Efficient Symbolic Execution for Analysing Cryptographic Protocol Implementations

Ricardo Corin[1,2] and Felipe Andrés Manzano[1]

[1] FaMAF, Universidad Nacional de Córdoba, Argentina [2] CONICET

Abstract. The analysis of code that uses cryptographic primitives is unfeasible with current state-of-the-art symbolic execution tools. We develop an extension that overcomes this limitation by treating certain concrete functions, like cryptographic primitives, as *symbolic functions* whose execution analysis is entirely avoided; their behaviour is in turn modelled formally via rewriting rules. Our code runs in a (simplified) LLVM virtual machine. We develop concrete and symbolic semantics for our LLVM, and we show our approach sound by proving operational correspondence between the two semantics. We present a prototype to illustrate our approach with several (sequential code) examples, and we discuss next milestones towards the symbolic analysis of fully concurrent cryptographic protocol implementations.

1 Introduction

Despite a large amount of research devoted to the formal analysis cryptographic protocols (see e.g. [9, 6, 3, 4, 19, 11]), the thorough analysis of existing industrial protocol implementations (like e.g. OpenSSL) remains largely unexplored. The difficulty is that exploring real low-level protocol code is much more complex than the formal models that are fed to protocol analyzers. For instance, implementations need to deal with memory manipulation, bitarray arithmetics, and intricate message formatting, all of which are abstracted away in formal models.

Previous research in this area focuses on extracting and verifying formal models from implementation code. This includes the work of Goubault [11], where protocols written in C are abstracted into horn-clauses. Another direction is taken by Bhargavan et al. [2], which develops an implementation of TLS [8] coded in ML. In this case, the implementation is interoperable with other industrial implementations and can be analyzed by automatically extracting and verifying its corresponding formal model (using Proverif [3] and Cryptoverif [4]). Still, these tools are not applicable to legacy or binary code. Some quite recent works attempt to address this; see e.g. the roadmap in [1] for extracting formal models from C legacy code (or earlier work for Java [12]); other work [17] takes the inverse approach: generating monitors from formal models that prevent invalid or malicious messages forbidden by the protocol specification.

Recently, methods based on symbolic execution (developed initially by [13]) have been proved to scale very well to real life non-protocol code. Here, the code

is dynamically explored through all its branches, searching implementation bugs like memory manipulation errors. In order to avoid trying the entire (possibly infinite) input space, program inputs are assumed to be symbolic variables that remain uninstantiated (but become constrained) at execution time. Whenever the program reaches a conditional branch that operates on symbolic terms, the execution is forked; each branch is taken, and the condition (or its negation) is added to a store of path conditions. A bitarray SMT solver (STP [10]) is used to detect infeasible branches. Additionally, whenever a program reaches an insecure state (e.g., memory corruption), STP can also be used to generate a counterexample by providing apropriate values for the (symbolic) inputs. Tools like Bitblaze [18], Catchconv [16] and KLEE [5] have been succesfully used to find bugs in real code like gnuutils. KLEE has the added advantage of working on top of the LLVM [15], a virtual machine that is becoming popular for its efficiency and flexibility. There exist efficient compilers that outputs LLVM code (e.g., gcc and clang). LLVM code can be efficiently interpreted in several architectures, and has several powerful optimizations that benefit from its type-rich framework. LLVM code is closer to machine code, and lower-level than a language like C; thus, performing analysis at this level is more realistic and can catch compiler-introduced bugs.

Our aim is to develop a symbolic execution analysis that can be applied to code that uses cryptographic primitives, like security protocols. Existing symbolic execution tools (like KLEE mentioned above) can not deal with such code, as the search space size blows up when it explores the insides of cryptographic functions. In this work, we propose an extension that avoids such difficulty by preventing the exploration of certain functions (e.g., cryptographic primitives), which we call "symbolic functions". In order to specify the behaviour of symbolic functions and to allow the execution to progress, symbolic functions are endowed with rewriting rules that detail abstractly their properties (e.g., that decryption inverts encryption). Replacing concrete cryptographic primitives with symbolic functions thus models ideal cryptography in the style of Dolev-Yao [9].

We aim for fully formal results, and so start by rigourously formalising the LLVM language and its semantics. In summary, our net contributions are:

1. A formalization of the LLVM, with its syntax and concrete, ideal semantics (Section 2). (This is a necessity for our general aim, but it already constitutes a useful result on its own, and we are not aware of the existence of such semantics besides the informal language reference of [15]; for instance, this can be used to formally prove correct many of the LLVM complex transformations or optimizations, like e.g. the aliasing and constant propagation transformations.)
2. A formalization of symbolic execution (in the style of KLEE) within our LLVM language (Section 2.4). We develop an alternative symbolic semantics, and formally prove that symbolic execution is sound, in the sense that symbolic executions correspond to concrete ones (Lemma 1 in Section 2).
3. Symbolic functions (Section 3). We introduce the concept of symbolic functions that abstract away concrete functions we are not interested on analyze.

Not every concrete function can be turned into symbolic, so we identify the necessary conditions. Then, we give a new semantic rule that enables the symbolic execution of a symbolic function.

4. Efficient code analysis and Rewriting rules (Section 3.3). The behaviour of symbolic functions is given in terms of rewriting rules that specify how different symbolic terms may be rewritten to simpler terms, so that standard symbolic execution can progress. This results in efficient symbolic analysis that would be impossible to achieve otherwise.
5. Soundness (Section 3.4). Under this extension we show again operational correspondence between the semantics with symbolic functions and the concrete ones (Theorem 1).
6. Prototype (Section 4 and 4.1). We develop a prototype coded as a non-intrusive patch of KLEE, and use it to illustrate our method with several examples.

The project website [7] contains a longer version of this document, including complete semantic rules, detailed explanations and proofs, as well as source code for the prototpye and examples.

