holds with $\eta' = \eta$.

8. (C-Apply) and (S-Apply) with `Apply` op

   The rule (C-Apply) places on the stack the bitstring $b = A_{op}(b_1, \ldots, b_n)$ together with its length, whereby $b_1, \ldots, b_n$ are taken from the stack. The rule (S-Apply) places on the stack the value $e = \mathrm{apply}(\mathit{op}, e_1, \ldots, e_n)$ together with $\mathrm{len}(e)$, whereby $e_1, \ldots, e_n$ are taken from the stack. We show that $[\![e]\!]_\eta = b$ so that the lemma holds with $\eta' = \eta$. By initial state correspondence we have $[\![e_i]\!]_\eta = b_i$ for all i. We enumerate the cases arising from the definition of apply given $b \ne \bot$:

   (a) $n = 2$, $e_1 = \mathrm{ptr}(pb, e_o)$, $e_2 \in \mathit{IExp}$, and $\mathit{op} = +_b$. In this case

   $$\begin{aligned}b &= [\![\mathrm{ptr}(pb, e_o)]\!]_\eta +_b [\![e_2]\!]_\eta \\ &= \eta(pb) +_b [\![e_o]\!]_\eta +_b [\![e_2]\!]_\eta \\ &= [\![\mathrm{ptr}(pb, e_o +_b e_2)]\!]_\eta \\ &= [\![\mathrm{apply}(+_b, \mathrm{ptr}(pb, e_o), e_2)]\!]_\eta\end{aligned}$$

   (b) $n = 2$, $e_1 = \mathrm{ptr}(pb, e_o)$, $e_2 = \mathrm{ptr}(pb, e_o')$, $\mathit{op} = -_b$. In this case

   $$\begin{aligned}b &= [\![\mathrm{ptr}(pb, e_o)]\!]_\eta -_b [\![\mathrm{ptr}(pb, e_o')]\!]_\eta \\ &= \eta(pb) +_b [\![e_o]\!]_\eta -_b (\eta(pb) +_b [\![e_o']\!]_\eta) \\ &= [\![e_o]\!]_\eta -_b [\![e_o']\!]_\eta \\ &= [\![\mathrm{apply}(-_b, \mathrm{ptr}(pb, e_o), \mathrm{ptr}(pb, e_o'))]\!]_\eta\end{aligned}$$

   (c) $e_1, \ldots, e_n \in \mathit{IExp}$. In this case

   $$\begin{aligned}b &= A_{op}([\![e_1]\!]_\eta, \ldots, [\![e_n]\!]_\eta) = [\![\mathit{op}(e_1, \ldots, e_n)]\!]_\eta \\ &= [\![\mathrm{apply}(\mathit{op}, e_1, \ldots, e_n)]\!]_\eta\end{aligned}$$

9. (C-Out) and (S-Out)

   The lemma holds trivially with $\eta' = \eta$.

10. (C-Test) and (S-Test)

    The rule (C-Test) removes a value b from the stack. The rule (S-Test) removes an expression e from the stack and adds e to the set of facts. We show that the lemma holds with $\eta' = \eta$. We only need to prove that $[\![e]\!]_\eta = i1$, but this follows from the assumption of the lemma that $[\![e]\!]_\eta = b$ and the condition $b = i1$ of the rule (C-Test).

11. (C-Store) and (S-Store)

    Both the concrete and the symbolic transition perform a memory update. These updates are

    $$\begin{aligned}\mathcal{M}^{c\prime} &= \mathcal{M}^c\left\{p^{\mathbb{N}} + i \mapsto b[i] \;\middle|\; i < |b|\right\}, \\ \mathcal{M}^{s\prime} &= \mathcal{M}^s\{pb \mapsto e_h'\}, \end{aligned} \qquad (1)$$

    where p and b are defined as in rule (C-Store), and pb and e_h' are defined as in rule (S-Store).

    We shall prove that the lemma holds with $\eta' = \eta$. We start by showing that s' is η-consistent. As the transition only updates the memory, we only need to check that $[\![e_h']\!]_\eta \ne \bot$ and $|[\![e_h']\!]_\eta| \le [\![\mathcal{A}^s(pb)]\!]_\eta^{\mathbb{N}}$. Let e_h, e_s,

e_{lh}, e_l, e_o, e, and pb be defined as in (S-Store). For $e_x \in \{e_h, e_s, e_{lh}, e_l, e_o\}$ let $b_x = [\![e_x]\!]_\eta$ and let $b_{pb} = [\![pb]\!]_\eta$. By initial state correspondence $[\![e]\!]_\eta = b$. The rule (C-Store) assumes $\left\{p^{\mathbb{N}}\right\}_{|b|} \subseteq \mathcal{A}^c$, which implies $|b| < 2^N$. Using the soundness of getLen and the definition of b_l we obtain

$$b_l^{\mathbb{N}} = [\![\mathrm{getLen}(e)]\!]_\eta^{\mathbb{N}} = |[\![e]\!]_\eta| = |b|\,.$$

By initial state correspondence

$$b_h = \mathcal{M}^c\left(\left\{b_{pb}^{\mathbb{N}}\right\}_{|b_h|}\right) = \mathcal{M}^c\left(\left\{b_{pb}^{\mathbb{N}}\right\}_{b_{lh}^{\mathbb{N}}}\right),$$

where the second equality follows by soundness of getLen and the fact

$$b_{pb}^{\mathbb{N}} + |b_h| < 2^N \tag{2}$$

established by the first equality.

We shall distinguish between two cases in the premise of the rule (S-Store). The first case is

$$\Sigma \vdash (e_o +_{\mathbb{N}} e_l < e_{lh}), \quad e_h' = \mathrm{simplify}_\Sigma(e_h''), \text{ where}$$
$$e_h'' = e_h\{0, e_o\}|e|e_h\{e_o +_{\mathbb{N}} e_l, e_{lh} -_{\mathbb{N}} (e_o +_{\mathbb{N}} e_l)\}.$$

In this case the same argument as for the rule (S-Load) yields

$$b_o^{\mathbb{N}} + b_l^{\mathbb{N}} < |b_h| = b_{lh}^{\mathbb{N}}. \tag{A3}$$

Substituting the value of b_h and expanding the definition of the function sub under consideration of (A3) we obtain

$$\begin{aligned}[\![e_h'']\!]_\eta = \mathcal{M}^c\left(\left\{b_{pb}^{\mathbb{N}}\right\}_{b_o^{\mathbb{N}}}\right)\Big|\, b \\ \Big|\mathcal{M}^c\left(\left\{b_{pb}^{\mathbb{N}} + b_o^{\mathbb{N}} + b_l^{\mathbb{N}}\right\}_{b_{lh}^{\mathbb{N}} - b_o^{\mathbb{N}} - b_l^{\mathbb{N}}}\right).\end{aligned} \tag{A4}$$

Applying the soundness of the function simplify we get $[\![e_h']\!]_\eta = [\![e_h'']\!]_\eta \neq \bot$. From $b_l^{\mathbb{N}} = |b|$ follows $|[\![e_h']\!]_\eta| = |[\![e_h]\!]_\eta|$. By initial state correspondence $|[\![e_h]\!]_\eta| \leq [\![\mathcal{A}^s(pb)]\!]_\eta^{\mathbb{N}}$. This proves η-consistency of s' in the first case.

The second case in the premise of the rule (S-Store) is

$$\Sigma \vdash (e_o +_{\mathbb{N}} e_l \geq e_{lh}) \wedge (e_o \leq e_{lh}) \wedge (e_o +_{\mathbb{N}} e_l \leq e_s),$$
$$e_h' = \mathrm{simplify}_\Sigma(e_h\{0, e_o\}|e).$$

Together with η-consistency of s this implies the following condition on bitstrings:

$$(b_o^{\mathbb{N}} + b_l^{\mathbb{N}} \geq b_{lh}^{\mathbb{N}}) \wedge (b_o^{\mathbb{N}} \leq b_{lh}^{\mathbb{N}}) \wedge (b_o^{\mathbb{N}} + b_l^{\mathbb{N}} \leq b_s^{\mathbb{N}}). \tag{B3}$$

This allows us to expand the definition of sub and apply soundness of simplify to obtain

$$[\![e_h']\!]_\eta = \mathcal{M}^c\left(\left\{b_{pb}^{\mathbb{N}}\right\}_{b_o^{\mathbb{N}}}\right)\Big|\, b \neq \bot, \tag{B4}$$

Using (B3)

$$\left|[\![e_h']\!]_\eta\right| = b_o^{\mathbb{N}} + |b| = b_o^{\mathbb{N}} + b_l^{\mathbb{N}} \leq b_s^{\mathbb{N}} = [\![\mathcal{A}^s(pb)]\!]_\eta^{\mathbb{N}},$$

which proves η-consistency of s' in the second case.

