T1: Secure Programming for Embedded Systems

NorthSec 2019

Developing for Embedded Systems

What is an “Embedded System”?



Embedded



Not embedded

What is an “Embedded System”?

Definition of “embedded system” is arbitrary.

What is meant here:

- Small 16-bit or 32-bit CPU (e.g. ARM Cortex M0+)
- RAM: 64 kB or less
- ROM: 256 kB or less
- Some network connectivity
- No operating system (“bare metal”)
- Strong constraints on size / power / thermal dissipation

(CPU is feeble but this does not matter much.)

Constraints: Consequences on Security

No memory management unit (MMU)

- All RAM is accessible read/write (and exec in some architectures)
- ROM (Flash) is all readable
- No sandbox / isolation
- No trapping of NULL pointer dereference
- No ASLR
- No guard page for stack overflows
  - Recursive algorithms must be banned

Constraints: Consequences on Security

No room for multiple or large stacks

- Multiple concurrent processes must run
- … but without locking the system
- A typical C stack needs at least 1-2 kB, more realistically 4 kB
- C tends to increase stack usage

Constraints: Consequences on Security

```
static
void battery_status_timeout_handler(void *p_context) {
    char msg[256];

    gfx_fillRect(0, 8, 128, 56, SSD1306_BLACK);
    gfx_setCursor(0, 12);
    gfx_setTextBackgroundColor(SSD1306_WHITE, SSD1306_BLACK);

    snprintf(msg, sizeof(msg),
        "Battery status:\n”
        "  Voltage: %04d mV\n”
        "  Charging: %s\n”
        "  USB plugged: %s\n",
        battery_get_voltage(),
        battery_is_charging() ? "Yes" : "No",
        battery_is_usb_plugged() ? "Yes" : "No");

    gfx_puts(msg);
    gfx_update();
}
```

Constraints: Consequences on Security

```
 0:   e92d 41f0      stmdb   sp!, {r4, r5, r6, r7, r8, lr}
 4:   b0c2           sub     sp, #264           ; 0x108
 6:   2400           movs    r4, #0
 8:   2338           movs    r3, #56 ; 0x38
 a:   2280           movs    r2, #128           ; 0x80
 c:   4620           mov     r0, r4
 e:   af02           add     r7, sp, #8
10:   9400           str     r4, [sp, #0]
12:   2108           movs    r1, #8
14:   f7ff fffe      bl      0 <gfx_fillRect>
18:   4620           mov     r0, r4
1a:   210c           movs    r1, #12
1c:   f7ff fffe      bl      0 <gfx_setCursor>
20:   4621           mov     r1, r4
22:   2001           movs    r0, #1
24:   f7ff fffe      bl      0 <gfx_setTextBackgroundColor>
(...)
```

24 bytes
264 bytes

Languages for embedded development

C

- Works everywhere
- “Portable assembly” but with a few hidden automatic costs
- Not *memory-safe*:
  - No check on array accesses
  - Manual allocation / deallocation → double-free, use-after-free, leaks…
  - Type punning
- “Undefined Behavior”
- Often required at some level (e.g. SDK offers only a C API)
  - It’s a C world

Languages for embedded development

Java ME

- GC, strong types,…
- Large RAM / ROM requirements
- Only ARM
- Needs an OS

Q: What are the system requirements for Oracle Java ME Embedded 8?

A: The high-level system requirements are as follows:

- System based on ARM architecture SOCs
- Memory footprint as low as 128 KB RAM and 1 MB ROM (see note)
- Very simple embedded kernel, or a more capable embedded OS/RTOS
- At least one type of network connection (wired or wireless)

Note: Footprint based on MEEP 8 Minimal Profile Set, optimized for single-function devices. Actual footprint will vary based on target device and use case.