2 LLVM

To the best of our knowledge, the syntax and semantics of LLVM have not been formalized yet, besides the informal language reference given at the LLVM website [15]. Thus, we start by formalizing the LLVM language and provide both concrete and symbolic operational semantics; for simplicity, we leave out some details and focus on the core LLVM instructions that capture most expressivity.

2.1 Syntax

We illustrate LLVM code by means of an example. (A larger version of this document contains the full LLVM language grammar, which for space limitations we omit here.)

```
i8 dec () {
   i8 c = inp();
   if (c <= 0)
     return 0;
   return c-1;
}
```

```
define i8 @dec() {
entry:
    %c = call i8 @inp()
    %cond = icmp slet i8 %c, 0
    br i1 %cond, label %true_b, label %false_b
true_b:
    ret i8 0
false_b:
    %res = sub i8 %c, 1
    ret i8 %res
}
```

On the left we show a small C program with its corresponding LLVM code on the right. The code decreases an input by one while it is still positive, otherwise returns zero. The LLVM function @dec returns an integer of one byte (type $i8$), and has three *basic blocks*, each tagged with a label: $entry$, $true_b$, and $false_b$. Identifiers starting with @ are globally accessible, while identifiers starting with % are local bindings. This code uses instruction *call* to input a byte using the *inp* function[1], then stores in $\%cond$ the result of comparing the input byte $\%c$ to 0, in which case the execution branches (using the *br* instruction) to the true or false label; in the former, the input is decreased by one. Execution finishes in both cases with the return instruction *ret*.

Of course, the LLVM language contains many more instructions than the ones show in this example; the interested reader can refer to the (informal) language description [15]. For instance, there are memory manipulation instructions (ALLOCA, LOAD, STORE), and both arithmetic and floating-point instructions. LLVM types can either be primitive or composed; type information is heavily used by the different (aggressive) optimizations (e.g., aliasing and constant propagation). A special bitcast instruction is used for casting values from one type to another. In our formalization, we use type information to calculate the length in bytes of a given value of type t (denoted by $size(t)$) so that we can store or retrieve it from memory.

2.2 Semantics

We provide two different semantics: one concrete and one symbolic. The former is *ideal* and intuitive, although difficult to reason with. The latter allows for symbolic execution, discussed in Section 2.4, and enables efficient path exploration. We show an operational correspondence between both semantics in Lemma 1.

Concrete Contexts. LLVM specifies two kinds of storage locations: bindings (containing infinite registers, either global or local to a function), and memory.

The bindings can be either global or local: local bindings $\mathcal{L}$ maps identifiers starting with '%' to typed values, for instance '%cond' to $(i8, 0)$; global bindings $\mathcal{G}$ maps identifiers starting with @ to typed values; for instance '@dec' maps to a memory location with the first instruction of function @dec.

We model memory as a partial map $\mathcal{M}$ from addresses (that range from 0 to some number K) to bytes. The allocation and memory release is modelled by enlarging and shrinking the domain of $\mathcal{M}$, respectively.

The execution state is represented by a *context*, defined as a tuple $\langle pc, \mathcal{M}, \mathcal{G}, fs\rangle$. Here, the program counter *pc* is an address (s.t. $\mathcal{M}(pc)$ holds the first byte of the next instruction to be executed). The stack of function frames *fs* is a sequence of *function frames*, where each frame $(id, \mathcal{L}, ret, \mathcal{A})$ contains all the data locally needed in context of a function call: *id* is the local binding name from the calling function where to put the result of the current function in execution; $\mathcal{L}$ are the local bindings, and *ret* is the return address where to continue executing after

[1] *inp* is defined by us to simplify the management of interactive inputs

this function returns. Finally, $\mathcal{A}$ is a set of memory addresses reserved during the execution of this function.

Initial context. We assume an initial context $\langle pc_0, \mathcal{M}_0, \mathcal{G}_0, (id_0, \mathcal{L}_0, ret_0, \emptyset)\rangle$ in which the operating system has loaded the LLVM code into memory $\mathcal{M}_0$, and populated the bindings appropiately. pc_0 is the memory address of the first instruction of a *@main* function. id_0 and ret_0 are addresses (their values are irrelevant since our semantics does not model an operating system; executions simply get stuck at the last step and no value is returned). $\mathcal{L}_0$ contains the parameters to the *@main* function.

2.3 Concrete semantics

Table 1 shows concrete semantic rules. All the operations on bitarrays refer to actual, concrete operations. Since we store code in memory, we need to parse its opcode to know which instruction we are executing. This is modelled by the auxiliary function $op_{\mathcal{M}}(pc)$.

Rules (SUB) and (ICMP) perform simple bitarray arithmetics. In the former, two operators op_2 is subtracted from op_2 and the result is placed in binding id. (Here, auxiliary function $v(x, \mathcal{L})$ evaluates to the pair $(type, value)$ defined in $\mathcal{G}$ or $\mathcal{L}$.) In the latter, id is stored with a one bit that is either set when a comparison (given by `cond`) holds or not for both operators. Rule (CALL) is used to jump to a subroutine. We set as next program counter the starting address $addr$ of f, and create a new function frame containing the argument parameters (id_i mapping to v_i), and as return address $nxt_{\mathcal{M}}(pc)$. Rule (RET) returns to the calling function: it pops the function frame, jumps to address ret, and frees the used local memory $\mathcal{A}$.

Rule for input (INP) takes a byte b (of type $i8$) and stores in the local binding id. This rule is non-deterministic and effectively forks computation in 256 ways, one for each possible input value. Rules (BRT) and (BRF) are used for branches; the former is when the condition c holds (i.e. it is exactly equal to a set bit), and the latter is for the false branch. Rules (ALLOCA), (LOAD), (STORE), are used for memory manipulation; here, functions *pck* and *unpack* store expressions into memory bytes (we use this to hide implementation details like representation order little or big endian). (ALLOCA) finds a pointer p in memory where to allocate the chunk of memory. (LOAD) looks up for the memory chunk starting in x_p, and unpacking it and returning it in v_p (here, $unpck(t, \mathcal{M}(\{x_p\}_{size(t)})$ is the result of unpacking and interpreting the memory chunk $\{x_p, \ldots, x_p{+}size(t)\}$. (STORE) makes the inverse operation using *pack*.

