Análisis simbólico eficiente para implementaciones de protocolos criptográficos

Felipe Manzano Ricardo Corin

FaMAF - UNC

13/09/2010

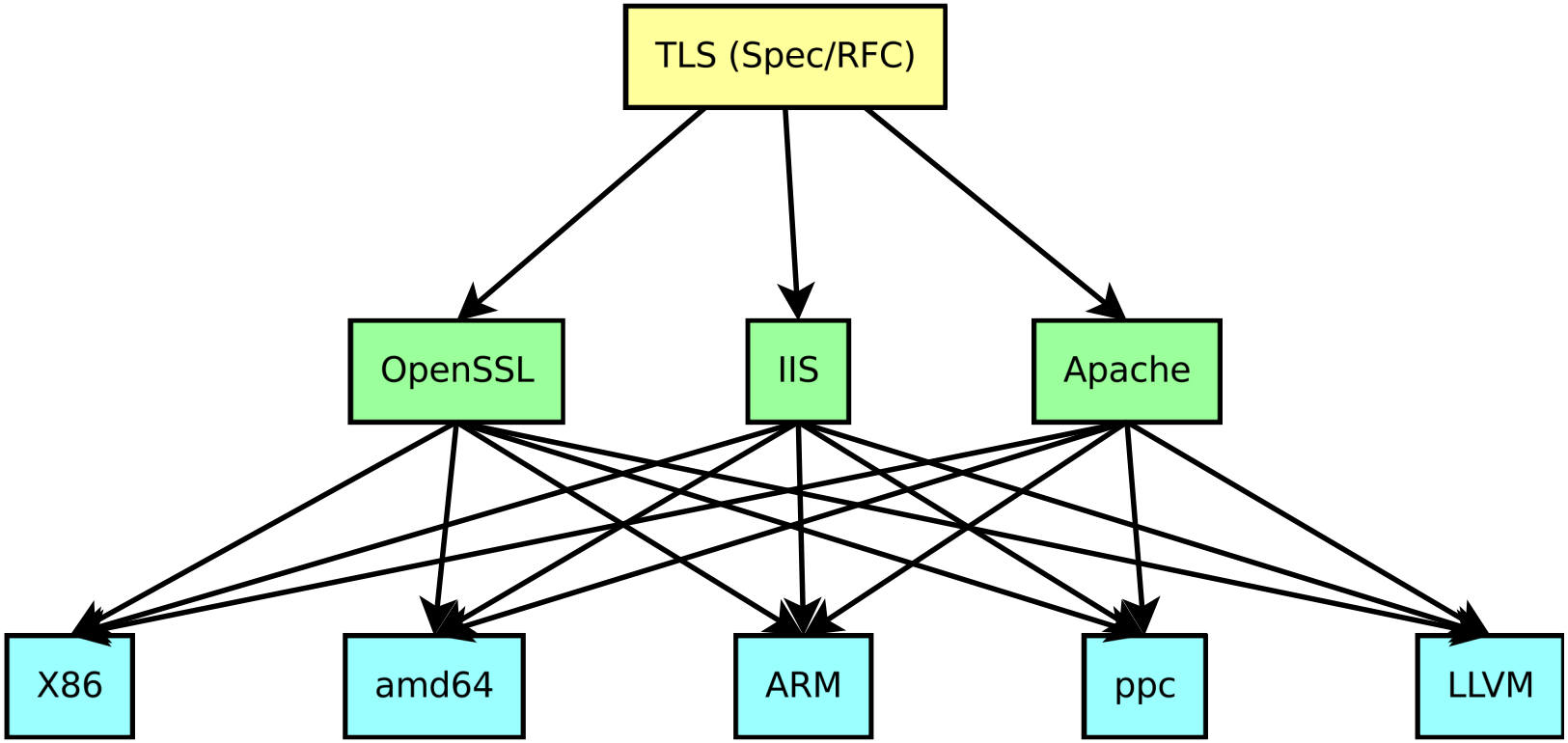
Problema

- En la actualidad sólo se analizan cripto protocolos a nivel abstracto
- Poco énfasis en sus implementaciones
- Estado del arte:
  - Formal:
    - Extraer un modelo de la implementación y verificarlo formalmente (C/ML -> proverif)
    - No funciona para implementaciones ya existentes (mainstream legacy code)
  - Ad-hoc

Implementaciones Vs. Modelos

- Siguen apareciendo errores en las distintas implementaciones.
- Muchos errores no aparecen en el análisis abstracto.
- Por cada especificación de un protocolo hay varias implementaciones (TLS -> openssl, apache, iis).
- Y varios compiladores optimizadores y arquitecturas.
- Necesitamos analizar código de bajo nivel.

