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 [14], CryptoVerif [13] or AVISPA [4]. 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 [27] and ASPIER [19], 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

```
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.

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.

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 [3].

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 transition system, to a computational security definition in the spirit of [16, 26, 42]. 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 [20]. 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 [31] 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 [18] or [25]) 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 [5] 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 [35] 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

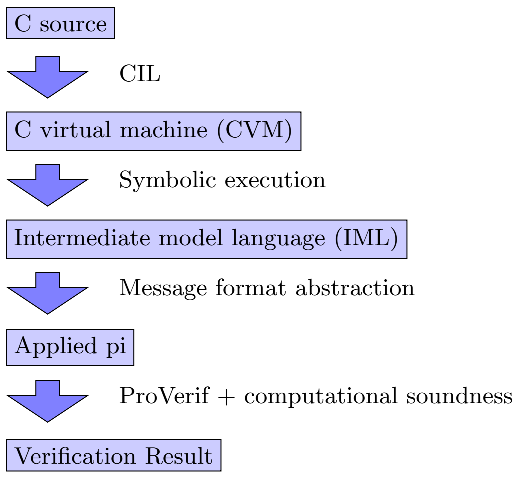
C source
CIL
C virtual machine (CVM)
Symbolic execution
Intermediate model language (IML)
Message format abstraction
Applied pi
ProVerif + computational soundness
Verification Result

Figure 2: An outline of the method

syntax and semantics of IML in section 4 and describe the symbolic execution in section 6.

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 $P_E[P_1, \ldots, P_n]$ 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 $P_1, \ldots, P_n$ are CVM processes and $\tilde{P}_1, \ldots, \tilde{P}_n$ are IML models resulting from their symbolic execution then for any environment process P_E the security of $P_E[\tilde{P}_1, \ldots, \tilde{P}_n]$ with respect to a trace property ρ relates to the security of $P_E[P_1, \ldots, P_n]$ 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 [15] and augmented with events). We can then apply a computational soundness result, such as the one from [5], 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 $P_1, \ldots, P_n$ 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 $P_1, \ldots, P_n$ 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 $P_1, \ldots, P_n$ 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 [38]. 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 [37]) 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 [27], 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 [19] 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 [20] report an extension of the KLEE test-generation tool [18] 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 [20] treats every byte in a memory buffer separately and thus only supports buffers of fixed length.

Finally, an online technical report includes more details and proofs of all results stated in this paper [2].

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 write $f(x) = \bot$ for $x \in X$ to indicate that f is not defined on x.

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 the endianness. Even though these functions capture an un-

$b \in BS$, $v \in Var$, $op \in$ **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` 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.

signed 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, $i10 = \text{bs}(10)$.

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.

The transitions between semantic configurations are labelled with protocol actions such as reading or writing values from the attacker or a random number generator, or raising events. The executing program can also read values from an environment η (a mapping from variables to bitstrings). We give an informal overview of the semantics of CVM; the details are in the technical report [2]. Before the program is executed, each referenced variable v is allocated an address $\text{addr}(v)$ 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 $\text{bs}(\text{addr}(v))$ on the stack.
- `Malloc` takes a value s from the stack, reads a value p from the attacker, and if the range $\{\text{val}(p)\}_{\text{val}(s)}$ 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 $\{\text{val}(p)\}_{\text{val}(l)}$ 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 read` or `In rnd` takes a value l from the stack. `In read` reads a value of length $\text{val}(l)$ from the attacker and `In rnd` requests a random value of length $\text{val}(l)$. The value is then placed on the stack.
- `Env` v places $\eta(v)$ and $\text{bs}(|\eta(v)|)$ on the stack
- `Apply` op with $\text{ar}(op) = n$ 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 $\text{val}(p)$.

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 the technical report [2] and is used to prove soundness of other verification steps. We believe that work on proving correctness of the CompCert compiler [32] 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 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:

```
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.

```
  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 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, 15]. 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 [5].

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:

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

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 technical report [2] describes the detailed semantics of IML, in our formalism of protocol transition systems, introduced in the next section.

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

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

Figure 5: The syntax of IML.

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

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

5. SECURITY OF PROTOCOLS

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 essentially 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 of the transitions capture replication. Full details are in the technical report [2].

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. At each step the attacker 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 $\text{Events}(T, E, t)$ 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 $\text{insec}(T, \rho, t)$ be the probability*

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

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

Intuitively $\text{insec}(T, \rho, t)$ 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 $P_1, \ldots, P_n$ we write $P_E[P_1, \ldots, P_n]$ to denote process P_E where each hole $[]_i$ is replaced by P_i.

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:

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

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 $\text{insec}([\![P_E[P_1, P_2]]\!]_{CI}, \rho, t)$ 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 CML 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.

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

Figure 7: Symbolic expressions.

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 $\text{ptr}(pb, e_o)$ 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 $\text{val}(a) = \text{val}(b)$ 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:

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

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 by the function apply, defined as follows:

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

The function receives an operation identifier and a number of expressions and decides how to apply the operation symbolically.

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

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

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

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

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

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

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

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

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

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

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

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

Figure 8: The symbolic execution of CVM.

$<$ 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* [22], 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:

$$\begin{aligned}
&\forall x, y, a\colon\ \text{len}(x) = a \wedge \text{len}(y) = a \Rightarrow \text{len}(x +_b y) = a, \\
&\forall x\colon\ \text{len}(\mathit{sha1}(x)) = i20.
\end{aligned}$$

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

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

The simplifications employed in our algorithm are described in the technical report [2].

For a partial function f we write def(f) to denote the set of all x such that $f(x) \neq \bot$ and write $f\{x \mapsto a\}$ to update functions. The algorithm for symbolic execution is determined by the set of semantic rules presented in fig. 8. A semantic configuration has the form $(\Sigma,\, \mathcal{A}^s,\, \mathcal{M}^s,\, \mathcal{S}^s,\, P)$, where

- Σ is a fact set,
- $\mathcal{A}^s\colon \mathit{PBase} \rightharpoonup \mathit{SExp}$ 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 def$(\mathcal{M}^s) = $ def$(\mathcal{A}^s)$,

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

- $\mathcal{S}^s$ is a list of symbolic expressions representing the execution stack,
- $P \in CVM$ is the executing program.

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 (`Init`, P). If L ends in a state with an empty program, let $l_1, \ldots, l_n$ be the sequence of labels of L and set $[\![P]\!]_S = l_1 \ldots l_n 0 \in IML$, 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 $t \in \mathbb{N}$:*

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

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 the technical report [2].

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)`. 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 ptr(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. The technical report [2] 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 $k_0 \in \mathbb{N}$. We define a translation function $\alpha(P)$ from IML to pi processes and call a process P *translatable* when $\alpha(P) \neq \bot$.

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

Backes et al. [5] 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 $P_1, \dots, P_n$ be implementations of protocol participants in CVM and let P_E be an IML process that describes an execution environment. Assume that $P_1, \dots, P_n$ 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 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 $P_1, \dots, P_n$ 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 [17].

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.

	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.

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 [5]), 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 [9] 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 (modulo the assumption that the encryption used is indeed IND-CCA). The implementation is designed to satisfy the soundness conditions of the computational soundness result outlined in the technical report [2]. Masking the second participant's identity check triggers Lowe's attack [33] as expected.

The CSur example is the code analysed in a predecessor paper on C verification [27]. 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 `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 [27], 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 [38] 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

```
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.

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 [39], 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 [41], so we could not use ProVerif to verify the protocol, but we are investigating the use of CryptoVerif for this purpose.

9. RELATED WORK

[27] 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. [19] 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 abstracted away. Also, significant parts of the code are abstracted away manually. In comparison, our method applies directly to C code including pointers and thus requires less manual effort. [29] 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. [40] 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, [20] 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 [20] treats every byte in a memory buffer separately and thus only supports buffers of fixed length. [21] 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 [11] and FS2CV translate F# to the process calculi which can be verified by the tools ProVerif [12] and CryptoVerif [13] versus symbolic and computational models, respectively. They have been applied to an implementation of TLS [10]. The refinement-type checker F7 [9] verifies security properties of F# programs versus a Dolev-Yao attacker. Under certain conditions, this has been shown to be provably computationally sound [7, 24]. [34] 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. [7] gives results on computational soundness of symbolic analysis of programs in the concurrent lambda calculus RCF. [6] reports on a type system for verifying crypto-protocol implementations in RCF. With respect to Java, [30] 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). [36] provides an approach for translating Java implementations into formal models in the LySa process calculus in order to perform a security verification. [28] presents an application of the ESC/Java2 static verifier to check conformance of JavaCard applications to protocol models. [23] describes verification of cryptographic primitives implemented in a functional language Cryptol. CertiCrypt [8] 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] M. Aizatulin, A. D. Gordon, and J. Jürjens. Extracting and verifying cryptographic models from C protocol code by symbolic execution (long version). Available at `http://users.mct.open.ac.uk/ma4962/files/paper-full.pdf`, 2011.

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

[4] 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.

[5] 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.

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

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

[8] 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.

[9] 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.

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

[11] 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.

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

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

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

[15] 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.

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

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

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

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

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

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

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

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

[34] 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.

[35] 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.

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

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

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

[39] 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.

[40] 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.

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

[42] 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.

```
//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 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 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 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.