Concrete execution traces. The semantic rules define a step relation $\longrightarrow$. For convenience, we add a label l to $\longrightarrow$ and write $\xrightarrow{l}$, where l is the instruction execution (i.e., $l = op_{\mathcal{M}}(pc)$). A sequence of (formal) labels is a trace tr, and we define $c \xrightarrow{tr}{}^{*} c' \doteq \exists c_1, \ldots, c_n : c \xrightarrow{l_1} c_1 \to \ldots \xrightarrow{l_n} c_n \to c'$ with $tr = l_1 \ldots l_n$.

$$\frac{op_{\mathcal{M}}(pc) = id = \texttt{add}\ \ t\ op_1, op_2 \qquad v(op_1, \mathcal{L}) = (t, x_{op_1}) \qquad v(op_2, \mathcal{L}) = (t, x_{op_2})}{\begin{array}{c}\langle pc, \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs\rangle \longrightarrow \\ \langle nxt_{\mathcal{M}}(pc), \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}\{id \to (t, x_{op_1} -_t x_{op_2})\}, ret, \mathcal{A}) :: fs\rangle\end{array}}\texttt{SUB}$$

$$\frac{\begin{array}{c}op_{\mathcal{M}}(pc) = id = \texttt{icmp}\ \ \texttt{cond}\ \ t\ op_1, op_2 \\ v(op_1, \mathcal{L}) = (t, x_{op_1}) \qquad v(op_2, \mathcal{L}) = (t, x_{op_2}) \qquad \mathcal{L}' = \mathcal{L}\{id \to (\mathbf{i1}, x_{op_1}\ \texttt{cond}\ {}_t x_{op_2})\}\end{array}}{\langle pc, \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs\rangle \longrightarrow \langle nxt_{\mathcal{M}}(pc), \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}', ret, \mathcal{A}) :: fs\rangle}\texttt{ICMP}$$

$$\frac{\begin{array}{c}op_{\mathcal{M}}(pc) = id = \texttt{call}\ f(a_0, a_1, \ldots a_n) \\ \mathcal{G}(f) = (t(t_0 id_0, t_1 id_1 \ldots t_n id_n), fp) \qquad \forall k : v(a_k, \mathcal{L}) = (t_k, v_k)\end{array}}{\langle pc, \mathcal{M}, \mathcal{G}, fs\rangle \longrightarrow \langle fp, \mathcal{M}, \mathcal{G}, ((t, id), \{id_0 \to v_0, \ldots id_n \to v_n\}, nxt_{\mathcal{M}}(pc), \emptyset) :: fs\rangle}\texttt{CALL}$$

$$\frac{\begin{array}{c}op_{\mathcal{M}}(pc) = \texttt{ret}\ t\ rslt \\ fs = ((t, id), \mathcal{L}, ret, \mathcal{A}) :: ((t_1, id_1), \mathcal{L}_1, ret_1, \mathcal{A}_1) :: fs' \qquad v(rslt, \mathcal{L}) = (t, x_p)\end{array}}{\langle pc, \mathcal{M}, \mathcal{G}, fs\rangle \longrightarrow \langle ret, \mathcal{M} - \{\mathcal{A} \times *\}, \mathcal{G}, (id_1, \mathcal{L}_1\{id \to (t, x_p)\}, ret_1, \mathcal{A}_1) :: fs'\rangle}\texttt{RET}$$

$$\frac{op_{\mathcal{M}}(pc) = id = \texttt{inp}() \qquad b\ byte\ in\ \mathcal{L} \qquad \mathcal{L}' = \mathcal{L}\{id \to (\mathbf{i8}, b)\}}{\langle pc, \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs\rangle \longrightarrow \langle nxt_{\mathcal{M}}(pc), \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}', ret, \mathcal{A}) :: fs\rangle}\texttt{INP}$$

$$\frac{op_{\mathcal{M}}(pc) = \texttt{br}\ c\ \texttt{labelt}\ l_1\ \texttt{labelt}\ l_2 \qquad \mathcal{L}(l_1) = (\mathbf{label}, bb) \qquad v(c, \mathcal{L}) = (\mathbf{i1}, 1)}{\langle pc, \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs\rangle \longrightarrow \langle bb, \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs\rangle}\texttt{BRT}$$

$$\frac{op_{\mathcal{M}}(pc) = \texttt{br}\ c\ \texttt{labelt}\ l_1\ \texttt{labelt}\ l_2 \qquad \mathcal{L}(l_2) = (\mathbf{label}, bb) \qquad v(c, \mathcal{L}) = (\mathbf{i1}, 0)}{\langle pc, \mathcal{M}, \mathcal{G}, fs\rangle \longrightarrow \langle bb, \mathcal{M}, \mathcal{G}, fs\rangle}\texttt{BRF}$$

$$\frac{\begin{array}{c}op_{\mathcal{M}}(pc) = id = \texttt{alloca}\ t, i32\ n \qquad \exists p : \mathcal{A}' - \mathcal{A} = dom(\mathcal{M}' - \mathcal{M}) = \{p\}_{size(t)\times n} \\ p + size(t) \times n < K \qquad \mathcal{L}' = \mathcal{L}\{id \to (t*, p)\}\end{array}}{\langle pc, \mathcal{M}, \mathcal{G}, (res, \mathcal{L}, ret, \mathcal{A}) :: fs\rangle \longrightarrow \langle nxt_{\mathcal{M}}(pc), \mathcal{M}', \mathcal{G}, (res, \mathcal{L}', ret, \mathcal{A}') :: fs\rangle}\texttt{ALLOCA}$$

