# TTC: Trust-Type Checking for C programs

**Benoît Boyer** & Adrien Champion[1]

2024/7/24

MITSUBISHI ELECTRIC R&D CENTRE EUROPE

MFR2024-ARC-0256

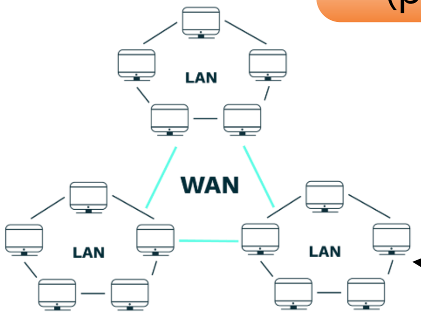
- Mitsubishi Electric is a 100 years old company (1921)
  - Long experience in code development in various domains
    ➔ Home appliance equipment … to … large and complex systems (plants)
  - Addressing safety-critical domains as well as cyber-security challenges
    ➔ Train, aerospace, satellite, plants, factory automation…
  - Large base of industrial C-code (embedded)

- Frama-C: a super toolbox for industrial needs
  - MERCE conducted experiments
    - Static analysis of legacy code (Frama-C/EVA, TrustInSoft Analyzer)
    - Automatic case test generation (PathCrawler)
    - Proving functional code analysis (Frama-C/WP)
    - …
  - MERCE also developed specific analyses (Frama-C plugins)
    - TTC is one of these projects

MITSUBISHI ELECTRIC
Changes for the Better

- SCADA systems: supervision of industrial systems

Public area
(potentially unsafe!)




Control SW must be secured

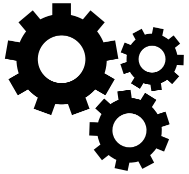
**integrity**

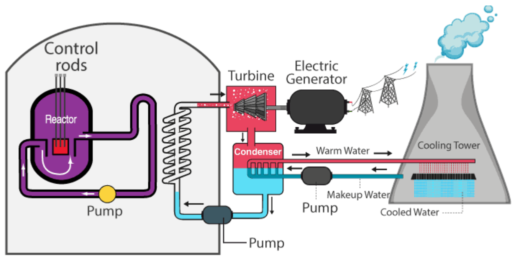
**security**

**confidence** **availability**

monitoring

actuating






- Critical area
- Restricted access

*"How to be sure that the isolation is implemented safely ?"*

Approach proposed

- Provide developers with security libs
- Automatically check that security functions are correctly used

# The considered workflow (roughly)

- Security experts in charge of
  - System analysis : weaknesses, threats…
  - Annotating API
    - Identification of the critical functions

      *e.g.*, actuation functions : `trusted` → `trusted / unsafe`
    - Explaining how unsafe data can be secured (security functions : `unsafe` → `trusted`)
- Developers
  - Implement the control SW (PLC programs)
  - Should respect the security policy (hopefully)
- TTC: automatic checking of the security policy
  - Rely on APIs annotated by security experts
  - Type errors ➔ security issues (`unsafe` data given while `trusted` content expected)
    - Should help developers to fix some security implementation issues

# Basic example : unsafe or trusted?

No annotation ⇒ unsafe
- read() is unsafe

Trusted data must be declared (with __attribute__)

```c
int read(void);

int __attribute__((trusted)) sanitize(
    int input
);

int __attribute__((trusted)) apply(
    int __attribute__((trusted)) input
);

void main_loop() {
    while (1) {
        int tmp = read();
        int safe = sanitize(tmp);
        int error = apply(tmp); // Using unsafe instead of trusted.
        if (error) {
            break;
        }
    }
}
```

Control flow (behavior) cannot rely on unsafe data!