The next step is to show that $\mathrm{conc}_\eta(s') = c'$. Both in the first and in the second case above $|[\![e_h']\!]_\eta| \geq |[\![e_h]\!]_\eta|$ (in the first case they are equal, in the second case it follows from (B3)). Comparing the definition of $\mathcal{M}^{sc\prime}$ and $\mathcal{M}^{sc}$ and using the relation (1) between $\mathcal{M}^{s\prime}$ and $\mathcal{M}^s$

$$\mathcal{M}^{sc\prime} = \mathcal{M}^{sc}\left\{b_{pb}^{\mathbb{N}} + i \mapsto ([\![e_h']\!]_\eta)[i] \;\Big|\; i < \left|[\![e_h']\!]_\eta\right|\right\}.$$

Substituting the value of $[\![e_h']\!]_\eta$ from either (A4) or (B4) and using the assumption $\mathcal{M}^{sc} = \mathcal{M}^c$ from the initial state correspondence we can simplify this to

$$\mathcal{M}^{sc\prime} = \mathcal{M}^c\left\{b_{pb}^{\mathbb{N}} + b_o^{\mathbb{N}} + i \mapsto b[i] \;\Big|\; i < |b|\right\}.$$

By initial state correspondence

$$p = [\![\mathrm{ptr}(pb, e_o)]\!]_\eta = \eta(pb) +_b [\![e_o]\!]_\eta = b_{pb} +_b b_o,$$

It is $b_{pb}^{\mathbb{N}} + b_o^{\mathbb{N}} < 2^N$ both in the first and in the second case above: in the first case it follows from (2) and (A3), in the second case it follows from (2) and (B3). This implies $p^{\mathbb{N}} = b_{pb}^{\mathbb{N}} + b_o^{\mathbb{N}}$. Thus

$$\mathcal{M}^{sc\prime} = \mathcal{M}^c\left\{p^{\mathbb{N}} + i \mapsto b[i] \;\Big|\; i < |b|\right\} = \mathcal{M}^{c\prime}.$$

■

We call a valuation η' *minimal with a property* ϕ iff η' satisfies ϕ and η'_{-x} does not satisfy ϕ for all $x \in \mathrm{dom}(\eta')$.

Lemma 3 *Let η_c, η_c', η, and η' be valuations and s and s' be symbolic states such that s is η-consistent, s' is η'-consistent, and there are transitions $(\eta_c, \mathrm{conc}_\eta(s)) \xrightarrow{l} (\eta_c', \mathrm{conc}_{\eta'}(s'))$ and $s \xrightarrow{\lambda} s'$ with $\lambda \neq \varepsilon$. Assume additionally that η' is a minimal extension of η with the property above. Then for all $P \in$ IML the following is a valid IML transition (fig. 15):*

$$\{(\mathrm{var}(\eta), lP)\} \xrightarrow{l} \{(\mathrm{var}(\eta'), P)\}.$$

□

Proof We prove the lemma by case distinction over all pairs of l and a that can occur.

1. $l =$ `read` b and $\lambda = \mathbf{in}(x)$; by rules (C-In) and (S-In).
   From the correspondence between the symbolic and the concrete transition we obtain $\eta'(x) = b$. Because η' was chosen to be minimal $\eta' = \eta\{x \mapsto b\}$, which also implies $\mathrm{var}(\eta') = \mathrm{var}(\eta)\{x \mapsto b\}$. The lemma follows by rule (I-In).
2. $l =$ `rnd` b and $\lambda = (\nu x[e_l])$; by rules (C-In) and (S-In).
   The correspondence between the symbolic and the concrete transition implies $\mathrm{var}(\eta') = \mathrm{var}(\eta)\{x \mapsto b\}$. Additionally the correspondence yields $|b| = [\![e_l]\!]_\eta^{\mathbb{N}}$, so that the lemma follows by rule (I-Nonce).
3. $l =$ `write` b and $\lambda = \mathbf{out}(e)$; by rules (C-Out) and (S-Out).
   From the correspondence between the symbolic and the concrete transition we obtain $[\![e]\!]_\eta = b$. Additionally $\eta' = \eta$ by minimality of η'. The lemma follows by rule (I-Out).
4. $l =$ `event` b and $\lambda = \mathbf{event}(e)$; by rules (C-Out) and (S-Out).
   The proof is exactly analogous to the case above, the lemma follows by rule (I-Event).
5. $l =$ `ctr` 1 and $\lambda = \mathbf{if}\; e\; \mathbf{then}$ by rules (C-Test) and (S-Test).
   From the correspondence between the symbolic and the concrete transition we obtain $[\![e]\!]_\eta = i1$. Additionally $\eta' = \eta$ by minimality of η'. The lemma follows by rule (I-Cond-True).

■

Lemma 4 *There exists a fixed polynomial p such that for any $P \in CVM$ with $[\![P]\!]_S \neq \bot$*

$$[\![P]\!]_C \lesssim_p [\![[\![P]\!]_S]\!]_I. \qquad \square$$

Proof Let P be a CVM program such that $[\![P]\!]_S \neq \bot$. Let $T = [\![P]\!]_C$ and $\tilde{T} = [\![[\![P]\!]_S]\!]_I$. We shall show that $T \lesssim_p \tilde{T}$ for some polynomial p by giving a relation $\lesssim$ between states of T and $\tilde{T}$ as well as a translation function τ that satisfy definition 4. Let $s_1, \dots, s_n$ be the symbolic execution trace of P with labels $\lambda_1, \dots, \lambda_{n-1}$ and let $\tilde{P}_i = \lambda_i \dots \lambda_{n-1} 0 \in \mathit{IML}$. This way $\tilde{P}_1 = [\![P]\!]_S$ and $\tilde{P}_n = 0$. Let $\mathcal{P}_1, \dots, \mathcal{P}_m$ be protocol states over T such that $\mathcal{P}_1 = \{\varepsilon \mapsto (\eta_1, (\mathtt{Init}, P))\}$ for some initial environment η_1 and there is a transition $\mathcal{P}_i \xrightarrow{(h_i, d_i),\, a_i} \mathcal{P}_{i+1}$ with a command (h_i, d_i) and an action a_i for each i. As CVM does not perform replication, each protocol state will be of the form $\mathcal{P}_i = \{h_i \mapsto (\eta_i, c_i)\}$ for some state c_i and valuation η_i.

No concrete trace of CVM is longer than the symbolic trace (both are bounded by the number of instructions in P), so clearly $m \leq n$. By definition the initial symbolic state $s_1 = (\mathtt{Init}, P)$ is η_1-consistent and $\mathrm{conc}_{\eta_1}(s_1) = c_1$. By setting $\tilde{\eta}_1 = \eta_1$ and repeatedly applying lemma 2 we obtain a sequence $\tilde{\eta}_1, \dots, \tilde{\eta}_m$ of valuations such that for each i the valuation $\tilde{\eta}_i$ is an extension of η_i, the state s_i is $\tilde{\eta}_i$-consistent and $\mathrm{conc}_{\tilde{\eta}_i}(s_i) = c_i$. Additionally we can choose the valuations such that $\tilde{\eta}_{i+1}$ is a minimal extension of $\tilde{\eta}_i$ satisfying the property. For each $i = 1, \dots, m$ we define a protocol state $\tilde{\mathcal{P}}_i$ over $\tilde{T}$ as $\tilde{\mathcal{P}}_i = \{\tilde{h}_i \mapsto (\mathrm{var}(\tilde{\eta}_i), \tilde{P}_i)\}$, where $\tilde{h}_i$ is obtained from h_i as follows: Let $I \subseteq \{1, \dots, n-1\}$ be the set of indices i such that $\tilde{P}_i \neq \tilde{P}_{i+1}$. Given a history h_i of the form $h_i = o_1 1 \dots o_{i-1} 1$ (every CVM rule only has one process on the right hand side, so the replication identifier is always 1) let $\tilde{h}_i = o_{i_1} 1 \dots o_{i_k} 1$, where $\{i_1, \dots, i_k\} = I \cap \{1, \dots, i-1\}$.