$$\frac{\begin{array}{c}op_{\mathcal{M}}(pc) = id = \texttt{load}\ t* p \\ v(p, \mathcal{L}) = (t, x_p) \qquad \{x_p\}_{size(t)} \subseteq dom(\mathcal{M}) \qquad v_p = unpck(t, \mathcal{M}(\{x_p\}_{size(t)}))\end{array}}{\langle pc, \mathcal{M}, \mathcal{G}, (id, \mathcal{L}, ret, \mathcal{A}) :: fs\rangle \longrightarrow \langle nxt_{\mathcal{M}}(pc), \mathcal{M}, \mathcal{G}, (id, \mathcal{L}\{id \to (t, v_p)\}, ret, \mathcal{A}) :: fs\rangle}\texttt{LOAD}$$

$$\frac{\begin{array}{c}op_{\mathcal{M}}(pc) = \texttt{store}\ t\ val, t* p \\ v(p, \mathcal{L}) = (t, x_p) \qquad v(val, \mathcal{L}) = (t, x_{val}) \qquad \{x_p\}_{size(t)} \subseteq dom(\mathcal{M})\end{array}}{\langle pc, \mathcal{M}, \mathcal{G}, fs\rangle \longrightarrow \langle nxt_{\mathcal{M}}(pc), \mathcal{M}\{x_p \to pck(t, x_{val})\}, \mathcal{G}, fs\rangle}\texttt{STORE}$$

Table 1. Concrete semantic rules (selected)

2.4 Symbolic semantics and symbolic execution

Although the concrete semantics is intuitive and easy to understand, it is not easy to implement and to analyze on top of it (e.g. due to the non-deterministic behaviour of rule (INP) and the existential quantifiers in some of the rule preconditions like (ALLOCA)).

Here we consider an alternative *symbolic* semantics, that uses symbolic expressions. These expressions are either simple constants or symbolic variables ι; more complex expressions are formed by applying operators op_t of a type t.

Definition 1. *Symbolic expressions and conditions are given by:*

$$\begin{array}{ll} op_t & ::= +_t, -_t, /_t, *_t \\ exp & ::= (t, \iota) \mid (t, const) \mid exp\ op_t\ exp \mid exp.n \\ cop_t & ::= =_t, <_t, >_t, \&_t, |_t \\ cond & ::= exp\ cop_t\ exp \end{array}$$

Expressions are either basic typed constants $(t, const)$ or symbolic variables (t, ι), or formed by applying arithmetic operators op_t. Finally, for an expression exp which is stored in memory as a sequence of bytes, we use projector $exp.n$ to project the nth byte. Condition expressions are formed by two expressions and one conditional operator cop_t. We assume a natural semantics to conditional expressions so we can decide whether a conditional expression $c\sigma$ holds or not, under a substitution σ that maps symbolic variables to concrete bitarrays.

We use symbolic expressions and conditions to carry out the symbolic execution of an LLVM program. To this end we provide an alternative *symbolic* semantics. The first is to add a *constraint store* Σ to contexts: $\langle pc, \mathcal{M}, \mathcal{G}, fs, \Sigma \rangle$ where Σ is a sequence of symbolic conditions that we use to record the path conditions taken in each branch during execution. We also change the range of $\mathcal{M}$ and $\mathcal{L}$: they now map addresses and identifiers to symbolic expressions, and not concrete values.

Definition 2 (Satisfiability of Σ). *We write $\vdash \Sigma$ when there exists a substitution σ s.t. every condition $c\sigma$ holds for every $c \in \Sigma$.*

In practice, this is decided by an bitarray SMT solver (STP [10] in our case).

Symbolic semantics Besides changing the range of $\mathcal{M}$ and $\mathcal{L}$, all the previous arithmetic rules now deal with expressions, not concrete operations. Rule (ADD), for instance, has operation $+_t$ not being the concrete one but the symbolic expression operator.

Table 2 shows the new symbolic semantic rules wrt the previous concrete. Rule (SINP) now returns a fresh symbolic ι variable (instead of non-deterministacally choosing a byte). Rules for branching now (SBRT) and (SBRF) now are both applicable, as long as the (updated) constraint store Σ' is solvable; this effectively models the forking of each execution branch.

$$\frac{op_{\mathcal{M}}(pc) = id = \texttt{inp}() \qquad \iota \textit{ fresh in } \mathcal{L} \qquad \mathcal{L}' = \mathcal{L}\{id \to (\texttt{i8}, \iota)\}}{\langle pc, \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs, \Sigma\rangle \longrightarrow \langle nxt_{\mathcal{M}}(pc), \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}', ret, \mathcal{A}) :: fs, \Sigma\rangle}\texttt{SINP}$$

$$\frac{\begin{array}{c} op_{\mathcal{M}}(pc) = \texttt{ br } c \texttt{ labelt } l_1 \texttt{ labelt } l_2 \\ \mathcal{G}(l_1) = (\textbf{label}, bb) \qquad \Sigma' = \Sigma :: v(c, \mathcal{L}) =_{\textbf{i1}} (\textbf{i1}, 1) \qquad \vdash \Sigma' \end{array}}{\langle pc, \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs, \Sigma\rangle \longrightarrow \langle bb, \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs, \Sigma'\rangle}\texttt{SBRT}$$

$$\frac{\begin{array}{c} op_{\mathcal{M}}(pc) = \texttt{ br } c \texttt{ labelt } l_1 \texttt{ labelt } l_2 \\ \mathcal{G}(l_2) = (\textbf{label}, bb) \qquad \Sigma' = \Sigma :: v(c, \mathcal{L}) =_{\textbf{i1}} (\textbf{i1}, 0) \qquad \vdash \Sigma' \end{array}}{\langle pc, \mathcal{M}, \mathcal{G}, fs, \Sigma\rangle \longrightarrow \langle bb, \mathcal{M}, \mathcal{G}, fs, \Sigma'\rangle}\texttt{SBRF}$$

Table 2. Symbolic semantic rules (selected)

Discussion: Symbolizing memory. In this work we enforce the parameters to the alloca primitive to be concrete: both the return pointer (indicating where the allocated memory starts) and the size of the requested memory are concrete, that is not symbolic. This means that the only symbolic variables are the program inputs, and not the output of memory allocation. We are not interested on exploring different executions that arise from alloca returning different pointers in memory. One could extend our work in this direction, at the cost of making the analysis more complicated and difficult to reason with (but we do not believe we would be covering much more interesting code).

