Information Flow Analysis and Type Systems for Secure C Language (VITC Project)

Jun FURUSE

The University of Tokyo
furuse@yl.is.s.u-tokyo.ac.jp

e-Society

MEXT project

toward secure and reliable software infrastructure
for highly networked information society

e-Society in Yonezawa lab.

Related 3 sub projects:

Safe language

- **Secure** existing programming languages and programs
  for **system description** (i.e. C/C++)

Safe OS by typing

- Construct **type secure OS kernel**
  using **TAL** (typed assembly language)

Safe OS by theorem prover

- Develop formal method
  to prove **correctness of safe memory management**
  using **Coq** theorem prover

Safe language sub-project

Goal

Securing **existing** C programs with **minimum** modifications by providing better compilers (VITC).

Current threat

Many security violation incidents and security hole alerts are reported around programs written in C language.

Final disaster: **security leaks**.

VITC in spotlight

Programs (written in C) **survive attacks**,
once compiled by VITC
(**V**ulnerability and **I**ntrusion **T**olerant **C**ompilation)

Memory safe

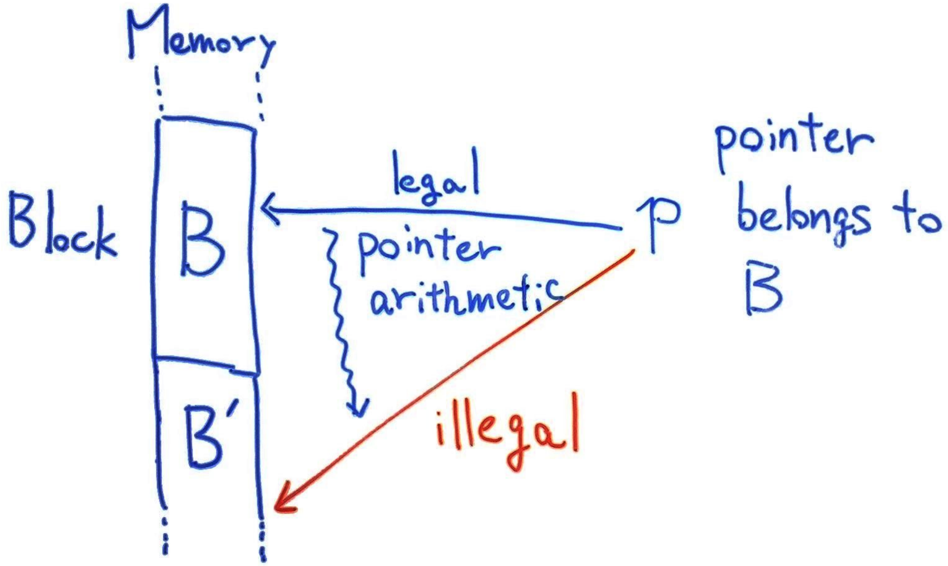
Memory accesses are checked to prevent **buffer overflow attacks**.

Information flow security

Programs **never leak** secret information.

Memory safety in C

Existing works:



StackGuard

By canary words

NX-bit

Approach from hardware

CCured, Fail-Safe C, etc.

Memory secure reimplementation of C compiler

- Range check for each memory access
- Optimization thanks to typing and pointer analysis

Safety by Failure

They are all **fail-safe**:

- StackGuard
- NX-bit
- CCured, Fail-Safe C

Detection of illegal memory access $\Longrightarrow$ Termination of program

Limitation of fail-safety

Fail safety is **secure**,
but **not sufficient** in some environment.

