Compiling C Programs into a Strongly Typed Assembly Language

Takahiro Kosakai
Toshiyuki Maeda
Akinori Yonezawa
(Univ. of Tokyo)

Brief Overview

- We propose a method to guarantee the memory safety of C programs: **Compile C programs into a typed assembly language**
- Our contribution:
  1. Designed a typed assembly language **$CTAL_0$**
  2. Implemented an experimental compiler from C to $CTAL_0$

Outline

- Background
- Our Language: $CTAL_0$
- Implementation of Compiler
- Related Work
- Conclusion & Future Work

Outline

- **Background**
- Our Language: $CTAL_0$
- Implementation of Compiler
- Related Work
- Conclusion & Future Work

Background (1/2)

- C is a classic programming language developed 35 years ago
- Even today, C is popular and a lot of security-critical software is written in C
  - Almost all of operating systems
  - Web servers
  - etc...

Background (2/2)

- However, C programs often have **memory-related** bugs that can easily lead to security vulnerabilities
  - Buffer overflow, dangling pointer, double free, ...
- About 40% of recent Linux kernel vulnerabilities are caused by memory-related bugs [1]

[1] SecurityFocus vulnerability database, January - June 2007.

Why so many memory bugs?

- Because C is **not** a **memory safe** language
  - E.g., No protection against out-of-bounds array access
    - **`array[i++] = 123;`**
    - May cause buffer overflow
- **Ensuring memory safety is a crucial step for ensuring security of software**

Existing Work

- There are several schemes to certify memory safety of C programs
- Two of such schemes are CCured [1] and Fail-Safe C [2]
  - Make C programs memory safe by program transformations
    - Inserting runtime bounds-checks, etc.

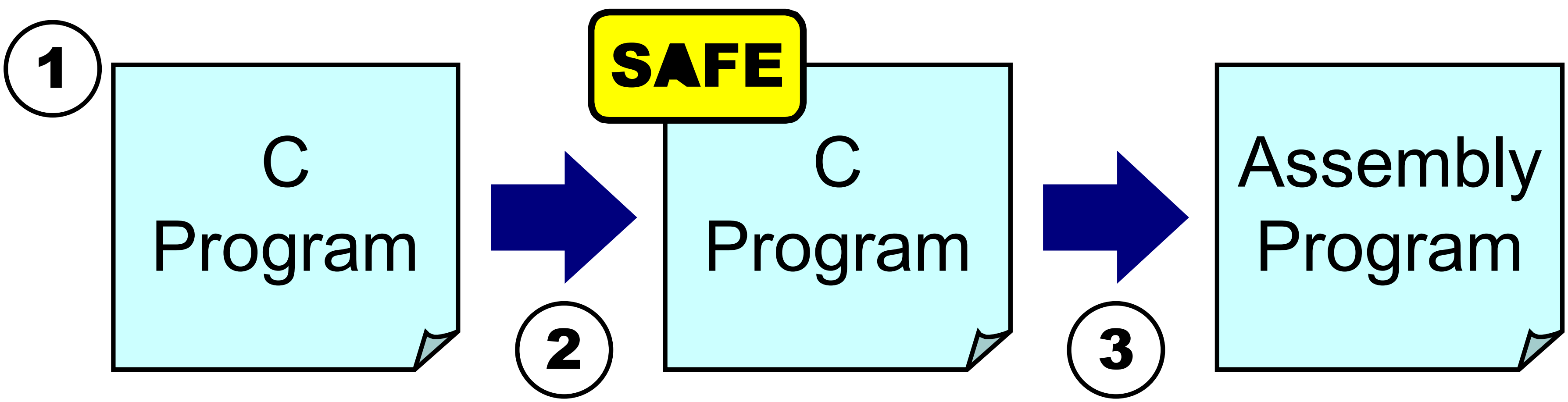
[1] G.C.Necula et al., *CCured: Type-safe retrofitting of legacy software*, TOPLAS '05.
[2] Y.Oiwa et al., *Fail-safe ANSI-C compiler: An approach to making C programs secure*, ISSS '02.