Implementaciones Vs. Modelos (Dibujito)



WYSINWYX

What you see is not what you execute!

```
#include <stdio.h>
#include <sys/types.h>
int main (void) {
int32_t i, *p2=&i;
int16_t *p1=&((int16_t*) &i)[0];
for (*p1=0; *p1<10; (*p1)++){
      *p2=0;
}
printf ("%08x:%d\n", p1, *p1);
printf ("%08x:%d\n", p2, *p2);
printf ("%08x:%d\n", &i, i);
return (0);
}
```

Una vulnerabilidad de TLS

- Muy difícil de capturar con análisis abstracto
- Altamente dependiente de la implementación
- `www.openssl.org/news/secadv_20090107.txt`

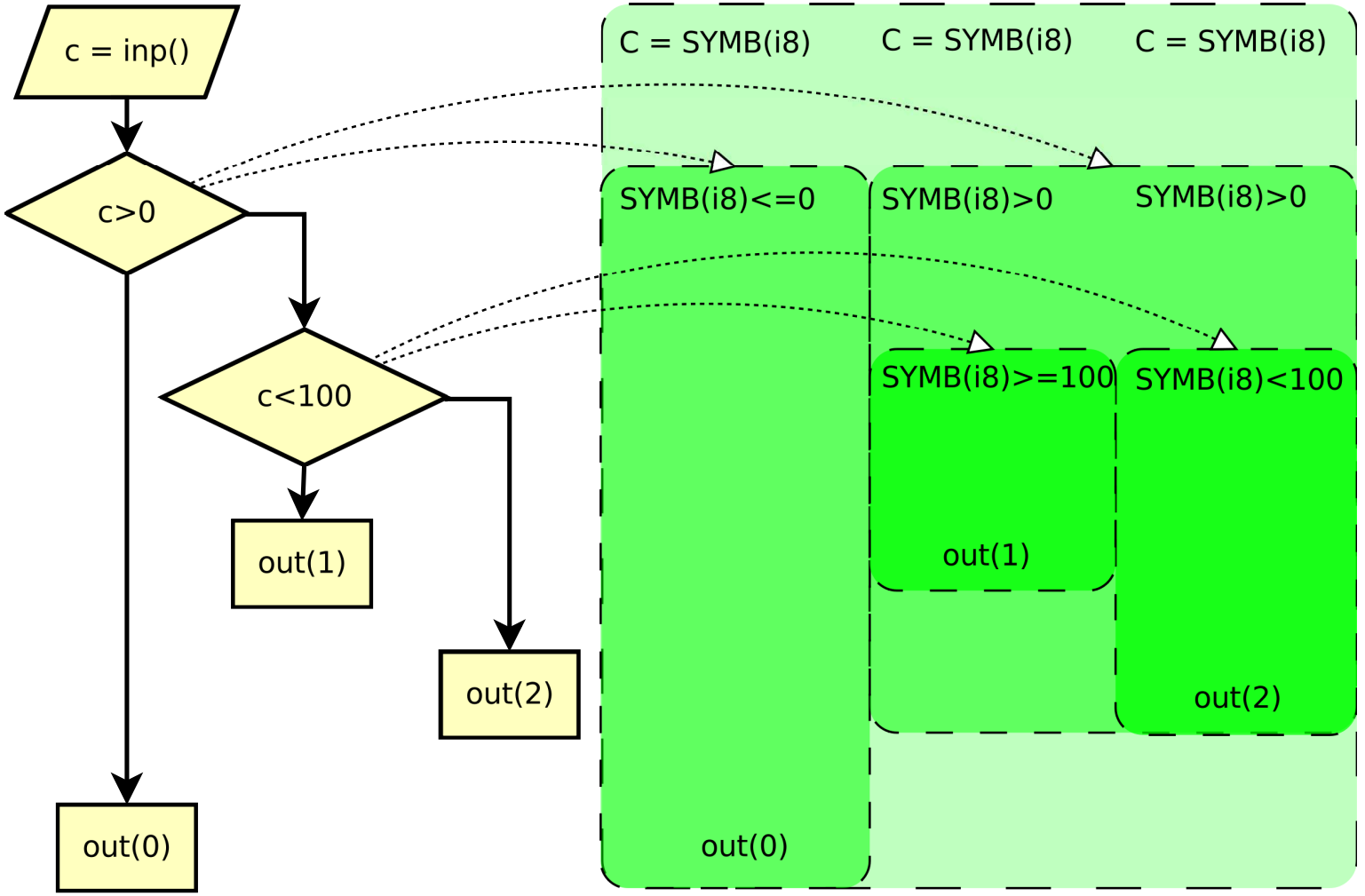
```
     ret=RSA_verify(NID_md5_sha1, buf,36, buf2,
     rsa_num, rsa_key[j]);
-    if (ret == 0)
+    if (ret <= 0)
     {
     BIO_printf(bio_err, "RSA verify failure\n");
```