Operational correspondence. Given a symbolic execution trace tr, we let Σ_{tr} denote the sequence of recorded path conditions at the end of executing tr. We say that Σ_{tr} is σ-satisfiable when $c\sigma$ holds for each condition $c \in \Sigma_{tr}$.

Lemma 1 (Soundness of symbolic execution). *Let tr be a symbolic trace (i.e., one obtained with our symbolic semantics) with Σ_{tr} σ-satisfiable. Then $tr\sigma$ is a valid concrete trace.*

Proof. (Sketch) We rely on the property: if $\Sigma' = \Sigma \cdot \Sigma''$ and $\vdash \Sigma'$ then $\vdash \Sigma$.

By induction on the length of tr. Each last step of tr is a case analysis on a possible semantics rule; the interesting cases are the conditionals, where we see that a (later) solution to a constraint yielding an assignment to a symbolic input variable ι can be applied earlier.

This lemma formalizes the symbolic execution frameworks (e.g., KLEE). The converse to Lemma 1 also holds, but it is of course not as useful in practice.

3 Symbolic functions

Our aim is to verify code that uses cryptographic primitives like encryption or hashing. This is difficult with current state-of-the-art symbolic execution tools. To see this, consider the piece of code:

```
int main () {
   int i;
   char buf[80];
   for (i=0; i<80; i++)
     buf[i]=inp();
   assert (0x12345678 == hash(buf,80));
   printf("OK\n");
}
```

where `hash` is a cryptographic hash function, like SHA-1 or MD5. This code is unfeasible to be symbolically analyzed, as one is basically asking for an input that inverts 0x12345678, which is exactly what `hash` is designed to make difficult. In this section we develop a method to abstract away from such calls, which we regard as "symbolic functions" (analogously to the abstraction of inputs into symbolic variables).

3.1 Symbolic functions

The first thing to notice is that not every concrete function can be abstracted away into a symbolic one. For instance, a function that has arbitrary (memory) side-effects cannot be symbolically modelled as its behaviour is unknown and may change the execution state in an uncontrolled manner. Thus, we restrict ourselves to concrete functions that simply perform some (non-interactive) calculation; nevertheless this covers all the functions we are interested in (that is, cryptographic primitives). More precisely, the functions we want to abstract as symbolic comply with the following restrictions:

1. The types and number of input and output parameters are statically known, and are simple pointers or basic (constant) values;
2. The function only returns values in the (previously allocated) memory locations given by its output parameters (that is, there are no side-effects);
3. The function is not-interactive and does not do any inputs;
4. The function terminates in all of its possible inputs (that is, it never enters an infinite loop).

These conditions are needed to encapsulate all the behaviour of a function so that it can be easily abstracted away in a symbolic term. In order to model symbolic functions we thus need as given parameter information (conditions (1) and (2), and a precondition that prescribes when the function terminates succesfully and does not crash (condition (4)).

Example 1. Our target functions usually take as input a series of (p, l) where p is a pointer to some buffer and l is its length.

Thus, for each symbolic function we need to specify its input and output parameters, and their corresponding lengths; these lengths can be specified as static or also passed as parameter.

For example, consider functions zip/unzip: `zip(inbuf, inlen, !outbuf, !outlen)` and `unzip(inbuf, inlen, !outbuf, !outlen)`. Both functions take

two inputs (a pointer to a input buffer and its length), and returns two outputs (a pointer to an output buffer and its length). (To keep track of which parameters are inputs and which are outputs we mark the outpus with the bang !.)

3.2 Semantics

We are given a set of functions $f_1, \ldots, f_n$ which we are going to assume symbolic. As we discussed above, for each symbolic function f we assume given information about its parameters and their lengths. For a parameter p_i we let $len(f, p_i)$ to return its length (this can be either constant or another input parameter, which is then an identifier that needs to be looked up in the current state). We are also given a global precondition pre_f, modelled simply as a symbolic conditional expression, that specifies on which cases the function terminates.

We modify the symbolic semantics so that when a symbolic function is called a special SCALL rule is triggered (as opposed to concrete function calls which still use old rule CALL). Given a call to symbolic function f, the goal of rule SCALL is to avoid its execution, and simply create a symbolic term standing for each ouput parameter o_i of f. So, before presenting rule (SCALL), we extend the grammar of symbolic expressions as follows: $exp ::= \ldots | \iota^c_{f,o_i}$. For each symbolic function f with output parameter pi. A term $\iota^c_{f,pi}$ then represents the output pi of a call to f under context $c = \langle pc, \mathcal{M}, \mathcal{G}, fs, \Sigma \rangle$.

We are now ready to define the SCALL rule:

$$\frac{\begin{array}{c} op_{\mathcal{M}}(pc) = id = \texttt{call } f(i_0, \ldots, i_n, o_0, \ldots o_m) \quad f \text{ symbolic} \quad \vdash \Sigma' \\ \forall k : v(a_k, \mathcal{L}) = (t_k, v_k) \quad \mathcal{M}' = \mathcal{M}\{o_i \rightarrow \iota_{f,o_i,\mathcal{S}.j})\}_{j=0\ldots len(f,o_i)} \quad \Sigma' = \Sigma :: \{pre_f\} \\ \mathcal{S} = \langle pc, \mathcal{M}, \mathcal{G}, (res, \mathcal{L}, ret, \mathcal{A}) :: ffs, \Sigma \rangle \quad fs' = (res, \mathcal{L}\{id \rightarrow \iota_{f,o_i,\mathcal{S}}\}, ret, \mathcal{A}) :: ffs \end{array}}{\mathcal{S} \longrightarrow \langle nxt_{\mathcal{M}}(pc), \mathcal{M}', \mathcal{G}, fs', \Sigma' \rangle} \texttt{SCALL}$$

Briefly, this rule sets the symbolic outputs of f as symbolic expressions, and continues to the next instruction.