Problem with Existing Schemes

- They are source-to-source translators

Procedure for ensuring memory safety

1. Get source code of software
2. Apply the schemes to get certified source code
3. Compile it with conventional compiler (e.g., GCC)

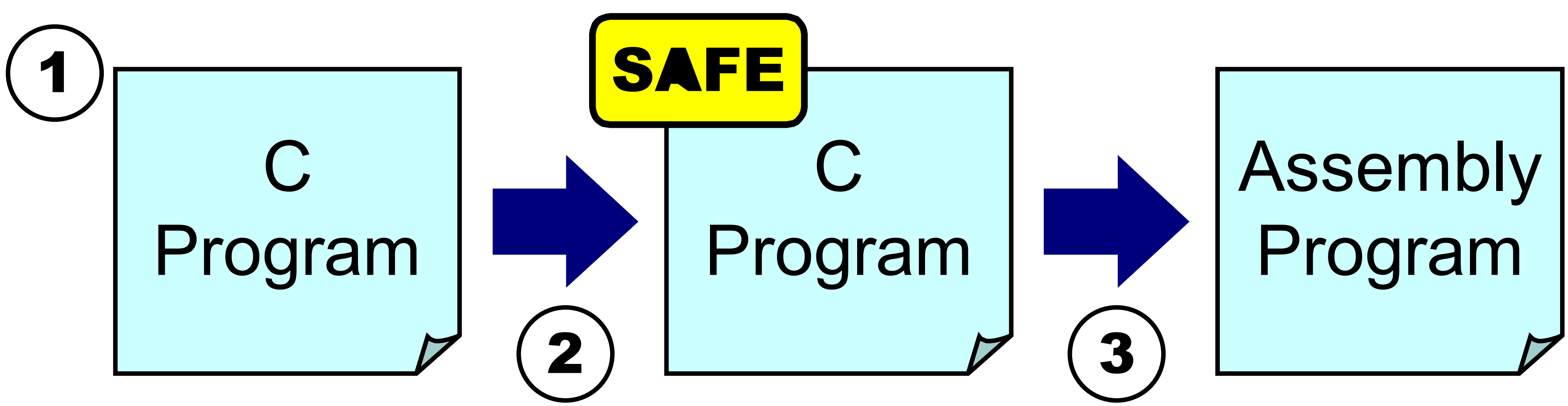