Given a protocol state $\mathcal{P}$ over T and a protocol state $\tilde{\mathcal{P}}$ over $\tilde{T}$ we define $\mathcal{P} \lesssim \tilde{\mathcal{P}}$ iff there exist sequences of states $\mathcal{P}_1, \dots, \mathcal{P}_m$ and $\tilde{\mathcal{P}}_1, \dots, \tilde{\mathcal{P}}_m$ as above such that $\mathcal{P} = \mathcal{P}_i$ and $\tilde{\mathcal{P}} = \tilde{\mathcal{P}}_i$ for some i. We define the function τ from commands to sequences of commands as follows:

$$\tau((h,d)) = \begin{cases} (\tilde{h}, d), & \text{if } h = o_1 1 \dots o_{i-1} 1, \text{ and } i \in I, \\ \varepsilon & \text{otherwise.} \end{cases}$$

We now show that the relation $\lesssim$ and the function τ satisfy definition 4, so that $T \lesssim_p \tilde{T}$ for some polynomial p. The conditions in definition 4 are satisfied as follows:

1. Any initial valuation η_1 is not an extended valuation so that $\mathrm{var}(\eta_1) = \eta_1$. By definition
$$\begin{aligned}\{\varepsilon \mapsto (\eta_1, (\mathtt{Init}, P))\} &\lesssim \{\varepsilon \mapsto (\mathrm{var}(\eta_1), \tilde{P}_1)\} \\ &= \{\varepsilon \mapsto (\eta_1, [\![P]\!]_S)\}.\end{aligned}$$

2. Let $\mathcal{P} \lesssim \tilde{\mathcal{P}}$ and assume that there exists a transition $\mathcal{P} \xrightarrow{(h,d),\, a} \mathcal{P}'$. By definition of the relation $\lesssim$ there exist sequences of states $\mathcal{P}_1, \dots, \mathcal{P}_m$ and $\tilde{\mathcal{P}}_1, \dots, \tilde{\mathcal{P}}_m$ as above such that $\mathcal{P} = \mathcal{P}_i$, $\mathcal{P}' = \mathcal{P}_{i+1}$ and $\tilde{\mathcal{P}} = \tilde{\mathcal{P}}_i$ for some $i < m$. It suffices to show that
$$\tilde{\mathcal{P}}_i \xrightarrow{\tau((h,d)),\, a}{}^{*}\, \tilde{\mathcal{P}}_{i+1}. \qquad (*)$$
If $i \in I$ then (*) follows from lemma 3. Let $i \notin I$, that is $\lambda_i = \varepsilon$ in the symbolic execution. Inspecting the proof of lemma 2 we see that then $\mathrm{var}(\tilde{\eta}_i) = \mathrm{var}(\tilde{\eta}_{i+1})$ and so $\tilde{\mathcal{P}}_i = \tilde{\mathcal{P}}_{i+1}$. The program performs no action so that $a = \varepsilon$ and by definition $\tau((h,d)) = \varepsilon$, thus (*) is satisfied.

3. To compute τ it is necessary to know I, but this can be computed by an inspection of P in linear time: it is $i + 1 \in I$ iff the ith instruction in P is one of `In`, `Out`, or `Test`, that is, an instruction that generates a nonempty label λ in the symbolic execution. Thus $\tau(c)$ is computable in time linear in $|c| + |P|$.

4. Assume that for some valuation η and attackers E and $\tilde{E}$ the machine $M = \mathrm{Exec}_\eta(T, E)$ reaches a state $\mathcal{P}$ in t steps and the machine $\tilde{M} = \mathrm{Exec}_\eta(\tilde{T}, \tilde{E})$ reaches a state $\tilde{\mathcal{P}}$ in $\tilde{t}$ steps and $\mathcal{P} \lesssim \tilde{\mathcal{P}}$. If η is the environment of the process in $\mathcal{P}$ and $\tilde{\eta}$ is the environment of the process in $\tilde{\mathcal{P}}$ then $\tilde{\eta}$ is an extension of η, in fact $\tilde{\eta} = \eta$, as both environments get updated by rules (C-In) and (I-Nonce), (I-In) in the same way. It is easy to see that
$$\tilde{t} = O(\tilde{n}_{tr} \cdot (\tilde{t}_e + |[\![P]\!]_S| + |\tilde{\eta}|)),$$
where $\tilde{n}_{tr}$ is the number of transitions performed by $\tilde{M}$ and $\tilde{t}_e$ is the number of steps to evaluate the most expensive IML expression during the execution of $\tilde{M}$. All of these values can be bounded in terms of t as follows: The IML model $[\![P]\!]_S$ performs at most the same number of transitions as the PTS program P, so that $\tilde{n}_{tr} \leq n_{tr} \leq t$, where n_{tr} is the number of transitions executed by M. By construction of the symbolic execution $|[\![P]\!]_S| = O(|P|) = O(t)$. Furthermore $|\tilde{\eta}| = |\eta| \leq t$. Finally we shall prove by induction that if $\tilde{t}_e$ is the number of steps to evaluate $[\![e]\!]_{\tilde{\eta}} = [\![e]\!]_\eta$ for some expression e then $\tilde{t}_e = O(t) \cdot |e|$. Consider the following cases:
   - $e = b$ for some $b \in \mathit{BS}$. In this case $\tilde{t}_e = |e|$.
   - $e = x$ for some $x \in \mathit{Var}$. In this case $[\![e]\!]_\eta = \eta(x)$, so that $\tilde{t}_e \leq t$.
   - $e = \mathit{op}(e_1, \dots, e_n)$ with some $\mathit{op} \in \mathbf{Ops}$. For bitstrings $b_1, \dots, b_n \in \mathit{BS}$ let $t_{op}(b_1, \dots, b_n)$ be the number of steps to evaluate $A_{op}(e_1, \dots, e_n)$ and let $\tilde{t}_i$ be the number of steps to evaluate $[\![e_i]\!]_\eta$. Every operation in e is also performed by M, thus
$$\begin{aligned}\tilde{t}_e &= t_{op}([\![e_1]\!]_\eta, \dots, [\![e_n]\!]_\eta) + \tilde{t}_1 + \dots + \tilde{t}_n \\ &\leq t_{op}([\![e_1]\!]_\eta, \dots, [\![e_n]\!]_\eta) + \sum_i |e_i| \cdot O(t) \\ &\leq O(t) \cdot \left(\sum_i |e_i| + 1 \right) \leq O(t) \cdot |e|\,.\end{aligned}$$
   - $e = e_1 | e_2$. Let $\tilde{t}_1$ and $\tilde{t}_2$ be the number of steps to evaluate $[\![e_1]\!]_\eta$ and $[\![e_2]\!]_\eta$ respectively. Then
$$\begin{aligned}\tilde{t}_e &\leq \tilde{t}_1 + \tilde{t}_2 + |[\![e_1]\!]_\eta| + |[\![e_2]\!]_\eta| \\ &\leq 2 \cdot (\tilde{t}_1 + \tilde{t}_2) \leq O(t) \cdot (|e_1| + |e_2|) \leq O(t) \cdot |e|\,.\end{aligned}$$
   - The cases $e = e'\{e_o, e_l\}$ and $e = \mathrm{len}(e')$ are proved analogously to the case $e = e_1 | e_2$. ∎

Restatement of theorem 1 *There exists a fixed polynomial p such that if $P_1, \dots, P_n$ are CVM processes and for* *each i* $\tilde{P}_i := \llbracket P_i \rrbracket_S \neq \perp$ *then for any IML process* P_E*, any trace property* ρ*, and resource bound* $t \in \mathbb{N}$

$$\begin{aligned}&\mathrm{insec}(\llbracket P_E[P_1, \ldots, P_n] \rrbracket_{CI}, \rho, t)\\ &\qquad \leq \mathrm{insec}(\llbracket P_E[\tilde{P}_1, \ldots, \tilde{P}_n] \rrbracket_I, \rho, p(t)).\end{aligned}$$

Proof By lemma 4 there exists a polynomial p_1 such that $\llbracket P_i \rrbracket_C \lesssim_{p_1} \llbracket \tilde{P}_i \rrbracket_I$ for each i. By lemma 1 the PTS $\llbracket P_E \rrbracket_I$ is a PTS with holes identifiable in p_2-time for some fixed polynomial p_2. Applying definition 7 and theorem 5 we see that there exists a polynomial p_3 depending only on p_1 and p_2 (and thus fixed) such that

$$\begin{aligned}\llbracket P_E[P_1, \ldots, P_n] \rrbracket_{CI} =& \llbracket P_E \rrbracket_I [\llbracket P_1 \rrbracket_C, \ldots, \llbracket P_n \rrbracket_C]\\ \lesssim_{p_3}& \llbracket P_E \rrbracket_I [\llbracket \tilde{P}_1 \rrbracket_I, \ldots, \llbracket \tilde{P}_n \rrbracket_I]\\ =& \llbracket P_E[\tilde{P}_1, \ldots, \tilde{P}_n] \rrbracket_I.\end{aligned}$$

By theorem 4 there exists a polynomial p_4 depending only on p_3 (and thus fixed) such that theorem 1 holds with $p = p_4$.■

G. VERIFICATION OF IML—DETAILS

We show how to simplify IML to the applied pi calculus that can be verified using ProVerif. As ProVerif works in the symbolic model, we shall employ a computational soundness result from [4] to justify its use. The result will guarantee that if ProVerif successfully verifies the translated pi calculus process then the process is asymptotically secure in our computational model. We start by illustrating the method on an example and then give a general description.

