IOT security design

SμV is a security middleware that uses selective software virtualisation and machine-level code verification to isolate a Trusted Software Module (TSM) from untrusted application software. The SμV approach rests on three pillars:
(i) selective software virtualisation of the microcontroller architecture,
(ii) deployment-time verification of incoming code at the assembly level and
(iii) toolchain modifications which allow developers to transparently compile their software for the virtual SμV architecture.

Furthermore, SμV is compatible with the vast majority of IoT microcontrollers and, crucially, as SμV does not require additional hardware security features, the approach may be applied to improve the security of millions of IoT devices that are already in the field. SμV is verified using VeriFast to be memory-safe and crash-free. As such, considering the verified properties, a high level of reliability, that is comparable to hybrid / hardwarebased approaches, is achieved, with respect to remote-only attacks.

DESIGN OF SμV

Memory isolation is the primitive used in almost all virtualization-based security systems. It denies any unauthorized access to sensitive physical memory locations. In this section, we provide an overview of the software mechanisms that SμV uses to provide memory isolation and support for various security features such as remote attestation. In particular, the SμV isolates itself and all accompanying secrets, crypto, and security-related functions from being tampered with by any untrusted user application that is loaded in the same physical memory.

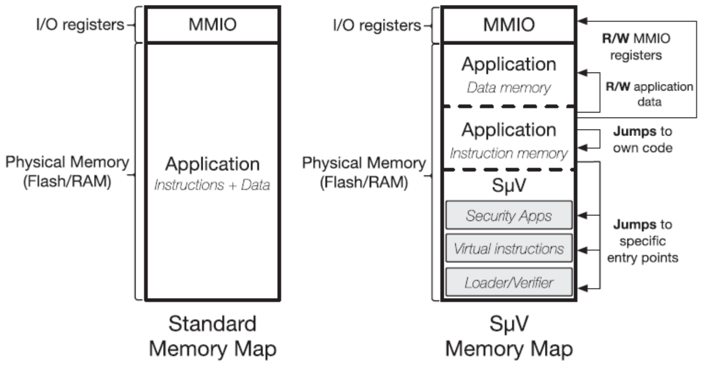

Fig. 1. Standard (left) and SμV (right) memory map. In the standard case, memory is monolithic and operations are unrestricted. SμV splits application memory into instruction and data memory and restricts possible sensitive operations.

The design of SμV shares similarities with the Softwarebased Fault Isolation (SFI) approach proposed by Wahbe et al. SFI prevents faults in untrusted software modules from corrupting other software on processors with a single shared address space and no memory protection. For each software module, SFI reserves a logically separate portion of the application's address space. The isolation of module address spaces is maintained at runtime by rewriting unsafe instructions to verify target addresses. SμV also uses selective software virtualization and assembly-level code verification to ensure full isolation of the trusted software module (SμV) from untrusted application software.

VeriFast

VeriFast is a sound and modular software verifier for C and Java programs. It is based on separation logic that extends Hoare logic. VeriFast checks and verifies certain safety properties of annotated C files. The annotations contain pre- and post- conditions expressed as separation logic assertions, ghost data structures (i.e., predicates), and ghost lemmas. The verification process in VeriFast is based on modular symbolic execution of the software module, where the body of each function of the program is executed symbolically, starting from the symbolic state identified by the function's preconditions, going through checking permissions of each memory location accessed by any statement inside the function, updating the symbolic state, and ending up with a symbolic state that should meet the function's postconditions. The safety property is verified if both pre- and post- conditions and any additional annotations are satisfied during the symbolic execution of each function. Thus, if VeriFast deems a program to be correct, then that program does not exhibit certain run-time errors that would violate the safety property.

We begin by annotating the code to indicate what it should do, and the verifier (in VeriFast) will then check whether the code complies with these annotations, providing proof steps that are automatically generated. Separation logic, the heart of VeriFast, is based on the concept of permissions. That is, each activation record (created by the symbolic execution of the annotated SμV code) holds a number of permissions, where accessing a specified memory location is granted if it holds the corresponding permission to do so. These permissions are identified through the various annotations. For any illegal access or operation, where there is no authorization to perform it, VeriFast reports an error.