Problem with Existing Schemes

- They are source-to-source translators

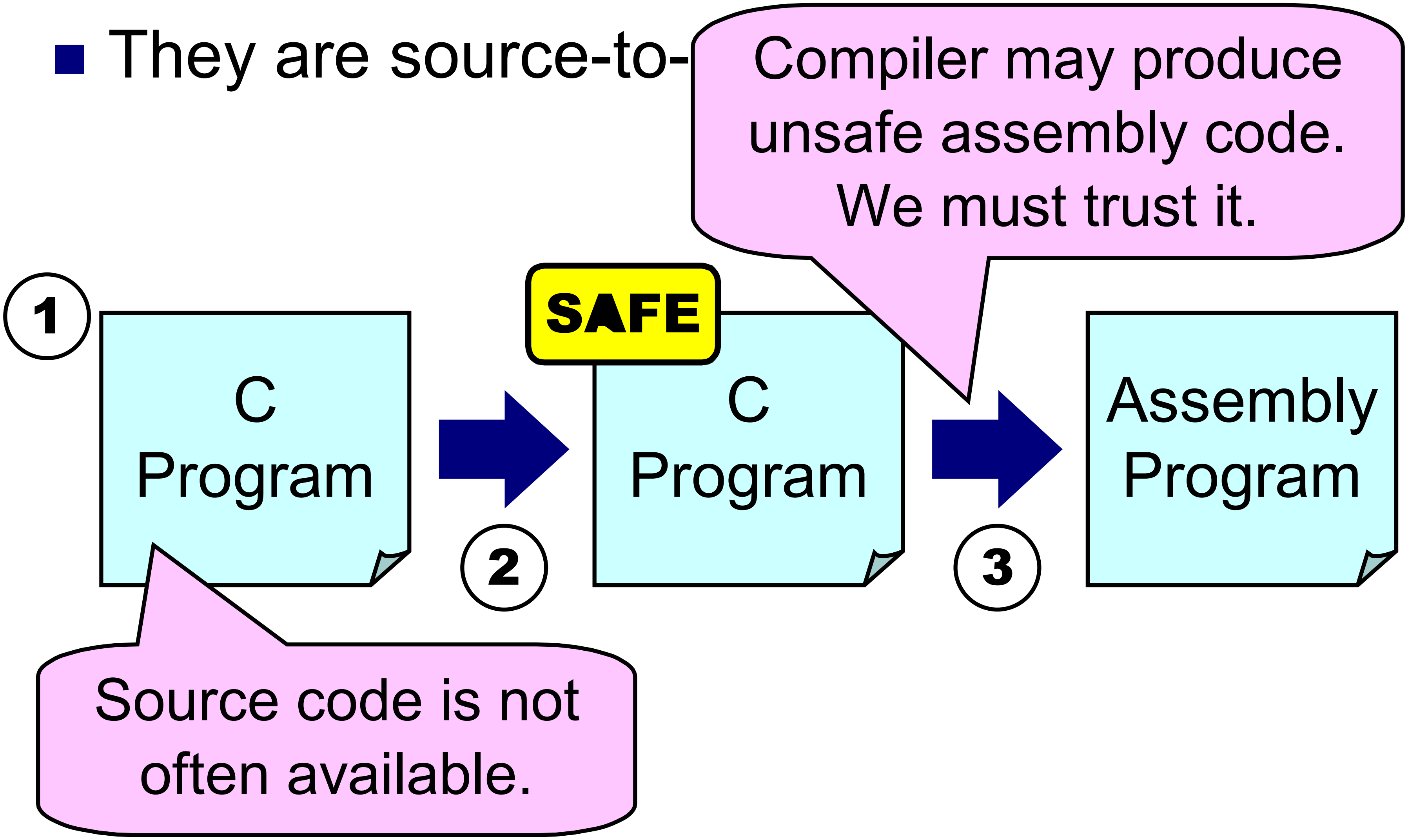
<u>Procedure for ensuring memory safety</u>

1. Get source code of software
2. Apply the schemes to get certified source code
3. Compile it with conventional compiler (e.g., GCC)