The main challenge when translating IML to the pi calculus is that IML processes contain bitstring manipulation primitives that are not valid in pi. An example of such a process is shown in fig. 17—it is an adapted excerpt from an IML model of the Needham-Shroeder-Lowe protocol implementation used in one of our experiments (the full model is shown in appendix H). The key observation is that the bitstring manipulation expressions in IML are most commonly employed to provide the tupling functionality. In our example the process A uses concatenations to construct a computational representation of the pair of n_A and pk_A. Similarly, process B uses range expressions to extract the second element of the pair. The idea of the translation is thus to enrich **Ops** with encoding and parsing operations with meanings given by the bitstring manipulation expressions. This way we hide the direct bitstring manipulation inside new opaque operations. Of course, to obtain a soundness result we need to prove certain properties of the extracted operations to make sure that they correctly implement tupling.

In our example we introduce new operations $conc_1$ and $parse_2$ with implementations given by

$$\begin{aligned}&A_{conc_1}(b_1, b_2) = \llbracket \text{"msg1"} |\, \mathrm{len}(b_1) | b_1 | b_2 \rrbracket,\\ &A_{parse_2}(b) =\\ &\quad \text{if } \llbracket \neg(b\{i4, iN\} +_b iN +_b i4 \leq \mathrm{len}(b)) \rrbracket \text{ then } \perp \text{ else}\\ &\quad \text{if } \llbracket \neg(b\{i0, i4\} = \text{"msg1"}) \rrbracket \text{ then } \perp \text{ else}\\ &\quad \llbracket b\{i4 +_b iN +_b b\{i4, iN\},\\ &\qquad \mathrm{len}(b) -_b i4 -_b iN -_b b\{i4, iN\}\} \rrbracket.\end{aligned}$$

In the implementation of the parsing expression we keep all the condition checks that are performed by the IML process before applying the parser. We follow the convention of IML that $i1$ and $i0$ represent truth values of bitstrings. Using the new operations we can simplify our example IML process to the pi calculus process shown in fig. 18, removing the if-statements that have been absorbed into the implementation of $parse_2$.

```
A =
  $(\tilde{\nu}\, n_A)$; $(\tilde{\nu}\, r)$;
  let $m_1$ = "msg1"| len($n_A$)|$n_A$|$pk_A$ in
  let $e_1$ = $encrypt(pk_X, m_1)$ in
  out($e_1$); ...

B =
  in($e_1$);
  let $m_1$ = $decrypt(sk_B, e_1)$ in
  if $m_1\{i4, iN\} +_b iN +_b i4 \leq$ len($m_1$) then
  if $m_1\{i0, i4\}$ = "msg1" then
  let $x_1 = m_1\{i4 +_b iN +_b m_1\{i4, iN\}$,
            len($m_1$) $-_b i4 -_b iN -_b m_1\{i4, iN\}\}$ in
  if $x_1 = pk_X$ then ...
```

Figure 17: An excerpt from the IML process for the NSL protocol. An expression len(...) **produces a result of fixed length** iN**.**

```
A =
  $(\tilde{\nu}\, n_A)$; $(\tilde{\nu}\, r)$;
  out($encrypt(pk_X, conc_1(n_A, pk_A))$); ...

B =
  in($e_1$);
  let $m_1$ = $decrypt(sk_B, e_1)$ in
  let $x_1$ = $parse_2(m_1)$ in
  if $x_1 = pk_X$ then ...
```

Figure 18: An excerpt from the pi calculus translation for the NSL protocol.

The syntax of the applied pi calculus is shown in fig. 19. It is a strict subset of the IML syntax with the following differences:

- The bitstring operations are no longer available.
- The only allowed form of the restriction operator is $(\tilde{\nu} x)$ with the same meaning as described in section 4.
- Parameters of events are restricted to be fixed bitstrings. This is a limitation of the result in [4].
- The conditional expression of IML with truth meanings for bitstrings $i0$ and $i1$ is no longer available. Instead we can use let expressions to conditionally choose based on equality of bitstrings by assuming that there exists an operation $eq \in$ **Ops** such that $A_{eq}(b, b) = b$ and $A_{eq}(b, b') = \perp$ for all $b \neq b'$.
- The input and output expressions only accept variables as parameters—all computations must be performed in let-expressions.

The calculus shown in fig. 19 is a restricted version of the pi calculus presented in [4], as we do not need the full generality used there. Our restrictions are as follows:

- There is only one public communication channel.
- We do not make a distinction between variables and names, as they behave identically for the purpose of the computational execution.

$b \in BS$, $x \in Var$, $op \in \mathbf{Ops}$

$e \in PExp ::=$	expression
x	variable
$op(e_1, \dots, e_n)$	constructor/destructor
$P, Q ::=$	process
0	nil
$!P$	replication
$P \vert Q$	parallel composition
$(\tilde{\nu} x);\ P$	randomness
$\mathbf{in}(x);\ P$	input
$\mathbf{out}(x);\ P$	output
$\mathbf{event}(b);\ P$	event
$\mathbf{let}\ x = e\ \mathbf{in}\ P\ [\mathbf{else}\ Q]$	evaluation

Figure 19: The syntax of the applied pi calculus.

$$[\![x]\!]^k_\eta = \eta(x), \text{ for } x \in Var,$$
$$[\![op(e_1, \dots, e_n)]\!]^k_\eta = \tilde{A}_{op}(k, [\![e_1]\!]^k_\eta, \dots, [\![e_n]\!]^k_\eta).$$

Figure 20: The evaluation of pi expressions, whereby $\bot$ propagates.

- We only allow computations in let-expressions, so that we do not make a distinction between constructors and destructors in the syntax.

Unlike CVM and IML which execute with regards to a fixed security parameter k_0 introduced in section 2, the computational semantics of the applied pi calculus is parameterised by a security parameter. In order to achieve that we assume that the operations in **Ops** possess a *generalised implementation* $\tilde{A}$ such that $\tilde{A}_{op} \colon \mathbb{N} \times BS^{\mathrm{ar}(op)} \rightharpoonup BS$ is the implementation of an operation $op \in \mathbf{Ops}$ that takes the security parameter as the first argument. For a security parameter k and inputs $\underline{m}$ the value $\tilde{A}_{op}(k, \underline{m})$ should be computable in time polynomial in $k + |\underline{m}|$. We require that $\tilde{A}_{op}(k_0, \cdot) = A_{op}$ for each $op \in \mathbf{Ops}$.

The semantics of the pi calculus is directly derived from the semantics of IML. Given a pi process P and a security parameter k, we define the semantics $[\![P]\!]^k_\pi$ as follows: The expression evaluation uses $\tilde{A}$ instead of A as shown in fig. 20. The semantics rules are obtained from the IML rules (fig. 15) by substituting all expression evaluations $[\![e]\!]_\eta$ with $[\![e]\!]^k_\eta$. The syntactic form $(\tilde{\nu} x)$ behaves as described in section 4, but now it is not a syntactic sugar anymore, so we add a new semantic rule shown in fig. 21.

$$\frac{b \in BS, \quad |b| = k, \quad r = \tilde{A}_{nonce}(k, b)}{(\eta,\ (\tilde{\nu} x); P) \xrightarrow{\mathtt{rnd}\ b} \{(\eta\{x \mapsto r\},\ P)\}}. \qquad \text{(pi-Nonce)}$$

Figure 21: Randomness generation in pi calculus.

We now give details regarding the translation procedure from IML to pi. In the following we shall assume that the IML processes do not contain else-branches; this is true for the processes produced by the symbolic execution. Removing if-statements from such processes does not reduce the set of traces and thus does not reduce insecurity. We shall therefore divide all if-statements into two groups: the *cryptographic* statements, that are likely to be relevant for the security of the process and should be kept in the translation, and the *auxiliary* statements that can be removed from the process without affecting security. The exact choice does not affect the soundness of the approach, but removing too many if statements might make the resulting pi process insecure, and removing too few may prevent the successful translation from IML to pi. We use the following heuristic: an if-statement is considered to be cryptographic iff it is of the form **if** $e_1 = e_2$ **then** P, where both e_1 and e_2 are variables or applications of cryptographic operations.

Given an IML process P we perform on it the following operations:

- Introduce intermediate let-statements so that all out-statements only contain variables, all cryptographic if-statements are of the form **if** $x_1 = x_2$ **then** P with variables x_1 and x_2 and every expression in the new let statements is of one of three types:
  - an *encoding expression*, that is, an expression containing only concrete bitstrings, len(), concatenations, arithmetic operations, and variables,
  - a *parsing expression*, that is, an expression containing only concrete bitstrings, len(), substring extraction, arithmetic operations, and a single variable,
  - a *cryptographic expression*, that is, an expression containing only variables and cryptographic operations.

  As an example, the IML processes in fig. 17 are already written in such a form.