3.3 Specifying the behavior of a symbolic function

In formal methods for security (following Dolev-Yao), behavior is usually modelled via simple rewriting rules stating that e.g. decryption inverts encryption, or, for our Example 1, we would have $\forall x, unzip(zip(x)) = x$. However as we saw in our setting we deal with functions that output several parameters, of variable lengths. Outputs of symbolic functions are disseminated in bytes in memory, which can then passed as parameters to other symbolic functions; at this point, a rewriting rule may specify that a given output can be rewritten.

Thus, our rewriting rules need to specify how whole sets of memory locations (bytes) can be substituted by another set of bytes. Moreover, we wish to allow several rewriting rules, each modelling a different behaviour. For instance, one rewriting rule may specify when unzip inverts zip, while another rule may specify when unzip fails altogether (because, for instance, the length is wrong). So we also assume that each rewriting rule has an associated precondition to it.

Example 2. Following Example 1, we specify that unzip inverts zip when the lengths are correct. Assume that in memory location pointed by o starts the (symbolic expression representing the) output *outbuf* of a call to *unzip*. Formally, right after calling *unzip*, we have in memory:

$$\{o + j \to \iota^{c}_{unzip,outbuf} \cdot j\}_{j=0..outbuflen^{c}} \tag{1}$$

Here c is the context at the time of calling *unzip*, say $c = \langle pc, \mathcal{M}, \mathcal{G}, fs, \Sigma \rangle$, and $outbuflen^{c}$ denotes the actual value for parameter *outbuflen* in context c.

We have to look at the other input pointer parameter *inbuf* of unzip, say i_1; the rewriting rule has to enforce that the input comes from a call to *zip*:

$$\{i_1 + k \to \iota^{c'}_{zip,outbuf.k}\}_{k=0..outbuflen^{c'}} \tag{2}$$

Here c' contains the provided input to zip, namely *inbuf* of length *inbuflen*, pointed by i:

$$\{i + k \to inbuf_k\}_{k=0..inbuflen^{c'}} \tag{3}$$

At this point we are ready to rewrite 1) with

$$\{o + k \to inbuf_k\}_{k=0..inbuflen^{c'}} \tag{4}$$

provided that the precondition (for this rule)

$$pre_0 = inbuflen^{c'} <= outbuflen^{c} holds. \tag{5}$$

In summary, each rewriting rule consists of a series of matches over a context: (1), (2), (3), and the checking of a precondition (4); if this holds we can rewrite (1) with (4).

We now can add a rule that rewrites terms:

$$\frac{\mathcal{M}', \Sigma' \;\; \text{result of rewriting in } \mathcal{M}, \Sigma :: \sigma \;\; \text{with precondition } pre_0 \qquad \vdash \Sigma'}{\langle pc, \mathcal{M}, \mathcal{G}, fs, \Sigma \rangle \longrightarrow \langle pc, \mathcal{M}', \mathcal{G}, fs, \Sigma' \rangle} \texttt{REWR}$$

We need rewriting systems that are *decidable*. For the case of cryptography this is a well-studied problem; we can in fact restrict to rewriting systems which are *s*ubterm convergent, which is sufficient for decidability [14].

3.4 Operational correspondence under symbolic functions

Calling symbolic functions with rule (SCALL) introduces symbolic function expressions which are of course not possible to solve with a bitarray solver like STP. However, such expressions may disappear by rewriting (that is, applications of the (REWR) rule). In that case, we can then call the bitarray solver and continue the execution. We thus need to separate in Σ the constraints that depend on symbolic functions from those that do not.

Definition 3. *We say that Σ is fully σ-satisfiable when every constraint in Σ does not have occurrences of symbolic function expressions (because they were not initially there or were replaced by rewriting rules). We say that Σ is* partially *σ-satisfiable when Σ' is σ-satisfiable and Σ' is the result of removing each constraint with symbolic functions from Σ.*

In practice, after calling a (constructor) symbolic function, execution continues for a while (possibly hitting a branch conditional that does not include an occurrence of a symbolic function), and then finally reaches a (deconstructor) symbolic function that is then rewritten off. Thus, it may happen that a partially solvable Σ becomes fully solvable. Thus, we let rules for symbolic branching (that use $\vdash \Sigma$) to apply when Σ is either fully or partially σ satisfiable.

In order to state our main result, we need to provide for each concrete function implementation our symbolic one. We assume those concrete implementations meet their specification dictated by their rewriting rules.

Definition 4. *Let Σ be a constraint store. Let σ' be a substituion assigning concrete implementations $c_i()$ to each symbolic function $f_i()$ occurring in Σ. Then we say that Σ is functionally σ'-satisfiable if for each constraint Σ with occurrences of symbolic functions is satisfiable after applying σ'.*

Definition 5. *Let σ be a substituion assigning concrete implementations $c_i()$ to each symbolic function $f_i()$ occurring in symbolic trace tr. We write $[tr]\sigma$ to denote the resulting trace of replacing each call to symbolic function f_i using rule SCALL with the actual computation carried out by c_i, and removing any call to rule REWR. We write $[tr]\cdot$ to denote the resulting trace of removing each call to symbolic function f_i using rule SCALL, and removing any call to rule REWR.*

Theorem 1 (Soundness of Symbolic Execution under Symbolic Functions). *Let tr be a symbolic trace (i.e., one obtained with our symbolic semantics with a given set of symbolic functions).*

1. *If Σ_{tr} is fully σ-satisfiable, then $[tr\sigma].$ is a valid concrete trace.*
2. *If Σ_{tr} is partially σ-satisfiable and functionally σ'-satisfiable, then $[tr\sigma]\sigma'$ is a valid concrete trace.*

Proof. (Sketch)

1. Here there are no symbolic functions, hence the inputs are complete. We need to only apply Lemma 1.
2. Here we know of only the existence of σ' (but cannot compute it efficiently). When the SCALL rule is replaced by the whole concrete computation, we need to rely on the conditions that say that that whole computation does not change any state besides de output parameters.