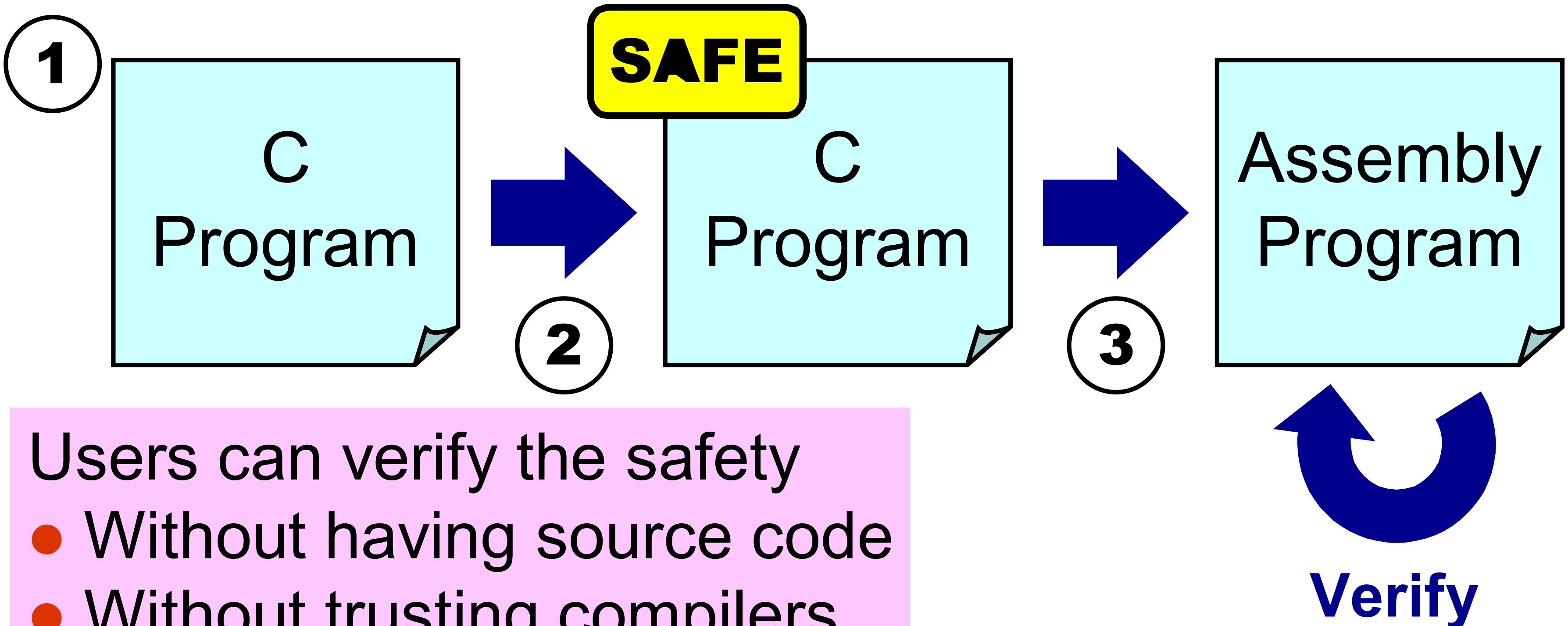
Problem with Existing Schemes

- They are source-to-



Our Approach

- Lower down the certification phase to **assembly-code level**



Users can verify the safety
- Without having source code
- Without trusting compilers

How to certify assembly code?

- We selected Typed Assembly Language (TAL) [1] as our starting point
- TAL is an assembly language equipped with a strong static type system
  - Well-typed assembly programs are memory safe
  - **Certification of memory safety can be done by simple type-checking**

[1] G.Morrisett et al., *From system F to typed assembly language*, POPL '98.

How to certify assembly code?

- TAL is not suitable for compiling from C
  - Its target is type-safe languages like ML
  - Operations that are not type-safe are not considered

- We propose an extension of TAL, called **$CTAL_0$**, which is aimed at C
  - It can handle lower-level issues such as non-type-safe casts and NULL pointers

Outline

- Background
- **Our Language: $CTAL_0$**
- Implementation of Compiler
- Related Work
- Conclusion & Future Work

A flavor of $CTAL_0$

- Simple program named "**`inc`**" that loops infinitely, incrementing the value of register **`r1`**

{ **`inc`** $\rightarrow \forall(x)$. code[**`r1`**: Int(x)] }

```
inc:
   add r1, 1
   apply inc, (x → x + 1)
   jmp inc
```

Type annotation & Pseudo-instruction

Ordinary assembly code

A flavor of $CTAL_0$

Annotation of ***heap type***
Meaning: "At address **`inc`**, there is a value of type $\forall(x)$. code[**`r1`**: Int(x)]"

{ **`inc`** $\rightarrow \forall(x)$. code[**`r1`**: Int(x)] }

`inc:`

code[**`r1`**: Int(x)] denotes instruction sequences that are executable if **`r1`** has type Int(x)

Int(x) denotes integers whose value is exactly equal to x

Type-Checking in $CTAL_0$ (1/5)

- Type-checking proceeds by manipulating:
  - Γ : Type of each register
  - Ψ : Type of each value in the heap
  - ϕ : Valid logical formula over variables

{ **inc** $\rightarrow \forall(x)$. code[**r1**: Int(x)] }

```
inc:
  add r1, 1
  apply inc, (x → x + 1)
  jmp inc
```

Type-Checking in $CTAL_0$ (2/5)

- First, Γ, Ψ, ϕ are initialized according to the heap type annotation

Γ = [**r1**: Int(x)]
Ψ = { **inc** $\rightarrow \forall(x)$. code[**r1**: Int(x)] }