- For each subprocess $P' = (\mathbf{let}\ y = e\ \mathbf{in}\ P'')$, where e is an encoding expression with variables $x_1, \dots, x_n$, add a new *encoding operation* c of arity n to **Ops** with the implementation given by

  $$A_c(b_1, \dots, b_n) = [\![e[b_1/x_1, \dots, b_n/x_n]]\!].$$

  Now substitute P' by **let** $y = c(x_1, \dots, x_n)$ **in** P''.

  In order to justify modelling the encoding operations as tuples symbolically, we need to check that their computational implementations fulfil certain conditions. The first condition is:

  (C1) the ranges of the functions A_c introduced above are disjoint.

  Checking the side conditions is described in appendix G.1.

- For each subprocess $P' = (\mathbf{let}\ y = e\ \mathbf{in}\ P'')$, where e is a parsing expression with a variable x, add a new *parsing operation* p of arity 1 to **Ops**. We need to check that before computing e the process P makes sure that x contains a result of a suitable encoding operation. More specifically, we check that there exists an encoding operation c such that the process rejects any x with the value outside the range of A_c and such that e computes an inverse of A_c. Let $e_1, \dots, e_n$ be expressions such that P contains an auxiliary if-statement of the form **if** e_i **then** ... above P' for some i. Let $x_1, \ldots, x_m$ be the variables of P with exception of x and let

$$\phi_p = \exists x_1, \ldots, x_m \colon e_1 \wedge \ldots \wedge e_n.$$

This way, whenever (η', P') is an executing process in a protocol state reached by $[\![P]\!]_I$ from some environment η, we have $[\![\phi_p]\!]_{\eta'} = i1$. We check the following conditions:

(C2) there exists an encoding operation c such that for every b not in the range of A_c it is $[\![\phi_p[b/x]]\!] = i0$. We say that c *matches* p,

(C3) the function $f_p \colon b \mapsto [\![e[b/x]]\!]$ is an ith inverse of A_c for some i, that is, $f_p(A_c(b_1, \ldots, b_n)) = b_i$ where n is the arity of c.

Appendix G.1 shows how to check the conditions (C1)–(C3) and how a successful check results in a quantifier-free formula ϕ'_p with x as the only variable such that ϕ_p implies ϕ'_p and the condition (C2) is still satisfied with ϕ'_p. Additionally ϕ'_p satisfies

(C4) for the encoding operation c that matches p and any b in the range of A_c it is $[\![\phi_p[b/x]]\!] = i1$.

We define the computational implementation for p as

$$A_p(b) = \text{if } [\![\phi'_p[b/x]]\!] \text{ then } [\![e[b/x]]\!] \text{ else } \bot$$

and substitute P' by **let** $y = p(x)$ **in** P''.

- Remove all auxiliary if-statements: for every such statement replace **if** e **then** P' by P'. Translate all cryptographic if-statements into the form expected by the pi-calculus: replace every occurrence of **if** $x_1 = x_2$ **then** P by **let** $\_ = eq(x_1, x_2)$ **in** P.

If the process P does not contain any else-branches and the above procedure yields a valid pi process $\tilde{P}$ then we say that *P is translatable to $\tilde{P}$*. A complete example of an IML program and its resulting pi calculus translation for the NSL protocol is shown in appendix H.

In order to obtain the computational semantics for the translated process, we need to specify the generalised implementations $\tilde{A}_c$ and $\tilde{A}_p$ for the newly introduced encoders and parsers. We can assume *any* generalisation of these operations to arbitrary security parameters that satisfies the conditions (C1)–(C4).

Clearly the translation preserves all the action sequences of the original process so the following holds:

Lemma 5 *There exists a fixed polynomial p such that for any IML process P translatable to a pi process $\tilde{P}$*

$$[\![P]\!]_I \lesssim_p [\![\tilde{P}]\!]_\pi^{k_0}. \qquad \square$$

Applying theorem 4 we obtain a statement that links the security of the pi translation to the security of the original IML process:

Restatement of theorem 2 *There exists a fixed polynomial p such that for any IML process P translatable to a pi process $\tilde{P}$, any trace property ρ and resource bound $t \in \mathbb{N}$*

$$\text{insec}([\![P]\!]_I, \rho, t) \leq \text{insec}([\![\tilde{P}]\!]_\pi^{k_0}, \rho, p(t)).$$

Now that we have translated IML to pi, we can enumerate the conditions under which the resulting pi process can be soundly verified using ProVerif. For this purpose we shall make use of a computational soundness result from [4], which places restrictions on the operation set **Ops** as well as on the shape of the pi process. More specifically, the computational soundness theorem is proved there for the set of constructors $\mathbf{C} = \{E/3, ek/1, dk/1, pair/2\}$ and destructors $\mathbf{D} = \{D/2, isenc/1, isek/1, ekof/1, fst/1, snd/1, eq/2\}$. The result includes soundness for signatures, but we omit them as they have not been used in our experiments so far. For simplicity the result presented here uses only one pairing construct (as in [4]), but it can be easily extended to an arbitrary number of tupling constructors and destructors, to correspond to our encoding and parsing operations introduced during the translation from IML. The symbolic behaviour of the operations is defined by the following equations:

$$\begin{aligned} D(dk(t_1), E(ek(t_1), m, t_2)) &= m, \\ isenc(E(ek(t_1), t_2, t_3)) &= E(ek(t_1), t_2, t_3), \\ isek(ek(t)) &= ek(t), \\ ekof(E(ek(t_1), m, t_2)) &= ek(t_1), \\ fst(pair(x, y)) &= x, \\ snd(pair(x, y)) &= y, \\ eq(x, x) &= x. \end{aligned}$$

Let $\mathbf{Ops}^S = \mathbf{C} \cup \mathbf{D} \cup \{nonce\}$. The *soundness conditions* that the implementations $\tilde{A}_x$ for $x \in \mathbf{Ops}^S$ need to satisfy are as follows:

1. There are disjoint and efficiently computable sets of bitstrings representing the types nonces, ciphertexts, encryption keys, decryption keys, and pairs. Let $Nonces_k$ denote the set of all nonces for a security parameter k.
2. Given $b \in BS$ with $|b| = k$ chosen uniformly at random, $\tilde{A}_{nonce}(k, b)$ returns $r \in Nonces_k$ uniformly at random.
3. The functions $\tilde{A}_E$, $\tilde{A}_{ek}$, $\tilde{A}_{dk}$, and $\tilde{A}_{pair}$ are length-regular—the length of their result depends only on the lengths of their parameters. All $m \in Nonces_k$ have the same length.
4. Every image of $\tilde{A}_E$ is of type ciphertext, every image of $\tilde{A}_{ek}$ and $\tilde{A}_{ekof}$ is of type encryption key, every image of $\tilde{A}_{dk}$ is of type decryption key.
5. For all $m_1, m_2 \in BS$ we have $\tilde{A}_{fst}(\tilde{A}_{pair}(m_1, m_2)) = m_1$ and $\tilde{A}_{snd}(\tilde{A}_{pair}(m_1, m_2)) = m_2$. Every m of type pair is in the range of $\tilde{A}_{pair}$. If m is not of type pair, $\tilde{A}_{fst}(m) = \tilde{A}_{snd}(m) = \bot$.
6. $\tilde{A}_{ekof}(\tilde{A}_E(p, x, y)) = p$ for all p of type encryption key, $x \in BS$, and a nonce y. $\tilde{A}_{ekof}(e) \neq \bot$ for any e of type ciphertext and $\tilde{A}_{ekof}(e) = \bot$ for any e that is not of type ciphertext.
7. $\tilde{A}_E(p, m, y) = \bot$ if p is not of type encryption key.
8. $\tilde{A}_D(\tilde{A}_{dk}(r), m) = \bot$ if $r \in Nonces_k$ and $\tilde{A}_{ekof}(m) \neq \tilde{A}_{ek}(r)$.
9. $\tilde{A}_D(\tilde{A}_{dk}(r), \tilde{A}_E(\tilde{A}_{ek}(r), m, r')) = m$ for all $r, r' \in Nonces_k$.
10. $\tilde{A}_{isek}(x) = x$ for any x of type encryption key. $\tilde{A}_{isek}(x) = \bot$ for any x not of type encryption key.
11. $\tilde{A}_{isenc}(x) = x$ for any x of type ciphertext. $\tilde{A}_{isenc}(x) = \bot$ for any x not of type ciphertext.

$$
\begin{aligned}
m,\, n &::= x \mid \mathit{pair}(m,n)\\
e &::= m \ \mid \mathit{isek}(e) \mid \mathit{isenc}(e) \mid D(x_d, e) \mid \mathit{fst}(e)\\
&\quad \mid \mathit{snd}(e) \mid \mathit{ekof}(e) \mid \mathit{eq}(e,e)\\
P,\, Q &::= \mathbf{out}(x);\ P \mid \mathbf{in}(x);\ P \mid 0 \mid\ !P \mid (P|Q) \mid (\tilde{\nu}x);\ P\\
&\quad \mid \mathbf{let}\ x = e\ \mathbf{in}\ P\ [\mathbf{else}\ Q] \mid \mathbf{event}(b);\ P\\
&\quad \mid (\tilde{\nu}r);\ \mathbf{let}\ x = ek(r)\ \mathbf{in}\ \mathbf{let}\ x_d = dk(r)\ \mathbf{in}\ P\\
&\quad \mid (\tilde{\nu}r);\ \mathbf{let}\ x = E(\mathit{isek}(D_1), D_2, r)\ \mathbf{in}\ P\ [\mathbf{else}\ Q]
\end{aligned}
$$

Figure 22: The syntax of key-safe processes.