Frama-C/TTC…

```
[kernel] Parsing example_0.c (with preprocessing)
[ttc] example_0.c:15: User Error:
  illegal call to function `apply`:
  parameter #1 (input) should be of type int trusted
  found expression of type int unsafe: tmp
[ttc] function `main_loop` is unsafe
[ttc] User Error: done with 1 errors
[kernel] Plug-in ttc aborted: invalid user input.
```

# Overview of the type system

- Organized in several layers “aligned" on C types (subset)

hierarchy

| Trust-types | | C - types |
|---|---|---|
| 1- Plain types | → | int, char, float… |
| 2- Compound types | → | struct {…}, union {…}, int arr[] |
| 3- References | → | int *i, *arr[], *struct {} |

- TTC analysis is sound for analyzed programs
  - Free of runtime errors
  - Single threaded

- Simple memory layout supported
  - No nested pointers ➔ OK for many PLC programs

# Plain Trust-Types

- Two main types for simple data types
  - Trusted
  - Unsafe

```
int a, b, c; //uninitialized vars are unsafe

a = 1;      //a is trusted, because constants are trusted
b = unsafe_get();
c = b * a;  //c is tainted unsafe because of b
while (c >= 0) {  // type error -> control flow based on unsafe data
    apply(b); // type error: apply requires Trusted data
    b = sanitize(b, a); // now, b is trusted
    c--;
}
apply(b); type error again....
```

$\texttt{cmp}(tt_1, tt_2)$ returns

- Some 0 — if $tt_1$ and $tt_2$ are the **same**,
- Some $n > 0$ — if $tt_1$ is **strictly more trusted** than $tt_2$,
- Some $n < 0$ — if $tt_2$ is **strictly more trusted** than $tt_1$, and
- None — otherwise.

- $\texttt{cmp} : \Pi ype \to \Pi ype \to \mathbb{N}\ \texttt{option}$
- $\texttt{join} : \Pi ype \to \Pi ype \to \Pi ype$

***More trusted ?***

# Plain Trust-Types

- Akin to tainting analysis…

unsafe
↑ ⊔
trusted

- Type checking implemented as abstract interpretation
  - The simplest lattice
  - Operations ⊔, ⊏
- Tainting ➔ ⊔ is sufficient
- Subtyping (⊏) : *"Any* ***`trusted`*** *data can be considered as* ***`unsafe`****"*

We introduced Functions

- $\mathtt{cmp} : \Pi\mathrm{ype} \to \Pi\mathrm{ype} \to \mathbb{N}$ `option`
- $\mathtt{join} : \Pi\mathrm{ype} \to \Pi\mathrm{ype} \to \Pi\mathrm{ype}$

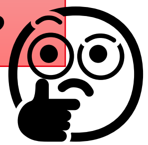
Comparison for subtyping

$\mathtt{cmp}(tt_1, tt_2)$ returns

| | | |
|---|---|---|
| - | Some 0 | if $tt_1$ and $tt_2$ are the **same**, |
| - | Some $n > 0$ | if $tt_1$ is **strictly more trusted** than $tt_2$, |
| - | Some $n < 0$ | if $tt_2$ is **strictly more trusted** than $tt_1$, and |
| - | None | otherwise |