ϕ = True

{ **inc** $\rightarrow \forall(x)$. code[**r1**: Int(x)] }

```
inc:
   add r1, 1
   apply inc, (x → x + 1)
   jmp inc
```

Type-Checking in $CTAL_0$ (3/5)

- When checking **add**, Γ(**r1**) is updated so that it reflects the effect of **add**

Γ = [**r1**: Int(x + 1)]
Ψ = { **inc** $\rightarrow$ $\forall$(x). code[**r1**: Int(x)] }

ϕ = True

{ **inc** $\rightarrow$ $\forall$(x). code[**r1**: Int(x)] }

```
inc:
  add r1, 1
  apply inc, (x → x + 1)
  jmp inc
```

Type-Checking in $CTAL_0$ (4/5)

- Next **`apply`** pseudo-instruction instantiates the polymorphic type of **`inc`** in Ψ

Γ = [**`r1`**: Int(x + 1)]
Ψ = { **`inc`** $\rightarrow$ code[**`r1`**: Int(x + 1)] }

ϕ = True

{ **`inc`** $\rightarrow$ $\forall(x)$. code[**`r1`**: Int(x)] }

```
inc:
  add r1, 1
  apply inc, (x → x + 1)
  jmp inc
```

Type-Checking in $CTAL_0$ (5/5)

- Type-checking **`jmp`** is to check the current state matches the type of jump destination

Γ = [**r1**: Int(x + 1)]

match ➔ OK

Ψ = { **inc** $\rightarrow$ code[**r1**: Int(x + 1)] }

ϕ = True

{ **inc** $\rightarrow$ $\forall(x)$. code[**r1**: Int(x)] }

```
inc:
  add r1, 1
  apply inc,
➔ jmp inc
```

($x \rightarrow x + 1$)

This program successfully passes the type-checking

Extensions to TAL

- Key extensions in $CTAL_0$ that make it suitable for compiling from C
  - Two characteristic types
    - **Untyped array types**: Array(i)
    - **Guarded types**: $\phi\ ?\ \tau_1 : \tau_2$
  - Support of **byte addressing**

Untyped Array Types (1/2)

- Motivation
  - How to deal with non-type-safe casts?

```
int arr[3];
((char *)arr)[2] = 'A';
```

  - **`arr`** is an array of **`int`**, but it is also used as an array of **`char`**
  - Thus **`arr`** cannot have type "Array(**`int`**)" in a strongly typed language

Untyped Array Types (2/2)

- Denotes "untyped" memory blocks
  - $CTAL_0$ type system imposes no restrictions on their contents
  - Original TAL arrays are typed (all elements must have uniform type)
- Can deal with non-type-safe casts

```
int arr[3];
((char *)arr)[2] = 'A';
```

  - **arr** can be an untyped array

Guarded Types (1/2)

- Motivation
  - How to deal with NULL pointers?

$$\{\,\mathbf{p} \rightarrow \mathbf{int}\,\}$$

  - This heap type means: "A value of type **int** exists at address **p**"
  - We want to allow **p** to be NULL

Guarded Types (2/2)

- Guarded type $\phi\ ?\ \tau_1 : \tau_2$ is ...
  - equal to type τ_1, if logical formula ϕ is true
  - equal to type τ_2, if logical formula ϕ is false
- Can represent "maybe-NULL" pointers

$$\{\, \mathtt{p} \rightarrow (\mathtt{p} \mathrel{!=} 0)\ ?\ \mathtt{int} : \langle\rangle \,\}$$

  - This heap type means: "If **`p`** is non-zero, then an **`int`** value is at address **`p`**" — i.e., **`p`** is either NULL or a pointer to **`int`**

Byte Addressing

- All bytes in memory blocks are accessible
  - Original TAL only considers word-size access
  - This extension itself is straight-forward, but slightly complicates formalization and proof of language safety