12. We define an encryption scheme $(KeyGen, Enc, Dec)$ as follows: $KeyGen$ picks a random r in $Nonces_k$ and returns $(\tilde{A}_{ek}(r), \tilde{A}_{dk}(r))$. $Enc(p, m)$ picks a random r in $Nonces_k$ and returns $\tilde{A}_E(p, m, r)$. $Dec(k, c)$ returns $\tilde{A}_D(k, c)$. We require that the defined encryption scheme is IND-CCA secure.
13. For all e of type encryption key and $m \in BS$ the probability that $\tilde{A}_E(e, m, r) = \tilde{A}_E(e, m, r')$ for uniformly chosen $r, r' \in Nonces_k$ is negligible.

The conditions on the pairing operations follow from the conditions (C1)–(C4) checked during the translation (length-regularity is fulfilled for any function given by an IML encoding expression), the other conditions (in particular that the encryption is IND-CCA) shall be assumed, because we are treating cryptographic operations as black boxes and not trying to verify them. The condition that all functions have disjoint ranges is quite restrictive and is unlikely to be fulfilled in actual implementations. For this reason in future we would like to use CryptoVerif to verify our models, to bypass the need for complex soundness conditions.

The soundness result of [4] is proved for a class of the so-called *key-safe* processes. In a nutshell, key-safe processes always use fresh randomness for encryption and key generation and only use honestly generated (that is, through key generation) decryption keys for decryption. Decryption keys may not be sent around (in particular, this avoids the key-cycle problems). The grammar of key-safe processes is summarised in fig. 22. We let x, x_d, k_s, and r stand for different sets of variables: general purpose, decryption key, signing key, and randomness variables.

Lemma 6 (Computational soundness [4]) *If a closed key-safe process symbolically satisfies a trace property ρ then it computationally satisfies ρ.* □

We now proceed to sketching out the proof of theorem 3 from section 7. For a process P let $\mathbf{Ops}_P$ be the set of operations used by P (including the *nonce* operation). The symbolic semantics and security of pi are defined in [4]. We do not detail the semantics here, as we only need to know that it is exactly the semantics that is used by ProVerif.

A function $f\colon \mathbb{N} \to \mathbb{R}$ is called *negligible* if for every $c \in \mathbb{N}$ there exists $n_0 \in \mathbb{N}$ such that $f(n) < 1/n^c$ for all $n > n_0$.

Restatement of theorem 3 *Let P be a pi process such that $\mathbf{Ops}_P \subseteq \mathbf{Ops}^S$ and the soundness conditions are satisfied. If P is key-safe and symbolically secure with respect to a trace property ρ then for every polynomial p the following function is negligible in k:*

$$\mathrm{insec}([\![P]\!]^k_\pi,\, \rho,\, p(k)).$$

```
A =
  $(\tilde{\nu}\, n_A);\ (\tilde{\nu}\, r);$
  $\mathbf{let}\ m_1 = \text{"msg1"}|\,\mathrm{len}(n_A)|n_A|pk_A\ \mathbf{in}$
  $\mathbf{let}\ e_1 = \mathit{encrypt}(pk_X, m_1)\ \mathbf{in}$
  $\mathbf{out}(e_1);\ \ldots$

B =
  $\mathbf{in}(e_1);$
  $\mathbf{let}\ m_1 = \mathit{decrypt}(sk_B, e_1)\ \mathbf{in}$
  $\mathbf{if}\ \mathrm{len}(pk_X) +_b iN +_b i20 +_b i4 = \mathrm{len}(m_1)\ \mathbf{then}$
  $\mathbf{if}\ m_1\{i0, i4\} = \text{"msg1"}\ \mathbf{then}$
  $\mathbf{if}\ m_1\{i4, iN\} = i20\ \mathbf{then}$
  $\mathbf{let}\ x_1 = m_1\{i4 +_b iN +_b m_1\{i4, iN\},$
              $\mathrm{len}(m_1) -_b i4 -_b iN -_b m_1\{i4, iN\}\}\ \mathbf{in}$
  $\mathbf{if}\ x_1 = pk_X\ \mathbf{then}\ \ldots$
```

Figure 23: An excerpt from the IML process for the NSL protocol (full version).

The main issue in the proof is to relate the notion of computational execution in [4] (their definition 18) to our notion of computational execution (definition 2). Both definitions are very similar. In [4] the state of the protocol consists of a single executing process together with valuations for variables in the process. In each step the attacker chooses an execution context to specify which subprocess of the complete process is supposed to perform a reduction. In our definition the attacker interacts with a multiset of processes, selecting the process to be executed by an attached handle. It is easy to see that both definitions of the security game are equivalent.

G.1 Parsing Conditions

We show how we check conditions (C1)–(C4) arising during the translation from IML to pi. The checks we perform are by no means complete (we might fail to detect that the conditions actually hold), but they are suitable for the protocols that we encountered so far. We shall use the excerpt from the IML process of the NSL protocol shown in fig. 23 as an example (fig. 17 contained a slightly simplified version).

For each encoding operation c and parsing operation p let e_c and e_p be the IML expressions that they replace. Let ϕ_p represent the set of facts that the IML process establishes before applying e_p, as described previously.

To prove (C1) we check that all encoding expressions e_c contain a concrete bitstring (a tag) at the same positions and that all tags are different. In the example of fig. 23 the bitstring "msg1" would be such a tag, and we would expect other messages to contain tags like "msg2", "msg3", etc.

To prove (C3) for an encoder c and a parser p we check that $\mathrm{simplify}_{\Sigma_{op}}(e_p[e_c/x]) = x_i$, where x is the variable of e_p and x_i is one of the variables of e_c. As an example, for the operations $conc_1$ and $parse_2$ introduced at the beginning of appendix G,

$$
\begin{aligned}
e_{conc_1} &= \text{"msg1"}|\,\mathrm{len}(x_1)|x_1|x_2,\\
e_{parse_2} &= x\{i4 +_b iN +_b x\{i4, iN\},\\
&\qquad \mathrm{len}(x) -_b i4 -_b iN -_b x\{i4, iN\}\}.
\end{aligned}
$$

Substituting e_{conc_1} for x in e_{parse_2} we obtain an expression that simplifies to x_2, thus we know that e_{parse_2} computes the second inverse of e_{conc_1}.

Given a parser p and a candidate encoder c, we check whether c matches p (C2) as follows: first check that e_c

is a concatenation of expressions, each of which is either a variable (a concatenation parameter), a length of a variable, or a constant expression. Formally e_c is required to be of a form $e_1|\ldots|e_n$, where $\{1,\ldots,n\} = I_x \cup I_l \cup I_t$ such that for all $i \in I_x$ it is $e_i = x_i$ for some variable x_i, for all $i \in I_l$ it is $e_i = \text{len}(x_j)$ for some $j \in I_x$ and for all $i \in I_t$ it is $e_i = b_i$ for some constant bitstring b_i. We require that all variables and length expressions are distinct (no variable repeats twice) and that $|I_x| = |I_l| + 1$, that is, the expression e_c contains lengths for all parameters except one—the missing length can then be derived from knowing the total length of the concatenation.

Given a bitstring b, in order to check that b is in the range of A_c, it is sufficient to check all the constant (tag) fields and to check that the sum of the length fields is consistent with the actual length of b. The following makes this precise.