The same attack now **kill** the program:

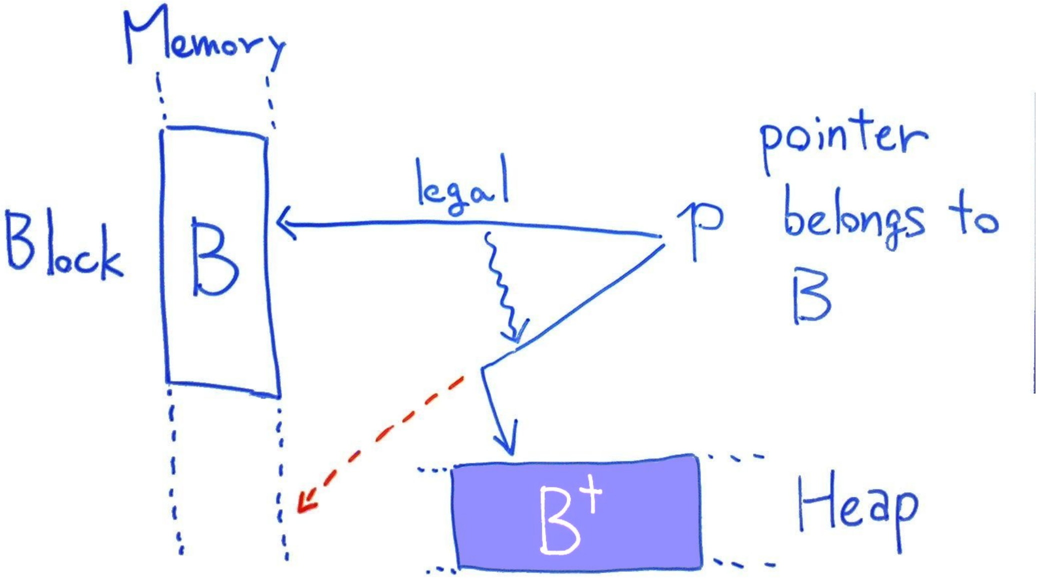
- **Server** programs are still vulnerable against **DoS attacks**.
- **Non server** programs are still **unstable**.
- The problem remains until **bug fixes**.

Programs should **survive attacks** and **continue to work**.
(Attack tolerance)

Attack tolerance

Extending fail-safety to attack tolerance
by boundless memory block[**Rinards**].

- Virtually infinite access range (**no** memory access error)
- Implemented by memory block extension on demand



Attack tolerance by boundless memory block

```
f(char *user, char *pass)
{
  char buf[256]; // This may cause buffer overflow
  sprintf(buf, "%s:%s", user, pass);
  ... /* use of buf */
```

Buffer is extended when buffer overrun detected,

as if it had **larger** size from the beginning.

```
f(char *user, char *pass)
{
  char buf[512]; // Buffer extended on demand
  sprintf(buf, "%s:%s", user, pass);
  ... /* use of buf */
```

Very **natural** recovery from errors.

Wow, then, there is nothing to do!

Answre is of course, **No**.

Attack tolerance needs more security

Careless use of boundless block: **new vulnerability!**

```
f(char *user, char *pass)
{
  char buf[256⇒512];
  sprintf(buf, "%s:%s", user, pass);
  ... /* use of buf for secret data */
  bzero(buf, 256⇏512);
  ... /* use of buf for public data */
```

Secret information of the extended part may **leak** to public.

Our claim

Information flow security is mandatory for attack tolerance:

- The final goal: protection of our **privacy**.
- Attack tolerance may introduce new **security leaks**, since it modifies program semantics.
- Such semantic modification is justified only if **no security leak is assured**.