Formal Properties

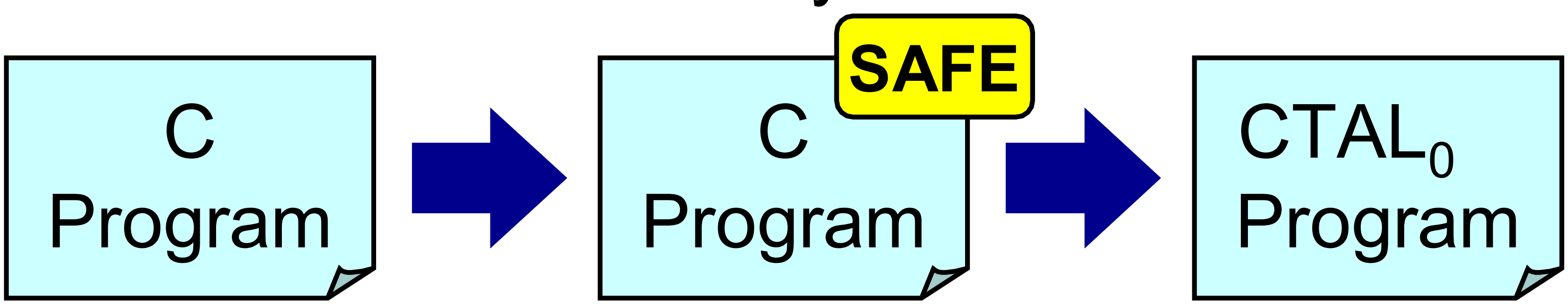
- Well-typed $CTAL_0$ programs are ...
  - Memory safe
    - Will ***never*** perform wrong memory accesses
  - Control-flow safe
    - Will execute ***only*** valid instructions

Outline

- Background
- Our Language: $CTAL_0$
- **Implementation of Compiler**
- Related Work
- Conclusion & Future Work

Compiling Strategy

- It is impossible to directly compile unsafe C programs into safe $CTAL_0$
- Our compiler takes 2 steps
  - First, establish safety by source-level program transformation
  - Then, compile it to $CTAL_0$, preserving the established safety



Program Transformation

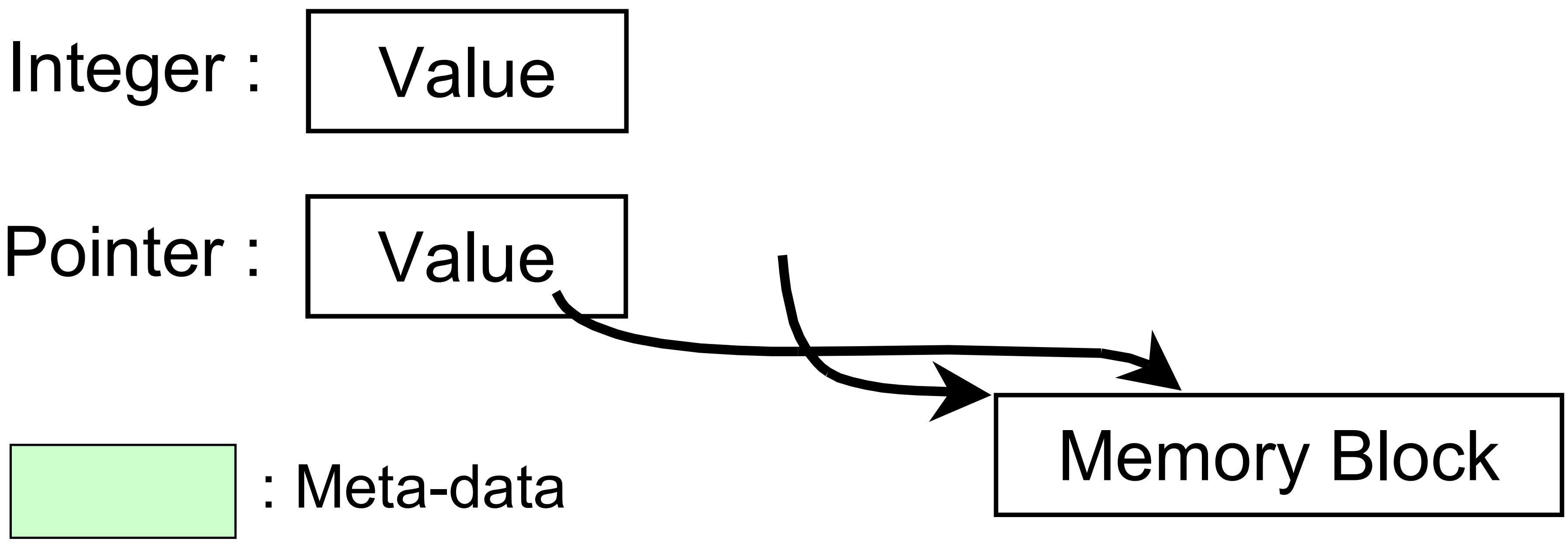
- Basically, insert bounds-checks before every memory dereference operation
  - Program will abort before doing wrong memory accesses
- To obtain correct bounds information for each pointer, we add some **meta-data**
  - Using several techniques of Fail-Safe C [1]