Languages for embedded development

Go

- Only with TinyGo: https://tinygo.org/
- Limited language / runtime support:
  - "*support for goroutines and channels is weak*"
  - Maps can only have up to 8 (eight!) entries
  - GC: only for ARM, other platforms "*will just allocate memory without ever freeing it*" (but GC is required for proper string management)

Languages for embedded development

Rust Embedded: https://www.rust-lang.org/what/embedded

- Inherits all the memory-safety features of Rust
- Heap is optional
  - But without the heap, everything is allocated on the stack
- Supports ARM Cortex-M and Cortex-R, RISCV, and MSP430 (experimental)
  - But not AVR or Xtensa or other architectures that LLVM does not support
- Typically more stack-hungry than C
- Lots of automatic magic

Languages for embedded development

Forth

- Many incompatible implementations
  - It’s more a concept than a single defined language (though there is an ANSI standard)
  - You are supposed to “write your own Forth”
- Very compact, with low RAM usage
- Even less safe than C, and extremely non-portable

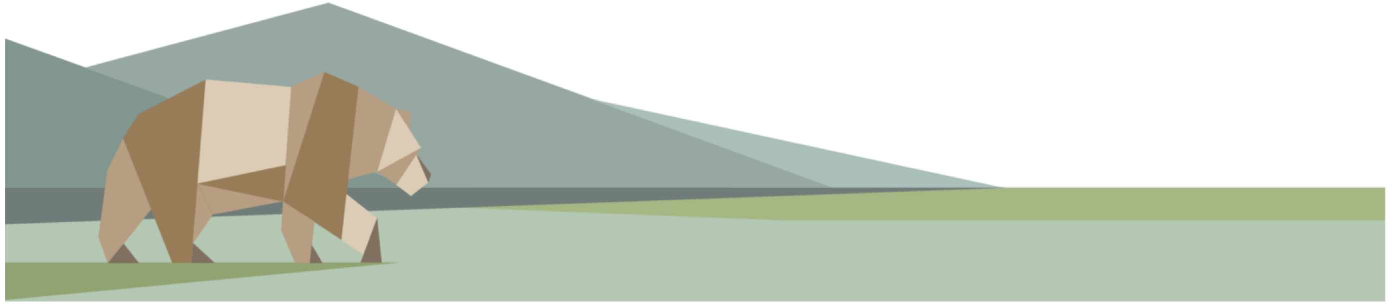
Languages for embedded development

Summary:

- No perfect language
- Adaptations from “larger languages” don’t solve the inherent issues, especially the cost of stacks for concurrent processing
- Often needs to interoperate with C
- Generic portability requires compiling *to* C
- Security is better addressed with a *non*-magic language

Success Story: BearSSL and T0

BearSSL



SSL/TLS library optimized for embedded systems

- Full-featured with uncompromising security (e.g. constant-time code)
- Portable, no dependency on any specific runtime, OS or compiler
- State-machine API
- No dynamic memory allocation whatsoever
- Can run in limited ROM and RAM (about 21 kB ROM and 25 kB RAM)
  - Can use less RAM, but requires support of small records by the peer

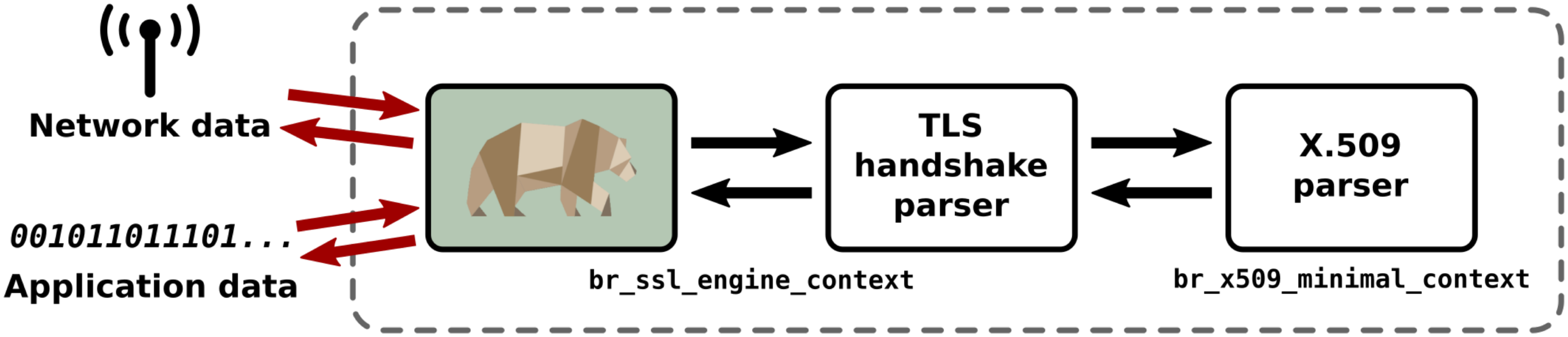
BearSSL

Problem: TLS handshake messages, and X.509 certificates, are complex, nested structures that can be large.