VITC

VITC = Attack tolerance by
Memory safety + Information flow security

They are mutual:

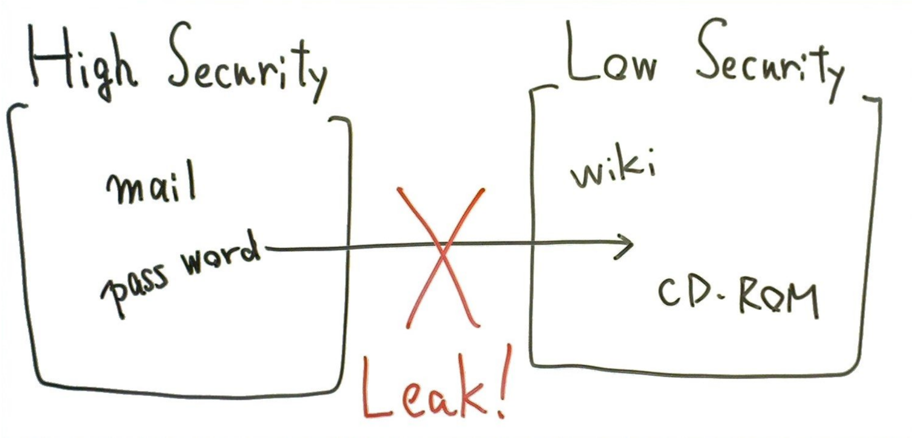
Memory safety with boundless memory block
Justified by **information flow analysis**.

Information flow security by static typing
Requires **memory safety**.

Information flow analysis by security typing for C

Information flow based security

Track the flow of secure information in the program and detect suspicious leak of secrecy.



Static typing

A type-based approach: **security typing [Volpano, Smith]**.

Non-interference

Modifications of higher secret information must not be observed as the change of results of lower secrecy.

Why typing?

Since it is **automatic** D.I.Y. security:

You do **not** need:

- Ph.D to use theorem prover
- Knowledge of internals of the program

All you need are:

- The **source**
- and the **compiler**
- security **policy** (small specifications of privacy)
- and some amount of **luck**.

Security typing

Similar to the normal typing, but they talk about **secrecy**:

Security labels $\ell \in (\mathcal{L}, \leq)$

Form a lattice $\mathcal{L}$, such as $\{L, H\}$ where $L \leq H$.

Type attached with security labels

$$\texttt{password} : \texttt{string}^H \qquad \texttt{3.141592} : \texttt{float}^L$$

Typing rules track down information flow

$$\frac{\Gamma \vdash e_1 : \mathrm{int}^H \qquad \Gamma \vdash e_2 : \mathrm{int}^L}{\Gamma \vdash e_1 + e_2 : \mathrm{int}^H}$$

Security typing in C: expressions

C as a memory safe, imperative language:

e	$::=$		expressions
	$\mid$	$n : t$	integer
	$\mid$	$x : t$	variable
	$\mid$	$*e : t$	dereference
	$\mid$	$*e = e : t$	update
	$\mid$	$(t)e : t$	cast
	$\mid$	$e + e : t$	addition
	$\mid$	$\text{new}(t) : t$	boundless allocation
	$\mid$	$\text{let } x : t = e \text{ in } e : t$	let binding

Security typing in C: types

Types are lists of security labels:

$$t ::= \ell \mid t; \ell$$

Ignoring the normal part of types:

With normal part	$\text{int}^H\ \text{ptr}^L\ \text{ptr}^L$
Formal type	$H;\ L;\ L$

- The normal part C typing is boring.
- Functional types are treated separately.
- Structure members have the same type.

Types and casts

Cast has been a big troublemaker of C programming.

Cast is a troublemaker also in security typing.

Modification of security labels by casts breaks **non-interference**:

$$\begin{array}{rcll} e & : & \mathrm{int}^{H}\ \mathrm{ptr}^{L} & \\ (\mathrm{int})e & : & \mathrm{int}^{L} & ? \\ (\mathrm{int}*)(\mathrm{int})e & : & \mathrm{int}^{?}\ \mathrm{ptr}^{L} & ??? \end{array}$$

Solution: we do not allow casts of security labels.

