

Verifying Implementations of Security Protocols in C

Mihhail Aizatulin¹, François Dupressoir¹

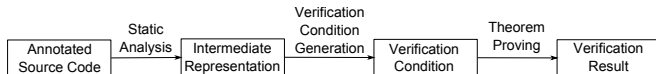
Supervisors: Andrew Gordon², Jan Jürjens^{1,2}, Bashar Nuseibeh¹

¹The Open University

²Microsoft Research Cambridge

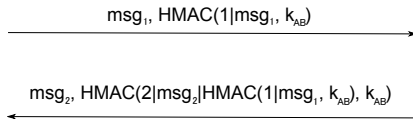
CRC PhD Student Conference
Open University
June 18-19, 2009

General purpose verifiers



- Used to prove different kinds of properties.
 - Memory safety: no memory access outside allocated locations.
 - Full functional correctness: the implementation is faithful to a given specification.
 - ...
- Annotations depend on the property.
 - Memory safety does not require many (if any) annotations.
 - For full functional correctness, annotations need to express the intended behaviour.
- Working on high or low-level languages makes a huge difference.

The many problems of low-level languages



The many problems of low-level languages

```

#include <ccv2.h>
#include <stdlib.h>
#ifdef VERIFY
#include <stdio.h>
#endif // VERIFY
#include "predicates.h"
#include "protocol.h"

size_t ClientCode(SOCKET channel, char* request, size_t lenReq, char* response, size_t lenRes)
{
    char *rcvbuf, *requestToMAC, *requestToSend, *responseToMAC;
    char requestMAC[MD_LEN], responseMAC[MD_LEN];
    size_t rcvSize, responseLen;

    // Getting the key information (with hardcoded IDs, for now)
    struct keys keystruct = getKey(1, 2);
    char *key = keystruct.key;
    size_t key_len = keystruct.length;

    // Putting the message into the output buffer
    requestToSend = malloc(lenReq + MD_LEN);
    memcpy_s(requestToSend, lenReq + MD_LEN, request, lenReq);

    // Computing the tagged MAC
    requestToMAC = malloc(lenReq + 1);
    requestToMAC[0] = 1; // Message number
    memcpy_s((requestToMAC + 1), lenReq, request, lenReq); // Message
    HMACSHA1(requestToMAC, (lenReq + 1), key, key_len, requestToMAC);

    // Concatenating the MAC to the message
    memcpy_s((requestToSend + lenReq), MD_LEN, requestToMAC, MD_LEN);

    // Sending the message
    sendNL(channel, requestToSend, lenReq + MD_LEN);

    // Freeing what can be freed.
#ifdef VERIFY
    free(requestToMAC);
    free(requestToSend);
#endif

    // Waiting for the responder to respond
    rcvbuf = malloc(lenRes + MD_LEN);
    rcvSize = receive(channel, rcvbuf, lenRes + MD_LEN);
    if(!rcvSize) return 0;
    responseLen = rcvSize - MD_LEN;

    // Copying out the MAC
    memcpy_s(responseMAC, MD_LEN, (rcvbuf + responseLen), MD_LEN);

    // Verifying the MAC
    responseToMAC = malloc(rcvSize + 1);
    responseToMAC[0] = 2; // Message number
    memcpy_s((responseToMAC + 1), rcvSize, rcvbuf, responseLen);
    memcpy_s((responseToMAC + responseLen + 1), MD_LEN, requestToMAC, MD_LEN);
    if(!HMACSHA1Verify(responseToMAC, rcvSize + 1, key, key_len, responseToMAC))
    {
        printf("The response could not be verified.\n");
        return 0;
    }

    // And returning properly
    memcpy_s(response, lenRes, rcvbuf, responseLen);
#ifdef VERIFY
    free(rcvbuf);
    free(responseToMAC);
#endif

    return responseLen;
}

```

The many problems of low-level languages

```
// Putting the message into the output buffer
requestToSend = malloc(lenReq + MD_LEN);
memcpy_s(requestToSend, lenReq + MD_LEN, request, lenReq);

// Computing the tagged MAC
requestToMAC = malloc(lenReq + 1);
requestToMAC[0] = 1; // Message number
memcpy_s((requestToMAC + 1), lenReq, request, lenReq); // Message
HMACSha1(requestToMAC, (lenReq + 1), key, key_len, requestMAC);

// Concatenating the MAC to the message
memcpy_s((requestToSend + lenReq), MD_LEN, requestMAC, MD_LEN);

// Sending the message
sendHL(channel, requestToSend, lenReq + MD_LEN);
```

Verifying Compliance

- This may seem more difficult than simply proving security properties.
- But we don't need to worry about the attacker.
- The idea: use the verifier to compare a simple symbolic version of the protocol to the target implementation.
- The symbolic version can be extracted from the specification.

Verifying Compliance (2)

How we did it and what we got from it

- Implement the protocol as a symbolic abstract state machine.
- All sent messages are compared to what they should be: "static monitoring".
- It works.
- But is it practical (automation, scalability)?
- We need to be able to do this without having to understand the program.

Correspondences and type-checking: F7

- Correspondence properties:
 - Correspondence properties describe an order relation on events.
 - They can be shown to correspond to security properties.
- Refinement types:
 - Extend standard type systems with logical formulas over the program variables (e.g. $[] : List\{length[] == 0\}$,
 $rev : \forall n \in \mathbb{N}, List_n \rightarrow List_n$)
 - Type-checking goes on normally, and a theorem prover is called when a formula needs to be proved.
- Using predicates and refinement types, it is possible to prove correspondence properties in $F\#$ [Bengtson *et al.*, 2008].

Using C contracts to mimic F7 refinement types

- We can do it directly, or we can be elegant about it...
- ... by abstracting C arrays as functional byte arrays.
- This gives two separate sets of predicates:
 - Axiomatically defined predicates for correspondences and high-level properties.
 - Concretely defined predicates to link high-level byte arrays to C arrays.

Example of byte array use

```

int WrapAndSend(SOCKET socket, char *message, unsigned short length)
//requires(mutable(message))
//requires(forall(size_t i; i < length; mutable(message + i)))
{
    spec(byte_array w, k, h);
    char *signature;

    struct keys keys = getKey(1, 2);
    size_t key_len = keys.length;
    char *key = keys.key;

    if (length >= DEFAULT_BUFLen - 5 - MD_LEN)
    {
        printf("This message is way too long...
            How did you get the courage to type all that anyway?\n");
        return 1;
    }

    signature = malloc(MD_LEN);
    assume(Disjoint(signature, MD_LEN, key, key_len));
    assume(Disjoint(signature, MD_LEN, message, length));

    Store(k, key, key_len);
    Store(w, message, length);
    assert(MACsays(k, w));

    HMACShal(message, length, key, key_len, signature);

    Store(h, signature, MD_LEN);
    assert(IsMAC(h, k, w));
    assert(Pub(h));

    if (sendHL(socket, message, length) <= 0) return 1;
    if (sendHL(socket, signature, MD_LEN) <= 0) return 1;

    return length;
}

```

The annotations are provable from simple function annotations and an axiomatic definition of the predicates.

The future: type-checking security in C?

- What is missing is an attacker model, that should be hard to add into the VCC/general verification process.
- Can we then delegate this attacker stuff to the theorem prover? At what cost?
- On the other hand, can we express an attacker in quantified C syntax?
- Can we adapt the type-checking approach directly onto C code without calling out on a general verifier?

Thank you

Any Questions?



Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis.

Refinement types for secure implementations.

In *CSF '08: Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, page 17–32, Washington, DC, USA, 2008. IEEE Computer Society.