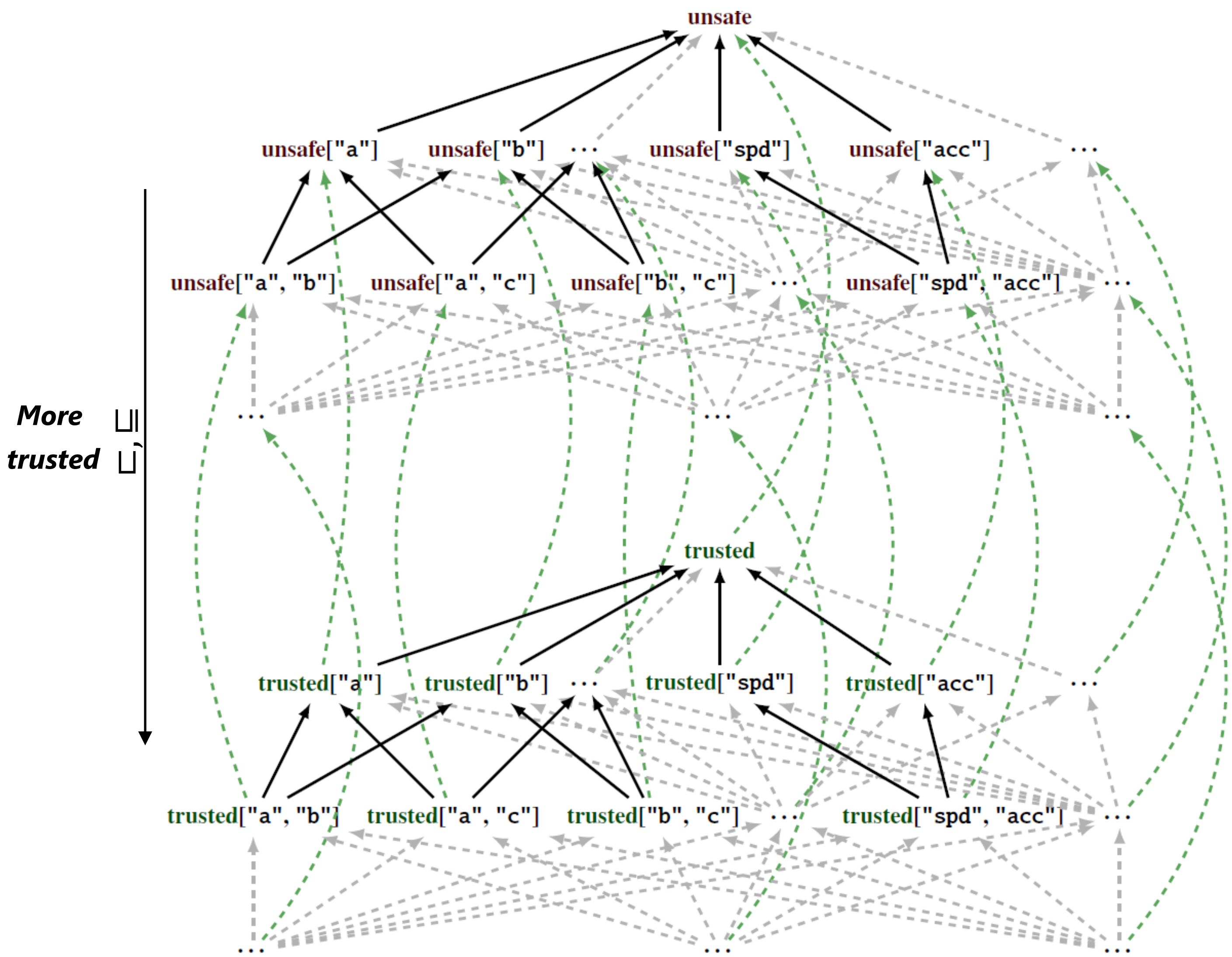
***More trusted ?***

***Partial order ???***



# More informative Trust-Types

- Quickly, { `trusted`, `unsafe` } became too limited ➔ Trust-types with **tags**




Examples:
`trusted[“key”], unsafe[“command”],`
`trusted[“user”],`
`trusted[“speed”, “accel”] …`

Practically, the lattice is finite, because

- considered tags = annotations (finite set)

No complexity issue

- new trust-types inferred from `join()`

| | | | | |
|---|---|---|---|---|
| join( | **unsafe**, | **trusted**["spd", "acc"] | ) = | **unsafe** |
| join( | **trusted**["a", "b"], | **trusted**["a", "c"] | ) = | **trusted**["a"] |
| join( | **trusted**["a", "c"], | **trusted**["spd", "acc"] | ) = | **trusted**[] |

# Composites trust-types

- For C-`struct`

**Definition** *A **composite** Πype for a composite type τ with fields* $field_1, \ldots, field_n$ *is a complete map from fields to plain Πypes. We will write composite Πypes as*

$$\{\, field_1 : tt_1,\ \ldots,\ field_n : tt_n \,\}.$$

- Comparing composite types

$$\mathtt{cmp}(\ \{field_1 : tt_1,\ \ldots,\ field_n : tt_n\},\ \{field_1 : tt'_1,\ \ldots,\ field_n : tt'_n\}\ )$$

*is Some res if*

$$\forall i \in [1, n],\ \mathtt{cmp}(tt_i,\ tt'_i) = \textit{Some res or Some } 0,$$