Ejecución simbólica

Es un método para explorar código exhaustivamente.

- Existen varias herramientas (KLEE, bitblaze, catchconv, EXE, SOCA)
- Utilizan SAT solvers para decidir factibilidad de caminos y producción de contraejemplos
  - KLEE usa STP, un SMT solver para arreglos de bits
- Necesitamos un lenguaje intermedio/buen denominador común donde analizar!
- Elegimos LLVM(Low Level Virtual Machine)
  - existen varios compiladores (gcc/clang) que generan código llvm
  - tiene motor de ejecución simbólica (KLEE).

Ejecución simbólica - Dibujito



Un ejemplo básico

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

//i8 va de [-127 a 127]

- 2^8 entradas posibles
- 2 caminos posibles

Otro no tan básico

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

- $(2^8)^{80}$ entradas posibles
- 2? caminos posibles, instancias a resolver super complejas

Contribuciones

- Formalización de *LLVM′* (semántica concreta y simbólica)
- Una extensión de ejecución simbólica para cripto
- Un nuevo concepto: funciones simbólicas
  - Se evita la ejecución de funciónes complejas (cripto)
  - Se propagan valores simbólicos producto de las funciones simbólicas
  - Se aplican sustituciones/reescrituras cuando es posible:
    - dec(K, enc(K, msg)) -> msg
- Correspondencia operacional (concreta vs simbólica)

LLVM Flash!

- Un ensamblador tipado. (Permite optimizaciones y análisis)
- Documentación informal:
  `llvm.org/docs/LangRef.html`

```
ADD:    %result = add i32 %op1, %op2
CALL:   %result = call i8 @inp()
ICMP:   %cond = icmp slt i32 %op1, %op2
ALLOCA:%buf = alloca [80 x i8]
STORE: store i8 'A', i32* %buf
LOAD:   %c = load i8* %buf
BRANCH:br i1 %cond, label %t_bb, label %f_bb
```

El ejemplo Basico en LLVM

```
define i8 @abs() {
entry:
    %c = call i8 @inp()
    %cond = icmp slt i8 %c, 0
    br i1 %cond,
            label %true_b,
            label %false_b
true_b:
    %neg_c = sub i8 0, %c
    ret i8 %neg_c
false_b:
    ret i8 %c
}
```

Original en C:

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

Semántica concreta - Contextos

$\mathcal{M}$: la memoria

$\mathcal{G}$: variables globales

pc: contador de programa :)

fs: pila de marcos de activación $(rslt, \mathcal{L}, ret, \mathcal{A})$

- $rslt$ nombre donde poner resultado
- $\mathcal{L}$ variables locales
- ret dirección de retorno
- $\mathcal{A}$ direcciones de memoria temporales

Ejemplito de contexto:

$$\langle pc, \mathcal{M}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs \rangle$$