Function Contracts. For each function in the SμV, a contract is identified, which is a formal specification of what the method guarantees. In particular, the contract consists of pre-conditions, specified by the keyword requires, and post-conditions, specified by the keyword ensures. The preconditions describe the permissions that the function requires in order to execute successfully. For example, in Listing 4, to initialize the state of the hash function, the function requires the permission to access struct sha1_ctx_t. This permission is granted by the pre-conditions identified, which let the function access the corresponding struct regardless the current values of its elements, where the symbol_ means that the function does not care about what values are stored in the corresponding variable. The postconditions describe the permissions that are transferred from the function to its caller when the function executes. According to Listing 4, the function should end up with a specified case, in which the elements of the *struct sha1_ctx_t* are filled in with custom values. Failing to do so will result in errors generated by VeriFast once the function is called. So, the contract says that the sha1_init function can access the memory locations of the corresponding struct and update the values of these locations. To this end, for each function call, the verifier in VeriFast verifies whether the implementation of that function satisfies its contract or not via symbolic execution. As it is shown in the example, all annotations in VeriFast are written inside special comments (= _ @:::@ _ = or ==@:::) which are ignored by the C compiler but recognized by the verifier (in VeriFast).

Predicates. To abstract over the set of permissions required by a function, permissions can be grouped and hidden via predicates. Predicates are named and parametrized assertions. In Listing 4, we use predicates in the contracts to hide the complexity of formalizing permissions needed. Listing 5 describes the body of the predicate state pred that groups and hides the permissions to access the various elements (e.g., each record of the h array, and the variable length) in *struct sha1_ctx_t*. Considering that the permission to access (i.e., read and write) the field f of an object *o* with value *v* is denoted as $o.f$ |- > *v*, the predicate indicates the permission to access any memory location of all elements attached as parameters along with the corresponding data structure, i.e., struct.

Predicates must be folded and unfolded explicitly using ghost statements, i.e., *open* and *close*, as shown in Listing 4. Verification of the code snippet in Listing 4 fails if any of the ghost statements is removed.

Loop Invariants. For each loop in any function of the SμV, we identify a loop invariant, a requirement by VeriFast, so that it can verify an arbitrary sequence of loop iterations by verifying the loop body once, starting from the initial symbolic state. To update the state in Listing 4, we have to iterate over the h array and update each record there. Thus, Loop invariants are required. This is clarified in Listing 6, in which, we use a predefined predicate by VeriFast, in the form of an array assignment, inside the loop invariant to control the size and values of the integer array. In the definition of any (pre-defined) predicate, some other pre-defined predicates might be used. Different types of lemmas are used in Listing 6 as we explain in the next section.

Lemmas and Auto-lemmas. (Auto)-Lemmas are ghost C-Like functions with the exception that they do not perform field assignment or call regular C functions. They are used for a customized need where other annotations cannot serve. The difference between auto-lemmas and lemmas is that the latter has to be called explicitly before the corresponding statement while the other is called by the verifier in VeriFast when needed. The major goal of lemmas is to transform some actual memory values into equivalent ones using other data types. For example, in Listing 7, the lemmas, i.e., *shiftRightNoUnderflow*, are used to verify that there is no arithmetic underflow once the arithmetic or bit-wise operations are performed in the *rotl32* function. The body of each lemma is a proof of its outcome.

```
...
typedef struct {
        uint32_t h[5];
        uint64_t length;
} sha1_ctx_t;

void sha1_init(sha1_ctx_t *state)
        //@ requires state_pred(state,?p,_,_,_,_,_,_);
        //@ ensures state_pred(state,p,0x67452301,
            ↪ 0xefcdab89,0x98badcfe, 0x10325476,
            ↪ 0xc3d2e1f0,0);
{
        //@ open state_pred(state,p,_,_,_,_,_,_);
        state->h[0] = 0x67452301;
        state->h[1] = 0xefcdab89;
        state->h[2] = 0x98badcfe;
        state->h[3] = 0x10325476;
        state->h[4] = 0xc3d2e1f0;
        state->length = 0;
        //@ close state_pred(state,p,0x67452301,
            ↪ 0xefcdab89,0x98badcfe, 0x10325476,
            ↪ 0xc3d2e1f0,0);
}
...
```

Listing 4. Simple example of using contracts in SμV

```
predicate state_pred(sha1_ctx_t *state, uint32_t *p,
    ↪ uint32_t h0, uint32_t h1, uint32_t h2, uint32_t
    ↪ h3, uint32_t h4,  uint64_t theLength) =
p == state->h &*& *p |-> h0 &*& *(p+1) |-> h1 &*& *(p+2)
    ↪ |-> h2 &*& *(p+3) |-> h3 &*& *(p+4) |-> h4 &*&
    ↪ state->length |-> theLength;
```