*None otherwise.*

- Extending `join`() to composite types

$$\mathtt{join}(\ \{field_1 : tt_1,\ \ldots,\ field_n : tt_n\},\ \{field_1 : tt'_1,\ \ldots,\ field_n : tt'_n\}\ )$$

*is*

$$\{field_1 : \mathtt{join}(tt_1, tt'_1),\ \ldots,\ field_n : \mathtt{join}(tt_n, tt'_n)\}.$$

(field wise join)

# Array trust-types

- Two cases
  - Known length array → $Array(tt_1, \ldots, tt_n)$
  - Unknown length (or too large !) → $Vec(tt)$, $tt$ *representing the trust-type of each cell*

- Comparison of arrays is cell-wise (if possible…)

$$
\begin{array}{llll}
cmp( & Array(tt_1,\ldots,tt_n), & Array(tt'_1,\ldots,tt'_n) & ) = Some\ m \;\middle|\; if\ \forall i \in [1,n],\ cmp(tt_i, tt'_i) = Some\ m\ or\ Some\ 0 \\
cmp( & Array(tt_1,\ldots,tt_n), & Array(tt'_1,\ldots,tt'_n) & ) = Some\ m \;\middle|\; otherwise \\
cmp( & Vec(tt), & Vec(tt') & ) = cmp(tt, tt') \\
cmp( & Array(tt_1,\ldots,tt_n), & Vec(tt) & ) = cmp(Vec(join(tt_1,\ldots,tt_n)), Vec(tt)) \\
cmp( & Vec(tt), & Array(tt_1,\ldots,tt_n) & ) = cmp(Vec(tt), Vec(join(tt_1,\ldots,tt_n)))
\end{array}
$$

- Join arrays

$$
\begin{array}{llll}
join( & Array(tt_1,\ldots,tt_n), & Array(tt'_1,\ldots,tt'_n) & ) = Array(join(tt_1,tt'_1),\ldots,join(tt_n,tt'_n)) \\
join( & Vec(tt), & Vec(tt') & ) = Vec(join(tt,\ tt')) \\
join( & Array(tt_1,\ldots,tt_n), & Vec(tt) & ) = join(Vec(join(tt_1,\ldots,tt_n)), Vec(tt)) \\
join( & Vec(tt), & Array(tt_1,\ldots,tt_n) & ) = join(Vec(tt), Vec(join(tt_1,\ldots,tt_n)))
\end{array}
$$

# Offset projections

- Access to fields

**Definition** *Given a field $f$ : string and a Πype $tt$ : Πype,*

$$\texttt{resolve\_field}(f, field_1 : tt_1, \ldots, f : tt, \ldots, field_n : tt_n) = tt.$$

- Access to arrays

**Definition** *Given an optional index $idx : \mathbb{N}$ option and a Πype $tt$ : Πype, function* resolve_index *outputs*

$$\begin{array}{llll|l}
\texttt{resolve_index}( & Some\ i, & Array(tt_1, \ldots, tt_n) & ) = tt_i & if\ i \leq n \\
\texttt{resolve_index}( & Some\ i, & Array(tt_1, \ldots, tt_n) & ) = \mathbf{unsafe} & if\ i > n \\
\texttt{resolve_index}( & None, & Array(tt_1, \ldots, tt_n) & ) = join(tt_1, \ldots, tt_n) & \\
\texttt{resolve_index}( & _, & Vec(tt) & ) = tt &
\end{array}$$

← Out of bounds error detected

We assume the program runs **safely** !
(no out of bound access, valid field accesses …)

# What about references?

- From the case studies (Factory Automation)
  - Simple memory layout (PLC applications)

    Static memory allocation, 1 level of referencing (pointer to structs to arrays), no nested pointers…

- Currently, TTC handles a very basic pointer manipulation
  - No need of complex aliasing analysis