Given a parsing expression p_i, we say that p_i *extracts the ith field from* e_c if the following holds: for an expression e let $e_c[e/e_i]$ be the expression obtained from e_c by substituting e_i with e. Then for a fresh variable x'

$$\text{simplify}_\Sigma(p_i[e_c[x'/e_i]/x]) = x',$$
$$\text{where } \Sigma = \Sigma_{op} \cup \{\text{len}(x') = \text{getLen}(e_i)\}.$$

Theorem 6 *Let c and p be an encoding and a parsing expression such that e_c is of a form $e_1|\ldots|e_n$ with $\{1,\ldots,n\} = I_x \cup I_l \cup I_t$ as described above. Assume that for each $i \in I_l \cup I_t$ the formula ϕ_p contains a parsing expression p_i as a term, such that p_i extracts the ith field from e_c. Let*

$$\phi_{tag} = \bigwedge_{i \in I_t} p_i = b_i,$$
$$\phi_{len} = \sum_{i \in I_l} p_i + \sum_{i \in I_t \cup I_l} \text{getLen}(e_i) \leq \text{len}(x).$$

Then a bitstring b is in the range of A_c iff

$$[\![\phi_{tag} \wedge \phi_{len}]\!]_{x \mapsto b} = i1. \qquad \square$$

Proof (sketch) Let $b \in BS$ satisfy the premises of the theorem. For each $i \leq n$ we obtain the length $l_i \in \mathbb{N}$ of the ith field in b as follows: for each $i \in I_l$ such that $e_i = \text{len}(x_j)$ for some $j \in I_x$ let $l_j = [\![p_i[b/x]]\!]^{\mathbb{N}}$. For each $i \in I_l \cup I_t$ let $l_i = [\![\text{getLen}(e_i)]\!]^{\mathbb{N}}$. For the single $i \in I_x$ such that $\text{len}(x_i)$ is not one of the fields of e_c let $l_i = |b| - \sum_{j \neq i} l_j$. Knowing the lengths allows us to split b into fields as follows: for each $i \leq n$ let $b_i = b\{\sum_{j=1}^{i-1} l_j,\, l_i\}$. This is well-defined according to ϕ_{len}. Clearly $b = b_1|\ldots|b_n$. We show that for each i it is $b_i = [\![e_i[b_j/x_j|j \in I_x]]\!]$ as follows.

- If $i \in I_x$ then $e_i = x_i$ and the equality holds trivially.
- If $i \in I_l$ then $e_i = \text{len}(x_j)$ for some $j \in I_x$. By construction $b_i = \text{bs}(l_i) = \text{bs}(|b_j|)$.
- If $i \in I_t$ then the equality follows from ϕ_{tag}.

Overall we have shown that $b = [\![e_c[b_j/x_j|j \in I_x]]\!]$, so that b is in the range of A_c. ■

Thus checking (C2) reduces to finding appropriate parsers p_i among the terms of ϕ_p and checking that $\phi_p \vdash \phi_{tag} \wedge \phi_{len}$. Furthermore, by choosing $\phi'_p = \phi_{tag} \wedge \phi_{len}$, we obtain a quantifier-free formula that satisfies (C2) and (C4), as required by the translation.

As an example, we can show that (C2) holds for $conc_1$ and $parse_2$ with respect to fig. 23 as follows: the conditions checked by the process B contain references to parsing expressions $m_1\{i0, i4\}$ and $m_1\{i4, iN\}$. We check that the first expressions extracts the first field (the tag) from e_{conc_1} and the second expression extracts the second field (the length of the first parameter). We then observe that the conditions checked by B imply

$$\phi_{tag} = (m_1\{i0, i4\} = \text{"msg1"}),$$
$$\phi_{len} = (iN +_b m_1\{i4, iN\} +_b i4 \leq \text{len}(m_1)).$$

Thus both the tag and the length consistency are properly checked.

Our implementation currently checks all the conditions automatically except $\phi_p \vdash \phi_{len}$. The reason is that we are planning to use CryptoVerif as a verification backend and expect to be able to relax the parsing conditions there.

H. NSL EXAMPLE CODE

We show all the stages of the verification of the NSL example, discussed in section 8

H.1 Client Source

The source code of the client is shown below. In our example N = `sizeof(size_t)` = 8 and k_0 corresponds to `SIZE_NONCE`, which is set to be 20.

```
#include <net.h>
#include <lib.h>

#include <proxies/common.h>

#include <string.h>
#include <stdio.h>

// #define LOWE_ATTACK

int main(int argc, char ** argv)
{
  unsigned char * pkey, * skey, * xkey;
  size_t pkey_len, skey_len, xkey_len;

  unsigned char * m1, * m1_all;
  unsigned char * Na;
  size_t m1_len, m1_e_len, m1_all_len;

  unsigned char * m2, * m2_e;
  unsigned char * xNb;
  size_t m2_len, m2_e_len;
  size_t m2_l1, m2_l2;

  unsigned char * m3_e;
  size_t m3_e_len;

  unsigned char * p;

  // for encryption tags
  unsigned char * etag = malloc(4);

  BIO * bio = socket_connect();

  pkey = get_pkey(&pkey_len, 'A');
  skey = get_skey(&skey_len, 'A');
  xkey = get_xkey(&xkey_len, 'A');

  /* Send message 1 */

  m1_len = SIZE_NONCE + 4 + pkey_len
      + sizeof(size_t);
  p = m1 = malloc(m1_len);
```

```
memcpy(p, "msg1", 4);
p += 4;
* (size_t *) p = SIZE_NONCE;
p += sizeof(size_t);
Na = p;
nonce(Na);
p += SIZE_NONCE;
memcpy(p, pkey, pkey_len);

m1_e_len = encrypt_len(xkey, xkey_len,
                       m1, m1_len);
m1_all_len = m1_e_len + sizeof(size_t) + 4;
m1_all = malloc(m1_all_len);
memcpy(m1_all, "encr", 4);
m1_e_len =
    encrypt(xkey, xkey_len, m1,
            m1_len,
            m1_all + sizeof(m1_e_len) + 4);
m1_all_len = m1_e_len + sizeof(size_t) + 4;
* (size_t *) (m1_all + 4) = m1_e_len;

send(bio, m1_all, m1_all_len);

/* Receive message 2 */

recv(bio, etag, 4);
recv(bio, (unsigned char*) &m2_e_len,
     sizeof(m2_e_len));
m2_e = malloc(m2_e_len);
recv(bio, m2_e, m2_e_len);

m2_len = decrypt_len(skey, skey_len,
                     m2_e, m2_e_len);
m2 = malloc(m2_len);
m2_len =
    decrypt(skey, skey_len,
            m2_e, m2_e_len, m2);

if(xkey_len + 2 * SIZE_NONCE
    + 2 * sizeof(size_t) + 4 != m2_len)
{
  printf("A:␣m2␣has␣wrong␣length\n");
  exit(1);
}

if(memcmp(m2, "msg2", 4))
{
  printf("A:␣m2␣not␣properly␣tagged\n");
  exit(1);
}

m2_l1 = *(size_t *) (m2 + 4);
m2_l2 = *(size_t *) (m2 + 4 + sizeof(size_t));

if(m2_l1 != SIZE_NONCE)
{
  printf("A:␣m2␣has␣wrong␣length␣for␣xNa\n");
  exit(1);
}

if(m2_l2 != SIZE_NONCE)
{
  printf("A:␣m2␣has␣wrong␣length␣for␣xNb\n");
  exit(1);
}

if(memcmp(m2 + 4 + 2 * sizeof(size_t),
          Na, m2_l1))
{
  printf("A:␣xNa␣in␣m2␣doesn't␣match␣Na\n");
  exit(1);
}
```

```
#ifndef LOWE_ATTACK
  if(memcmp(m2 + m2_l1 + m2_l2
            + 2 * sizeof(size_t) + 4,
            xkey, xkey_len))
  {
    printf("A:␣x_xkey␣in␣m2␣doesn't␣match␣xkey\n");
    exit(1);
  }
#endif

  xNb = m2 + m2_l1 + 2 * sizeof(size_t) + 4;

  /* Send message 3 */

  m3_e_len = encrypt_len(xkey, xkey_len,
                         xNb, m2_l2);
  m3_e = malloc(m3_e_len + sizeof(size_t) + 4);
  memcpy(m3_e, "encr", 4);
  m3_e_len =
      encrypt(xkey, xkey_len, xNb,
              m2_l2,
              m3_e + sizeof(m3_e_len) + 4);
  * (size_t *)(m3_e + 4) = m3_e_len;

  send(bio, m3_e,
       m3_e_len + sizeof(m3_e_len) + 4);

  return 0;
}
```

H.2 Proxy Functions

We show examples of proxy functions that replace calls to `nonce`, `encrypt`, etc. in the symbolic execution. Each function starts by calling the actual function that it replaces so that the concrete execution can proceed as usual—recall that we observe a run of the program in order to identify the main path. The proxy functions then call the special *symbolic interface* functions to create new symbolic values and place them in memory. These symbolic interface functions are interpreted specially by the symbolic execution and perform the following actions:

- **load_buf**(**const unsigned char** * buf,
          size_t len, **const char** * hint)

  Retrieves from memory the expression located at `buf` of length `len` and places it on the stack. The value `hint` is attached to the expression for naming purposes. For instance, the names of variables in the IML model shown in appendix H.3 are derived from hints.

- **store_buf**(**const unsigned char** * buf)

  Takes an expression from the stack and stores it in the location in memory pointed to by `buf`.