Listing 5. An example of a predicate in SμV

```
/*@
        lemma void getStatePredFromElements(sha1_ctx_t
            ↪ *state)
                requires chars(?x, 20, _) &*&
                    ↪ sha1_ctx_t_length(state,
                    ↪ ?theLength) &*& x == (char *)
                    ↪ (state->h);
                ensures state_pred(state,?p,?h0,
                    ↪ ?h1,?h2,?h3,?h4,theLength);
        {
                chars_split(x, 16);
                chars_split(x, 12);
                chars_split(x, 8);
                chars_split(x, 4);
                chars_to_integer_(x, 4, false);
                chars_to_integer_(x + 4, 4, false);
                chars_to_integer_(x + 8, 4, false);
                chars_to_integer_(x + 12, 4, false);
                chars_to_integer_(x + 16, 4, false);
                close state_pred(state,_,_,_, _,_,_,_);
        }
        lemma void assume_ADD_NoOverflow_long( uint64_t
            ↪ x, uint32_t y)
                requires 0 <= x &*& 0 <= y &*& x <=
                    ↪ ULLONG_MAX &*& y <= 65535 ;
                ensures x+y <= ULLONG_MAX;
        {
                assume(false);
        }
@*/
```

```
...
uint32_t a[5];
uint8_t t;
...
 //@ getStatePredFromElements(state);
//@ open state_pred(state,_,_,_,_,_,_,_);
/* update the state */
for(t=0; t<5; ++t)
        /*@ invariant t >= 0 &*& a[..5] |-> _ &*&
            ↪ state->h[..5] |-> _;  @*/
{
        uint32_t temp_h = state->h[t];
        //@ produce_limits(temp_h);
        uint32_t temp_a = a[t];
        //@ produce_limits(temp_a);
        state->h[t] = /*@ truncating @*/ (state->h[t] +
            ↪ a[t]);
}
//@ assume_ADD_NoOverflow_long(state->length, 512);
state->length += 512;
//@ close state_pred(state,_,_,_,_,_,_,_);
...
```

Listing 6. An example of a Loop invariant in SμV

```
/*@
        lemma void shiftRightNoUnderflow( uint32_t x,
             ↪ uint8_t y)
                requires y >= 0 &*& x >= 0 &*& y <= 255
                     ↪ &*& x <= UINT32_MAX;
                ensures x >> y >= 0;
        {
                shiftright_limits(x, N32, nat_of_int(y));
        }
        lemma void shiftRight32NoOverflow( uint32_t x,
             ↪ uint8_t y)
                requires y >= 0 &*& x >= 0 &*& y <= 255
                     ↪ &*& x <= UINT32_MAX;
                ensures x >> y <= UINT32_MAX;
        {
                shiftright_limits(x, N32, nat_of_int(y));
                pow_nat_nat_minus(2, N32, nat_of_int(y));
        }
        lemma void OR_NoOverflow( uint32_t x, uint32_t y)
                requires y >= 0 &*& x >= 0 &*& x <=
                     ↪ UINT32_MAX &*& y <= UINT32_MAX;
                ensures ((uint32_t)(x | y)) <= UINT32_MAX;
        {
                Z zx = Z_of_uint32(x);
                Z zy = Z_of_uint32(y);
                bitor_def(x, zx, y, zy);
                Z zt = Z_of_uint32(UINT32_MAX);
                Z_of_uint32_ltOReq(Z_or(zx,zy), zt);
        }
...
@*/
uint32_t rotl32(uint32_t n, uint8_t bits)
        //@ requires bits <= 32;
        //@ ensures 0 <= result &*& result <= UINT32_MAX;
{
        //@ produce_limits(n);
        //@ produce_limits(bits);
        uint32_t result = 0;
        uint32_t temp1 = /*@ truncating @*/ (n << bits);
        //@ shiftRightNoUnderflow(n , 32-bits);
        //@ shiftRight32NoOverflow(n , 32-bits);
        uint32_t temp2 = n >>(32-bits);
        //@ produce_limits(temp1);
        //@ OR_NoUnderflow(temp1,temp2);
        //@ OR_NoOverflow(temp1,temp2);
        result = (temp1 | temp2);
        return result;
}
```

Listing 7. An example of using Lemmas in SμV