- X.509 certificate chain can be up to 16 MB
  - Realistically, 2 to 10 kB; sometimes larger (OpenSSL's default max is 100 kB)
- Data can be fragmented over different records
- Cannot buffer a complete message or certificate
  - Must perform streamed processing
  - Processing must be interruptible and restartable

Idea: run the decoding process as a *coroutine*

BearSSL



- BearSSL is computational only (application handles low-level I/O)
- Handshake parser and X.509 validation run as two coroutines
  - Each has its own state (stacks, variables)
  - Parsing proceeds when data becomes available, by chunks

T0

T0 is a Forth-like language used to implement the handshake parser and the X.509 validation engine.

- Compiled to *threaded code*
- Uses two custom stacks (data & system stack) of limited size (128 bytes each)
- Runs in a flat, small interpreter loop that can be stopped and restarted at will
- Instructions are a single byte each (*token threading*)
- Compiler is written in C# and performs some static analysis (maximum stack usage)

Threaded Code

```
\ Read one byte, enforcing current read limit.
: read8 ( lim -- lim x )
        dup ifnot ERR_X509_INNER_TRUNC fail then
        1- read8-nc ;

\ Read a 16-bit value, big-endian encoding.
: read16be ( lim -- lim x )
        read8 8 << swap read8 rot + ;

\ Read a 16-bit value, little-endian encoding.
: read16le ( lim -- lim x )
        read8 swap read8 8 << rot + ;
```

Executable code is (mostly) a sequence of function calls.

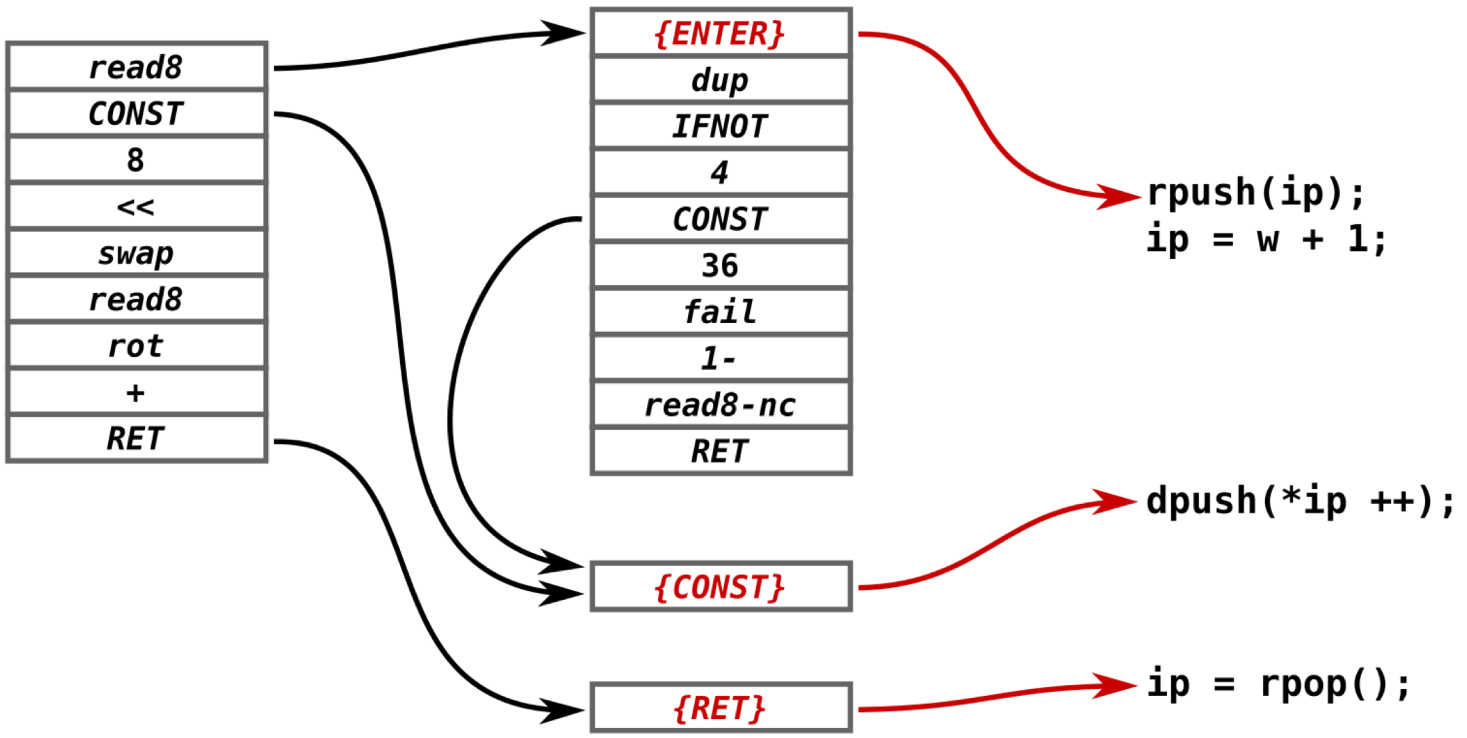
Indirect Threaded Code

```
: read8 ( lim -- lim x )
    dup ifnot 36 fail then 1- read8-nc ;
: read16be ( lim -- lim x )
    read8 8 << swap read8 rot + ;
```

```
for (;;) {
    w = *ip ++;
    (*w)();
}
```