Semántica concreta - Regla ADD

$$\frac{\begin{array}{c} op_{\mathcal{M}}(pc) = id = \texttt{add} \quad t\ op_1, op_2 \\ v(op_1, \mathcal{L}) = (t, x_{op_1}) \qquad v(op_2, \mathcal{L}) = (t, x_{op_2}) \end{array}}{\begin{array}{c} \langle pc, \mathcal{M}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs \rangle \rightarrow \\ \langle next_{\mathcal{M}}(pc), \mathcal{M}, (rslt, \mathcal{L}\{id \rightarrow (t, x_{op_1} +_t x_{op_2})\}, ret, \mathcal{A}) :: fs \rangle \end{array}}\ \texttt{ADD}$$

Semántica concreta - Reglas INP/ALLOCA

$$\frac{op_{\mathcal{M}}(pc) = id = \texttt{inp}() \qquad b\ byte}{\langle pc, \mathcal{M}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs\rangle \rightarrow \langle next_{\mathcal{M}}(pc), \mathcal{M}, (rslt, \mathcal{L}\{id \rightarrow (\mathbf{i8}, b)\}, ret, \mathcal{A}) :: fs\rangle}\ \texttt{INP}$$

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

Semántica concreta - Reglas LOAD/STORE

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

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

Semántica concreta - Reglas BRANCH

$$\frac{op_{\mathcal{M}}(pc) = \texttt{br label } id \qquad \mathcal{L}(id) = (\textbf{label}, bb)}{\langle pc, \mathcal{M}, fs\rangle \rightarrow \langle bb, \mathcal{M}, fs\rangle}\texttt{BR0}$$

$$\frac{\begin{array}{c} op_{\mathcal{M}}(pc) = \texttt{br } c \texttt{ label } l_t \texttt{ label } l_f \\ \mathcal{L}(l_t) = (\textbf{label}, bbt) \qquad v(c, \mathcal{L}) = (\textbf{i1}, 0) \end{array}}{\langle pc, \mathcal{M}, fs\rangle \rightarrow \langle bbt, \mathcal{M}, fs\rangle}\texttt{BRT}$$

$$\frac{\begin{array}{c} op_{\mathcal{M}}(pc) = \texttt{br } c \texttt{ label } l_t \texttt{ label } l_f \\ v(c, \mathcal{L}) = (\textbf{i1}, 0) \qquad \mathcal{L}(l_f) = (\textbf{label}, bbf) \end{array}}{\langle pc, \mathcal{M}, fs\rangle \rightarrow \langle bbf, \mathcal{M}, fs\rangle}\texttt{BRF}$$

Semántica concreta - Reglas CALL/RET

$$\frac{op_{\mathcal{M}}(pc) = id = \texttt{call}\ f(a_0, a_1, \ldots a_n) \quad \mathcal{G}(f) = (t(t_0 id_0, t_1 id_1 \ldots t_n id_n), addr) \quad \forall k : v(a_k, \mathcal{L}) = (t_k, v_k)}{\langle pc, \mathcal{M}, fs\rangle \rightarrow \langle addr, \mathcal{M}, ((t, id), \{id_0 \rightarrow v_0, \ldots id_n \rightarrow v_n\}, next_{\mathcal{M}}(pc), \emptyset) :: fs\rangle}\ \texttt{CALL}$$

$$\frac{op_{\mathcal{M}}(pc) = \texttt{ret}\ t\ res \quad v(res, \mathcal{L}) = (t, x_p) \quad fs = ((t, id), \mathcal{L}, ret, \mathcal{A}) :: ((t_1, id_1), \mathcal{L}_1, ret_1, \mathcal{A}_1) :: fs'}{\langle pc, \mathcal{M}, fs\rangle \rightarrow \langle ret, \mathcal{M} - \mathcal{A}, (id_1, \mathcal{L}_1\{id \rightarrow (t, x_p)\}, ret_1, \mathcal{A}_1) :: fs'\rangle}\ \texttt{RET}$$

Semántica simbólica - Restricciones

- Cada vez que se elije una bifurcación en un **if**/**br** necesitamos recordar las condiciones necesarias para llegar a ese punto exacto en el código
- Nos interesan sólo restricciones sobre valores derivados de la entrada (valores simbólicos)
- Definimos un pequeñísimo lenguaje de expresiones ...