The first item (Theorem 1(1)) says that if we reach an execution that has no occurrences of symbolic functions (e.g., we rewrote them off), then we can use the underlying bitarray solver to find an exact assignment for inputs (σ)

that reaches the same context concretely. The second item (Theorem 1(2)) says that even if we could not remove all symbolic functions, it *could* be possible to find a corresponding concrete execution (although this now depends on the path conditions that are still undecided).

For instance, in examples of encrypt-decrypt or zip-unzip we may reach the first case of the theorem; in a case of hash (see code at table **??**), we would reach the second case of the theorem: reaching the state that prints out "OK" depend on whether the assert holds, and that depends on a symbolic function which we are not going to solve directly.

4 Prototype and Examples

We developed a proof-of-concept prototype that works for symbolic functions that have a single output parameter (although they can take many inputs) of fixed length (this works for encryption/decryption primitives like Rijndael, or the hashing function SHA-1). As future work we plan to develop a general implementation that can deal with general symbolic functions as defined above.

KLEE's internal representation of symbolic expresions cannot handle our symbolic functions natively, since it only deals with expressions built from simple symbolic variables. We initially envisaged to augment KLEE's representation, but decided against so rather than modifying KLEE's internals we chose a less intrusive approach that works as an add-on wrapper for symbolic functions. KLEE already deals with environment and system calls by providing a hand-written library versions (for instance, a simpler version of `libc`). We follow a similar approach and code a wrapper that contains for each symbolic functions a code stub that works as follows:

1. The stub maintains a table that record the different calls made to it, along with the passed parameters and a unique *call identifier* related to that call;
2. Before accepting the passed parameters, the function preconditions are asserted, and the parameters are checked to see whether valid values and memory locations were provided;
3. If the same parameters were passed in an earlier call, so they exist already in the table, return the call identifier as response;
4. If new inputs are passed, try to deconstruct the call it using the rewriting rules; if possible, return the (deconstructed) parameters in the table;
5. If the call not be deconstructed, a new entry is generated in the table, with a fresh random value generated as call identifier and provided as output.

The advantage of this approach is that all this code is fed along with the original program that can be analysed directly by KLEE+STP, since at this point we have implemented the symbolic functions themselves as a library.

Note that on item (3) KLEE will generate symbolic comparisons when running the table lookups. So this enables KLEE to find conditions over the symbolic input that was put (or copied) in the stubs' internal tables.

On item (4) if it can decontruct the arguments a posibly symbolic value is taken from the tables and returned. If a case in wich undeconstructable arguments are pased it simply fill the output with random fresh data, also saving that in the internal tables. We have assumed that new symbolic function expresions will not be used in any comparison outside the interveining functions.

4.1 Experiments

Example of Section 3 with SHA-1. We have verified that if we choose SHA-1 as concrete implementation of the cryptographic `hash`, this prevents KLEE to find the two trivial paths in a reasonable time (we stopped after 2 hours in a system with an Intel Core 2 Duo T5500 with 4GB of RAM).

Example of Section 3 with CRC32 Of course, the notion of symbolic function also works for non-cryptographic code. If instead of choosing a hard-to invert SHA-1 as done above we take the checksum function CRC32 that is deployed everywhere, then KLEE+STP can compute it, that is it can solve CRC32 checksums (although this is clearly undesirable and expensive). Figure **??** in Appendix B shows the cost of solving inputs of up to 17 bytes in length (experiments run in the same system as before).

Hash as a symbolic function To overcome this we defined `hash` as a symbolic function and coded a stub to emulates its original behaviour. Thus, hash returns a fresh call identifier value named `ret`, and it saves the relation `buf[] -> ret` in its internal table. Using this stub, execution ends immediately. Since we do not have rewriting rules, `hash` is not invertible and then KLEE will find only one path. In our implementation this is modelled by the the call identifier being fresh; there are negligible chances that the stub `hash` returns exactly `0x12345678`; this spurious case is ignored.

Encrypt-decrypt We consider the C interface of Rijndael/AES, from the cryptographic API taken verbatim from RFC. Appendix A lists the relevant code. There, 16 bytes are read from input, then a rijndael context is set with a hard-coded key and its length. Then we encrypt and immediately decrypt the input data. In summary, we aim at being able to translate the symbolic conditions applied over the `decrypted` text to the original clear text `input`. We need to explore the two obvious paths, avoiding the pitfall of exploring the complexity of the crypto calls. The two trivial paths are walked using an `input` starting or not with the character 'A'.

We verified that if we mark the input as symbolic and we use the original `rijndael` implementation, KLEE fails to terminate and find the two simple path conditions (again, we tested a reasonable amount of time using the same system as in the previous experiments). This is not surprising of course since the cryptographic functions are designed on purpose to behave exactly like that.

5 Conclusions and future work

In standard symbolic execution, the inputs are assumed to be any bitarray; however, we can refine this assumption by stating that the inputs (which provide from an adversary) really depend on knowledge learnt by an attacker. Consider e.g. a program that outputs a secret key depending on some input. Then an attacker can use this key to decrypt some other message and get new potential inputs. We plan to model this by using standard knowledge theories.

We of course wish to extend to concurrent threads (protocol participants). In this new setting, consider an initiator and a responder: we have an adversary that chooses inputs to both parties, and can in addition touch en-route messages. In this setting it is interesting to model how different symbolic inputs flow from one thread to another (we do not know of any standard symbolic execution framework for dealing with plain, non-crypto concurrency).

KLEE tests for memory safety properties by automatically embedding tests in each state. In our case, we wish to encode higher-level properties, that depend on the specification of the code we are analysing. In the case of cryptographic protocols, we need to be able to encode correspondence assertions [19] that relate different parts of the (joint concurrent) memory state.