Virtual CPU

```
CALL read8
CONST 8
CALL <<
CALL swap
CALL read8
CALL rot
CALL +
RET
```



nccgroup

Indirect Threaded Code

- Each function is a memory structure whose first field (CFA) is a pointer to native code.
- For *primitive* functions, there is only that pointer.
- *Interpreted* functions use the generic entry code (*{ENTER}*); CFA is followed by the function code as a sequence of pointers to function structures.
- Some primitive functions extract arguments located in the calling code (e.g. local jumps).
- Execution proceeds with a virtual CPU loop and two stacks:
  - *Data stack*: for function arguments and returned values
  - *Return stack*: for return addresses and local variables

→ **Stack usage is explicit**

Token Threaded Code

- Each pointer to a function structure is replaced with a token (index in a table of pointers).
- One extra indirection per instruction.
- Most/all instructions fit on one byte.
- Primitive function code can be integrated inside the virtual CPU loop.

T0 Compilation

```
$ ./T0Comp.exe -o src/x509/x509_minimal -r br_x509_minimal src/x509/asn1.t0 src/x509/x509_minimal.t0
[src/x509/asn1.t0]
[src/x509/x509_minimal.t0]
main: ds=17 rs=25
code length:   2836 byte(s)
data length:    299 byte(s)
total words: 203 (interpreted: 142)
```

- Compiler reads and interprets T0 code
  - *Immediate functions* are executed on-the-fly (metaprogramming)
- C source code is produced with tokens, primitives and virtual CPU
- X.509 validator compiled size (ARM Cortex M4):

```
$ size x509_minimal.o
   text    data     bss     dec     hex filename
   6259       0       0    6259    1873 x509_minimal.o
```

T0 Advantages

- Code can run as a coroutine with very small state (168 bytes for the two stacks)
- No dynamic memory allocation; streamed processing
- Guaranteed maximum stack usage
- Compiler verifies “types” (stack depth at all points)
- Small code footprint
- *No magic*

- … but not completely *memory-safe*

T1

T1

Evolution of T0 with extra features:

- Memory-safe
- Optional dynamic memory allocation (controlled) with GC
- Rich type system (including generics)
- OOP support
- Namespaces and modules

Memory Safety

Memory safety is a set of memory-related features:

- No uncontrolled type punning
- Array accesses outside of bounds are prevented
- No use-after-free or double-free
- Guaranteed stack usage (no overflow)
- Guaranteed maximum heap usage
- All allocated memory is released (no leak)
- Concurrent writing is controlled or prevented
- Etc…

Memory Safety in T1

Runtime checks:

- Array bounds on access
- Automatic memory management (garbage collector)

Compile-time checks:

- Maximum stack sizes
- Escape analysis (for stack-allocated objects)
- All method lookups are solvable
- No memory is interpreted with the wrong type
- No write access to static constant objects

OOP

```
class A {
    void foo(A a) {
        System.out.println("foo AA");
    }
    void foo(B b) {
        System.out.println("foo AB");
    }
}
class B extends A {
    void foo(A a) {
        System.out.println("foo BA");
    }
    void foo(B b) {
        System.out.println("foo BB");
    }
}
class C {
    public static void main(String[] args) {
        A x = new B();
        A y = new B();
        x.foo(y);
    }
}
```

Java code:

- Method call has a special first parameter (object *on which* the method is called)
- Method lookup uses the dynamic (runtime) type of the first parameter
- For other parameters, the static (compile-time) type is used

→ This program prints:

```
foo BA
```

OOP

```
struct A
end

struct B <sub> A
end

: foo (A A)
    "foo AA" println ;
: foo (A B)
    "foo AB" println ;
: foo (B A)
    "foo BA" println ;
: foo (B B)
    "foo BB" println ;

: main ()
    B new B new ->{ x y }
    x y foo ;
```

T1 code:

- No special parameter
- Method lookup uses the dynamic types of all parameters
- No explicit static type analysis

→ This program prints:

```
foo BB
```

Types

- Each value is a *pointer*
  - Plain integers, Booleans… are also “pointers”
  - No “value type”
- Every access to an object field is through an *accessor* (dedicated method)
  - Accessors locate the field unambiguously
- Basic types:
  - Booleans: `bool`
  - Plain integers: `int`
  - Modular integers: `u8 u16 u32 u64 i8 i16 i32 i64`

NULL

There is no null pointer value.

- Reading from an uninitialized object field triggers a runtime error
- Some object fields (basic types) are initialized at zero
- Possible reads from uninitialized local variables are detected at compilation

Plain Integers and Overflows

Strategies when integer operations overflow the representable range:

- Use modular arithmetic (C#, Java, Go)
- Report an error (Ada)
- Do one or the other, depending on external circumstances (Rust)
- Transparently upgrade to big integers (Python, Scheme)
- Use floating point (JavaScript)
- Anything goes (C, C++)

T1 uses the Ada way for "plain integers" (`int`) and modular arithmetic for exact-width integers (`u16`, `i32`...)

Whole Program Analysis

```
: triple (object)
    dup dup + + ;

: main ()
    4i32 triple println
    "foo" triple println ;
```

- Compute the complete call tree with possible stack contents.
- Each call of a given function is a different node.

```
main
 ├─4i32          [i32]
 ├─triple
 │   ├─dup       [i32 i32]
 │   ├─dup       [i32 i32 i32]
 │   ├─+         [i32 i32]
 │   └─+         [i32]
 ├─println       []
 ├─"foo"         [string]
 ├─triple
 │   ├─dup       [string string]
 │   ├─dup       [string string string]
 │   ├─+         [string string]
 │   └─+         [string]
 └─println       []
```

Whole Program Analysis

- Complete flow analysis from entry point:
  - For each function call, only cares about which types can actually be present on the stack.
  - Types for function definition are for call routing, not type restriction.
  - No syntax to express *potential* parameter types.
  - Return types are computed.
  - Dead opcodes and unreachable functions are detected.
- Multiple nodes for each function (one per call site):
  - All functions are *generic*.
  - Recursion would lead to an infinite tree (disallowed).
- Includes escape analysis and detection of writes to constant instances.

Current Status

Web site: https://t1lang.github.io/

Done:

- Specification + rationale
- Bootstrap interpreter/compiler:
  - Interpreter
  - Whole program analysis
  - Code generator (partial)

TODO:

- Finish bootstrap compiler
- Standard library (at least lists and sorted maps)
- Rewrite T1 compiler in T1