- **symL**(**const char** * sym, **const char** * hint,
        size_t len, **int** deterministic)

  Applies the operation `sym` to all the expressions on the stack as parameters. Sets the length of the new expression to be equal to `len`. The last parameter can be used to specify that the application is non-deterministic, that is, conceptually it takes an extra random argument, without having to specify that argument explicitly. Calls to this function are also used to model random variable generation. For instance, the symbol *nonce* created in `nonce_proxy` is treated specially and translates to the ν operator of IML.

-
  ```
  symN(const char * sym, const char * hint,
       size_t * len, int deterministic)
  ```

  Behaves like `symL`, but instead of assigning a known length to the new expression e, keeps its length unrestricted and writes len(e) into `len`.

The proxy functions are trusted to represent the true behaviour of the actual cryptographic operations. For instance, the function `encrypt` is supposed to check the well-formedness of the key (corresponding to the symbolic operation *isek*). The actual cryptographic functions are required to satisfy the conditions listed in appendix G for the soundness result to hold.

```
void nonce_proxy(unsigned char * N)
{
  nonce(N);

  symL(”nonce”, ”nonce”, SIZE_NONCE, FALSE);
  store_buf(N);
}

size_t encrypt_len_proxy(unsigned char * key,
                         size_t keylen,
                         unsigned char * in,
                         size_t inlen)
{
  size_t ret =
    encrypt_len(key, keylen, in, inlen);

  symL(”encrypt_len”, ”len”, sizeof(ret), FALSE);
  store_buf(&ret);

  if(ret < 0) exit(1);

  return ret;
}

size_t encrypt_proxy(unsigned char * key,
                     size_t keylen,
                     unsigned char * in,
                     size_t inlen,
                     unsigned char * out)
{
  size_t ret =
    encrypt(key, keylen, in, inlen, out);

  unsigned char nonce[SIZE_NONCE];

  nonce_proxy(nonce);

  load_buf(key, keylen, ”key”);
  symN(”isek”, ”key”, NULL, TRUE);
  load_buf(in, inlen, ”msg”);
  load_buf(nonce, SIZE_NONCE, ”nonce”);
  symN(”E”, ”cipher”, &ret, TRUE);
  store_buf(out);

  if(ret > encrypt_len_proxy(key, keylen,
                             in, inlen))
    fail(”encrypt_proxy:␣bad␣length”);

  return ret;
}

unsigned char * get_pkey_proxy(size_t * len,
                               char side)
{
  unsigned char * ret = get_pkey(len, side);

  char name[] = ”pkX”;
  name[2] = side;

  readenv(ret, len, name);

  return ret;
}
```

H.3 IML Model

The IML model extracted from both the client and the server is shown below. The notation $e\langle l\rangle$ is a shorthand for "e such that len(e) = l". For instance, **in**(c, var1<8>); means **in**(c, var1); **if** len(var1) = 8 **then**.

The model contains several castToInt expressions. These result from the fact that the implementation uses `size_t` as the length type, but the OpenSSL functions that we call use `int`. These type conversions are recorded during the symbolic execution. For now we assume no numeric overflows, as mentioned in section 8, so the casts are removed before translating to pi.

```
let A =
new nonce1<20>;
let msg1 = 6d736731|i20|nonce1|pkA in
new nonce2<20>;
let cipher1 = E(isek(pkX), msg1, nonce2) in
let msg2 = 656e6372|len(cipher1)<8>|cipher1 in
out(c, msg2);
in(c, msg3<8>);
let var1 = (msg3 castToInt TSBase(int ))
                castToInt TSBase(unsigned long ) in
in(c, msg4<var1>);
let msg5 = D(skA, msg4) in
if len(pkX)<8> + i40 + i16 + i4 = len(msg5)<8> then
if msg5{0, 4} = 6d736732 then
if msg5{4, 8} = i20 then
if msg5{12, 8} = i20 then
let var2 = msg5{20, msg5{4, 8}} in
if var2 = nonce1 then
let var3 =
  msg5{msg5{4, 8} + msg5{12, 8} + i16 + i4,
      len(msg5) − (msg5{4, 8} + msg5{12, 8} + i16 + i4)} in
if var3 = pkX then
let msg6 = msg5{msg5{4, 8} + i16 + i4, msg5{12, 8}} in
new nonce3<20>;
let cipher2 = E(isek(pkX), msg6, nonce3) in
let msg7 = 656e6372|len(cipher2)<8>|cipher2 in
out(c, msg7); 0.

let B =
in(c, msg1<8>);
let var1 = (msg1 castToInt TSBase(int ))
                castToInt TSBase(unsigned long ) in
in(c, msg2<var1>);
let msg3 = D(skB, msg2) in
if len(pkX)<8> + i8 + i20 + i4 = len(msg3)<8> then
if msg3{0, 4} = 6d736731 then
if msg3{4, 8} = i20 then
let var2 = msg3{i8 + msg3{4, 8} + i4,
               len(msg3) − (i8 + msg3{4, 8} + i4)} in
if var2 = pkX then
let var3 = msg3{4, 8} in
let var4 = msg3{12, msg3{4, 8}} in
new nonce1<20>;
let msg4 = 6d736732|var3|i20|var4|nonce1|pkB in
new nonce2<20>;
let cipher1 = E(isek(pkX), msg4, nonce2) in
let msg5 = 656e6372|len(cipher1)<8>|cipher1 in
out(c, msg5);
```

```
in(c, msg6<8>);
let var5 = (msg6 castToInt TSBase(int ))
                    castToInt TSBase(unsigned long ) in
in(c, msg7<var5>);
let msg8 = D(skB, msg7) in
if len(msg8)<8> = i20 then
if msg8 = nonce1 then
event endB(); 0.
```

H.4 ProVerif Model

The ProVerif model resulting from the translation of the IML process is shown below. The processes A and B as well as the symbolic rules for the new encoding and parsing expressions $conc_i$ and $parse_i$ are generated automatically from the source IML process. The rules for encryption and decryption, the query, and the environment process (including A' and B') are specified by hand.

The events are used without parameters—this is a limitation of the result in [4], but our symbolic execution as well as ProVerif can easily deal with parameterised events. The modelling is similar to [13, 4]. There the client A' executes an event *beginA*() only if it is supposed to talk to B and B' executes an event *endB*() only if it supposed to talk to A. The event *endB*() is executed at the end, so conceptually B' needs to execute

if pkX = pkA **then** B; **event** endB(). **else** B.

Unfortunately, B; **event** endB(). does not form a valid process, so we use an equivalent formulation using an event *notA*() instead—*endB*() is always executed, but it is counted only if *notA*() has not been executed.

The meaning of if-statements in pi is different from their meaning in IML. A pi calculus statement **if** $e_1 = e_2$ **then** P corresponds to the IML **let** _ = $eq(e_1, e_2)$ **in** P.

```
free c.
fun ek/1.
fun dk/1.
fun E/3.
reduc
  D(dk(a), E(ek(a), x, r)) = x.
reduc
  isek (ek(a)) = ek(a).


data conc2/2.

data conc5/3.

data conc11/1.

reduc
  parse2(conc5(x0, x1, x2)) = x0.
reduc
  parse3(conc5(x0, x1, x2)) = x2.
reduc
  parse4(conc5(x0, x1, x2)) = x1.
reduc
  parse6(conc2(x0, x1)) = x1.
reduc
  parse7(conc2(x0, x1)) = x0.

query
  ev:endB() ==> ev:beginA() | ev:notA().

query
  ev:endB() ==> ev:notA().

let A =
new nonce1;
new nonce2;
let var1 =
  conc11(E(isek(pkX), conc2(nonce1, pkA), nonce2)) in
out(c, var1);
in(c, msg1);
in(c, var2);
let var3 = parse2(D(skA, var2)) in
if var3 = nonce1 then
let var4 = parse3(D(skA, var2)) in
if var4 = pkX then
new nonce3;
let var5 =
  conc11(E(isek(pkX), parse4(D(skA, var2)), nonce3)) in
out(c, var5); 0.

let B =
in(c, msg2);
in(c, var27);
let var28 = parse6(D(skB, var27)) in
if var28 = pkX then
new nonce4;
new nonce5;
let var29 =
  conc11(E(isek(pkX),
           conc5(parse7(D(skB, var27)), nonce4, pkB),
           nonce5)) in
out(c, var29);
in(c, msg3);
in(c, var30);
let var31 = D(skB, var30) in
if var31 = nonce4 then
event endB(); 0.


let A' =
  in(c, pkX);

  if pkX = pkB then
  event beginA(); A
  else A.

let B' =
  in(c, pkX);

  if pkX = pkA then B else
  event notA(); B.

process
  !
  new A; new B;
  let pkA = ek(A) in
  let skA = dk(A) in
  let pkB = ek(B) in
  let skB = dk(B) in
  out(c, pkA); out(c, pkB);
  (! A' | ! B')
```