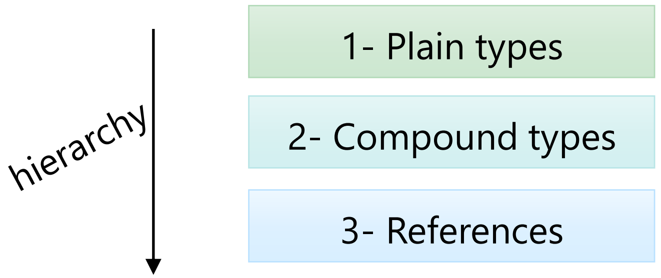
- We introduce **references** on top of plain/composite/array trust-types

- 3 kinds of references

  **unknown** : not initialized or no information about it

  **exact** : the reference target is well-known

  **corruption** : the reference may have several targets



# References - example

```
int read(void);
int __attribute__((trusted)) sanitize(
    int input
);
void apply(
    int * __attribute__((trusted)) input
);
int __attribute__((trusted)) get_cond(void);

void main_loop() {
    int v_1;
    int v_2;
    int *pntr;

    while (1) {
        if (get_cond()) {
            v_1 = read();
            pntr = &v_1;
        } else {
            v_2 = read();
            pntr = &v_2;
        };
        *pntr = sanitize(*pntr);

        apply(pntr);
    }
}
```

At the end of **`if-then-else`**, we deduce that

v_1 : **unsafe**
v_2 : **unsafe**
pntr : Corrupt( None, {v_1 ↦ {[]}, v_2 :↦ {[]}} )

Because we have no idea

whether **`pntr`** points to **`v1`** or **`v2`**

➔ We don't know which of **`v1`**, **`v2`** has been sanitized

BUT we guarantee that **`pntr`** is **trusted** (sanitized)

v_1 : **unsafe**
v_2 : **unsafe**
pntr : Corrupt( Some **trusted**, {v_1 ↦ {[]}, v_2 :↦ {[]}} )

TTC deduces that the call to **`apply()`** is safe !

While it would not be with **`&v1`** or **`&v2`**

# Fragile functions

Using a critical function with unsafe data

```
trusted apply (trusted input,
               trusted input2);
```

raises an error !

Sometimes we would like to use the same function with `unsafe/trusted` contexts

A `fragile` function becomes unsafe it is fed with `unsafe` content

```c
/// Reads an unsafe integer.
int read(void);

/// Sanitizes an untrusted integer.
int bad_sanitize(
    int input
);

int  __attribute__((trusted)) good_sanitize(
    int input
);
/// Applies something, input integer must be trusted.
///
/// Return value is an error flag (true if error) and is trusted.
int __attribute__((fragile,trusted)) apply(
    int __attribute__((trusted)) input,
    int __attribute__((trusted)) input2
);

/// Entry point.
void main_loop() {
    while (1) {
        int tmp1 = read();
        int tmp2 = read();
        int safe1 = bad_sanitize(tmp1);
        int safe2 = good_sanitize(tmp2);
        int error = apply(safe1,safe2);
        if (error) {
            break;
        }
    }
}
```

unsafe data provided !

**Fragility !**

TTC Error: control flow on unsafe data

# To sum up

- TTC – trust types checking
  - Akin of tainting analysis
  - Quick check for detect for security implementation issues
  - Embedded control SW

- Limitations
  - absence of runtime errors using abstract interpretation ?

    Buffer overflow is a major issue

- Perspectives
  - Function annotations = contracts ➔ verifying function implementations *vs.* contracts
  - Improve the alias analysis, handle more complex memory layout (addressing other domains than FA)

# Feedback from R&D Japan

- The tool have been evaluated by R&D in Japan… issues drawbacks
  - Implementation in Ocaml in industrial context… (no internal support for the language & tool)
  - Too limited support of windows platforms (common development platforms)
  - Additional effort and work for integrating the tool in existing workflows
    - Mitsubishi Electric provides an IDE for factory automation
    - Difficult for MERCE to anticipate all the needs, case-by-case study to adapt the technology…

- MERCE's objectives for formal methods
  - Identify the targets and technologies to be used
  - Demonstrate and highlight the benefits of formal methods for industry
  - Evaluate the scientific and technological issues, (jointly with Japanese R&D)
  - Promote and provide integration means to easy technology adoption

MITSUBISHI ELECTRIC

*Changes for the Better*