Types and casts #2

Cast can change the normal part of types,
but **not** security labels:

$$\begin{array}{rcr} e & : & \mathrm{int}^H\ \mathrm{ptr}^L \\ (\mathrm{int})e & : & ?^H\ \mathrm{int}^L \\ (\mathrm{int}*)(\mathrm{int})e & : & \mathrm{int}^H\ \mathrm{ptr}^L \end{array}$$

Even a mere integer type may have much longer security labels:

$$?^H\ ?^H\ ?^H\ ?^L\ \mathrm{int}^L \qquad (H;\ H;\ H;\ L;\ L)$$

Types and casts #3

Sometimes label sequence becomes **infinite**

```
int *p; // $t;\ell$
int length = 0;
...
while (p != NULL){
  length++;
  p = (int*)*p; // $t;\ell = t$
}
```

Such types will be expressed as fixed points: $\mu\alpha.\alpha;\ell$.

Subtyping

$(\leq)$ for labels is extended to **subtype** relation:

$$\frac{\ell \leq \ell'}{\ell \leq \ell'} \qquad\qquad \frac{\ell \leq \ell'}{t; \ell \leq t; \ell'}$$

The content type t of pointer types $t; \ell$ is invariant, just like the subtyping of references.

Typing rules

Quite straightforward (since we have omitted many):

$$\Gamma \vdash n : t \qquad \frac{t \in \Gamma(x)}{\Gamma \vdash x : t} \qquad \frac{\Gamma \vdash e : t' \quad t' \leq t}{\Gamma \vdash e : t}$$

$$\frac{\Gamma \vdash e : t'; \ell \quad t' \leq t \quad \ell \lhd t}{\Gamma \vdash *e : t} \qquad \frac{\Gamma \vdash e_1 : t; \ell \quad \Gamma \vdash e_2 : t \quad \ell \lhd t}{\Gamma \vdash *e_1 = e_2 : t}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash (t)e : t} \qquad \frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 + e_2 : t} \qquad \Gamma \vdash \mathrm{new}(t) : t; \ell$$

$$\frac{\Gamma \vdash e_1 : t \quad \Gamma[x \mapsto t] \vdash e_2 : t'}{\Gamma \vdash \mathrm{let}\ x = e_1\ \mathrm{in}\ e_2 : t'} \qquad \frac{\ell \leq \ell'}{\ell \lhd \ell', \quad \ell \lhd t; \ell'}$$

Typing rules #2

Integer has any sequence of labels for interaction with pointers:

$$\Gamma \vdash n : t$$

Cast does **nothing**:

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash (t)e : t}$$

new(t) has a pointer type $t; \ell$:

$$\Gamma \vdash \mathrm{new}(t) : t; \ell$$

More on typing (what I omitted today)

Implicit flow so called *pc*

Stop security leaks due to conditionals:

$\texttt{if secret}^{H}\texttt{ then x = 0}^{L}\texttt{ else x = 1}^{L}$

Function types with effects

For flows produced by side effects inside functions

Polymorphism

For genericity of functions

Type inference

Constraint based system

Future work

Measure impact of the new typing

Cast typing may be too restrictive.

- Need to check using various examples.
- **Allowing casts** of security types with **dynamic typing**.

Interaction with OS security information

Dynamic security policies obtained from OS

Dynamic checking

Risk of new implicit information flow by run-time checks.

Dependent types will be one of the keys.

Yet more: Auto-securing of C programs

Memory safe C compilers produce memory safe programs **without any fix** of the C source code.

Possible also for information flow security?

Idea: Closing security leaks from H to L by replacing secret data by something lower:

`let f x = print "your message is "; print x`

`f "hello"`^{L} $\Rightarrow$ `your message is hello`

`f password`^{H} $\Rightarrow$ `your message is <secret>`

`let f x`$^{\ell}$ `= print "your message is ";`

`if` $\ell = L$ `then print x else print "<secret>"`

Conclusion

VITC is C program compilation:

Memory safe

No more memory vulnerability attacks such as buffer overflow

Attack tolerance

Programs can survive attacks.

Information flow security

Programs never leak secret information,
even if they are attacked.