$$
\begin{array}{lll}
op_t & ::= & +_t, -_t, /_t, *_t \\
\text{exp} & ::= & (t, \iota) \mid (t,\text{const}) \mid \text{exp}\ \ op_t\ \ \text{exp} \\
 & & \mid\ \iota,exp\ \textbf{array(t)} \mid \text{exp}\textbf{[exp]} \\
cop_t & ::= & =_t, <_t, >_t, \&_t, |_t \\
\text{cond} & ::= & \text{exp}\ cop_t\ \text{exp}
\end{array}
$$

- Agregamos al contexto Σ para registrar las condiciones de camino

Semántica simbólica - Reglas SINP/SADD

$$\frac{op_{\mathcal{M}}(pc) = id = \texttt{inp}() \qquad \iota \text{ fresh in } \mathcal{L}}{\begin{array}{c}\langle pc, \mathcal{M}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs, \Sigma\rangle \rightarrow \\ \langle next_{\mathcal{M}}(pc), \mathcal{M}, (rslt, \mathcal{L}\{id \rightarrow (\mathbf{i8}, \iota)\}, ret, \mathcal{A}) :: fs, \Sigma\rangle\end{array}}\ \texttt{SINP}$$

$$\frac{\begin{array}{c}op_{\mathcal{M}}(pc) = id = \texttt{add} \;\; t \; op_1, op_2 \\ v(op_1, \mathcal{L}) = (t, x_{op_1}) \qquad v(op_2, \mathcal{L}) = (t, x_{op_2})\end{array}}{\begin{array}{c}\langle pc, \mathcal{M}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs, \Sigma\rangle \rightarrow \\ \langle next_{\mathcal{M}}(pc), \mathcal{M}, (rslt, \mathcal{L}\{id \rightarrow (t, x_{op_1} +_t x_{op_2})\}, ret, \mathcal{A}) :: fs, \Sigma\rangle\end{array}}\ \texttt{SADD}$$

Semántica simbólica - Reglas SBRANCH

$$\frac{op_{\mathcal{M}}(pc) = \texttt{br}\ c\ \texttt{label}\ l_t\ \texttt{label}\ l_f \quad \mathcal{L}(l_t) = (\textbf{label}, bbt) \quad \Sigma' = \Sigma :: v(c, \mathcal{L}) =_{\textbf{i1}} (\textbf{i1}, 1) \quad \vdash \Sigma'}{\langle pc, \mathcal{M}, fs, \Sigma\rangle \rightarrow \langle bbt, \mathcal{M}, fs, \Sigma'\rangle}\ \texttt{SBRT}$$

$$\frac{op_{\mathcal{M}}(pc) = \texttt{br}\ c\ \texttt{label}\ l_t\ \texttt{label}\ l_f \quad \mathcal{L}(l_f) = (\textbf{label}, bbf) \quad \Sigma' = \Sigma :: v(c, \mathcal{L}) =_{\textbf{i1}} (\textbf{i1}, 0) \quad \vdash \Sigma'}{\langle pc, \mathcal{M}, fs, \Sigma\rangle \rightarrow \langle bbf, \mathcal{M}, fs, \Sigma'\rangle}\ \texttt{SBRF}$$

Correspondencia operacional

Soundness of symbolic execution (lemma)

Toda ejecución simbólica se puede instanciar en una ejecución concreta que sigue el mismo camino.

Si las entradas simbólicas llevan a un error o estado particular, podemos producir entradas concretas que llevan al mismo estado.