Shorter term goals to address include: encode libraries for common symbolic functions, not only cryptographic; tackle more (real and reasonably large) code examples; develop a (more thorough) prototype.

References

1. Mihhail Aizatulin, Andrew Gordon, Jan Jurgens, and Bashar Nuseibeh. Verifying implementations of security protocols in c (`http://users.mct.open.ac.uk/ma4962/files/abstract202010.pdf`).
2. Karthikeyan Bhargavan, Cédric Fournet, Ricardo Corin, and Eugen Zalinescu. Cryptographically verified implementations for tls. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM Conference on Computer and Communications Security*, pages 459–468. ACM, 2008.
3. Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *CSFW*, pages 82–96. IEEE Computer Society, 2001.
4. Bruno Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Trans. Dependable Sec. Comput.*, 5(4):193–207, 2008.
5. C Cadar, D Dunbar, and Engler. D.r.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *In: Proceedings of OSDI. (2008) 209224.*
6. R. Corin. *Analysis Models for Security Protocols.* PhD thesis, University of Twente, 2006.
7. Ricardo Corin. http://cs.famaf.unc.edu.ar/ rcorin/.
8. T. Dierks and E. Rescorla. The Transport Layer Security (tls) protocol. RFC 4346, Internet Engineering Task Force, April 2006. http://www.ietf.org/rfc/rfc4346.txt.
9. D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.
10. V Ganesh and Dill. D.: Stp: A decision procedure for bitvectors and arrays, http://theory.stanford.edu/ vganesh/stp.

11. Jean Goubault-Larrecq. Csur: Static analysis of C code. Available at http://www.lsv.ens-cachan.fr/csur/, 2002. Written in OCaml (12648 lines).
12. Jan Jürjens. Security analysis of crypto-based java programs using automated theorem provers. In *ASE*, pages 167–176. IEEE Computer Society, 2006.
13. James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
14. Steve Kremer, Stphanie Delaune, and Steve Kremer. March, 2009computing knowledge in security protocols under convergent equational theories .
15. Chris Lattner and Vikram Adve. The LLVM language reference manual. http://llvm.org/docs/LangRef.html.
16. D A Molnar and D Wagner. Catchconv: Symbolic execution and run-time type inference for integer conversion errors. Technical report, 2007.
17. Alfredo Pironti and Jan Jrjens. Formally-based black-box monitoring of security protocols. In Fabio Massacci, Dan S. Wallach, and Nicola Zannone, editors, *Engineering Secure Software and Systems, Second International Symposium, ESSoS 2010, Pisa, Italy, February 3-4, 2010. Proceedings*, volume 5965 of *Lecture Notes in Computer Science*, pages 79–95. Springer, 2010.
18. D Song, D Brumley, H Yin, J Caballero, I Jager, M G Kang, Z Liang, J Newsome, P Poosankam, and P Saxena. Bitblaze: A new approach to computer security via binary analysis. In *In Proceedings of the 4th International Conference on Information Systems Security*, 2008.
19. Thomas Y. C. Woo and Simon S. Lam. A semantic model for authentication protocols, 1993.

Appendix A: Rijndael Code

```
struct rijndael_ctx;
void rijndael_set_key(rijndael_ctx * ctx,
                      u_char *key, int key_len, int do_encrypt);
void rijndael_decrypt(rijndael_ctx * ctx,
                      u_char *clear, u_char *crypt);
void rijndael_encrypt(rijndael_ctx * ctx,
                      u_char *crypt, u_char *clear);
```

Several implicit preconditions over the memory buffers and parameters passed to these functions apply:

- `ctx` points to a `sizeof(*ctx)` sized memory buffer
- `key` points to a string of `key_len/8` bytes being the key
- `key_len` is either `64` or `128` or `256`.
- `clear` and `crypt` shall point to memory buffers of 16 bytes.

If we consider the following simple artificial code:

```
int main() {
  int i;
  rijndael_ctx ctx;
  char input[16], encrypted[16], decrypted[16];
```

```
  for (i=0; i<16; i++) input[i]=inp();
  rijndael_set_key (&ctx, KEY, KEY_LEN, 0);
  rijndael_encrypt (&ctx, input, encrypted);
  rijndael_decrypt (&ctx, encrypted, decrypted);
  if (decrypted[0] == 'A')
      return 1;
  return 0;
}
```

The implementation of the symbolic function stubs for `rijndael` follows the description in table 3. Using this implementation KLEE founds the two paths instantly.

rijndael_set_key: pushes a fresh symbolic value to `ctx` pointed memory, also it saves the relation `key[], key_len, do_encrypt -> ctx[]` in its internal table.

rijndael_encrypt: pushes a fresh symbolic value to `crypt` pointed memory, also it saves the relation `ctx[], clear[] -> crypt[]` in its internal table.

rijndael_decrypt: is a potential deconstructor. It seraches the tables for a `clear` text to return. All original values `(key, key_len, do_crypt, crypt)` shall match the apropiate rewriting rule. If there is no applicable rule it pushes a fresh symbolic value on `clear` pointed memory and saves the relation `ctx[], crypt[] -> clear[]` in its own table.

Table 3. Rijndael stubs

Appendix B: Hash with CRC32 figure

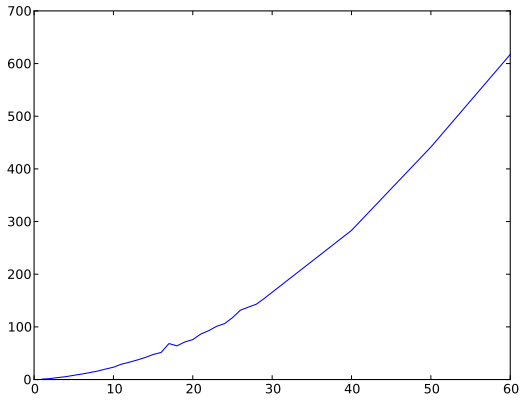

Fig. 1. KLEE+STP inverting CRC32 checksums: X is the number of bytes in the input, Y is the number of seconds of computation.