Funciones simbólicas

- Algunas funciones son marcadas como simbólicas
- No se recorre la ejecución de ellas
- Sólo podemos hacer simbólicas a funciones que:
  - se asumen correctas; se comportan exactamente como está especificado
  - no llaman a inp(); realizan operaciones puramente de cálculo
  - no tienen efectos secundarios; devuelven los resultados en memoria pasada por referencia
  - no consumen memoria; liberan toda memoria dinámica que utilizan (trivial con alloca/ret)

Funciones simbólicas - Especificación

Necesitamos incorporar información sobre

- la signatura de las funciones simbólicas
- los parámetros de entrada o salida (que deben ser tipos básicos)
- el conjunto de reglas de reescritura al que responden
  - `dec(K,enc(K, x)) = x`
  - `unzip(zip(x)) = x`
  - `hash(x)`

(Para implementaciones reales, la regla de reescritura es mucho más compleja)

Funciones simbólicas - SCALL

- Cuando ocurre una llamada a una función simbólica se aplica una nueva regla `SCALL`
- Los parámetros de salida son asignados con una expresión símbolica especial
- Los contextos son equivalentes modulo reescritura sobre las expresiones

Funciones simbólicas - Correspondencia operacional

Correspondencia operacional con funciones simbólicas (Teorema)

Cuando todas las ocurrencias de funciones simbólicas pueden eliminarse aplicando las reglas de reescritura, la semántica con funciones simbólicas se corresponde operacionalmente con la semántica simbólica (y vía Lemma anterior con la semántica concreta)

Funciones simbólicas - Un ejemplo de encrypt

```
rijndael_encrypt (rijndael_ctx *ctx,
                  char input[16],
                  char encrypted[16]);
```

`ctx` es un contexto previamente inicializado con una clave, su tamaño es fijo (`sizeof(*ctx)`)

`input` apunta a los 16 bytes que serán encriptados

`encrypted` apunta a memoria reservada donde rij pondrá los datos encriptados

Al ejecutarse simbólicamente `encrypted[16]` se le asigna el símbolo:

```
rijndael_encrypt(ctx[488],input[16])
```

Funciones simbólicas - Un ejemplo de decrypt

```
rijndael_decrypt (rijndael_ctx *ctx,
                  char encrypted[16],
                  char decrypted[16]);
```

`ctx` es un contexto previamente inicializado con una clave, su tamaño es fijo (`sizeof(*ctx)`)

`encrypted` apunta a los 16 bytes que serán des-encriptados

`decrypted` apunta a memoria reservada donde rij pondrá los datos des-encriptados

Al ejecutarse simbólicamente una regla de reescritura podría entrar en funcionamiento.`decrypted[16]` es asignado con el símbolo:

```
rijndael_decrypt(ctx[488],encrypted[16])
```

Funciones simbólicas - Prototipo

- Nuestro prototipo es un parche no intrusivo sobre KLEE.
- Las funciones simbólicas son re-implementadas en base a tablas.
- Las reglas de reescritura se codifican en LLVM usando tablas.
- Los símbolos producto de ejecutar una f simbólica son emulados con arreglos de bits normales inicializados aleatoriamente.

El main

```
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;
}
```

Explota, explota mi ejecución

- KLEE no rompe rijndael.
- Una exploracion simbólica con una implementación normal de rijndael no termina.
- En nuestras pruebas KLEE no termina en tiempo razonable para crc32 de cadenas de más de 15 bytes.
- Utilizando nuestra implementación de funciones simbólicas el ejemplo termina inmediatamente.

Implementación con tablitas

rijndael_set_key: asigna un valor simbólico a ctx y guarda en una tabla interna todos los datos involucrados.

rijndael_encrypt: asigna un valor simbólico a encrypted y guarda en una tabla interna todos los datos involucrados.

rijndael_decrypt: debe buscar en las tablas de llamadas simbólicas una forma de deconstruir los parámetros aplicando cierta regla de reescritura.

Finalmente `rijndael_decrypt` puede retornar el texto claro original en el caso en que `encrypted` sea product de haber encriptado el texto claro con el mismo `ctx`.

Demo

DEMO

Conclusiones y trabajo futuro

- Un primer paso necesario para aislar lógica de protocolo de llamadas externas (primitivas cripto).
- Proveer un marco formal que muestra rigurosamente la corrección de nuestro método
- Primer prototipo simple y directo ya da resultados inmediatos en KLEE.

Trabajo futuro:

- Casos de estudio mas realistas (no solo cripto, ej. zlib, crc32)
- Lenguaje para definir interfaces de funciones simbólicas / propiedades de seguridad.
- Modelar adversarios con información parcial.
- Agregar concurrencia (como se propaga el simbolismo entre procesos).

PREGUNTAS