[1] Y.Oiwa et al., *Fail-safe ANSI-C compiler: An approach to making C programs secure*, ISSS '02

Transformation: Step 1 / 3

- Extend each integer and pointer to 2 words
  - **"Fat integer"**

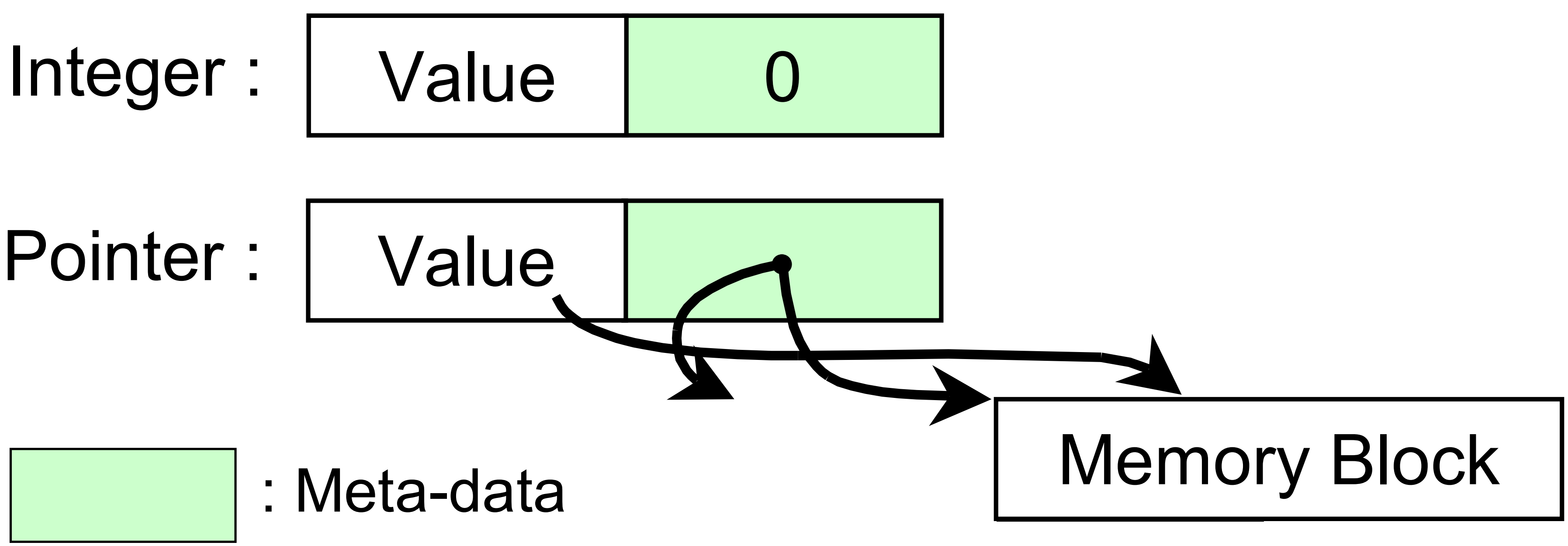
Casts between integers and pointers are freely possible



Transformation: Step 2 / 3

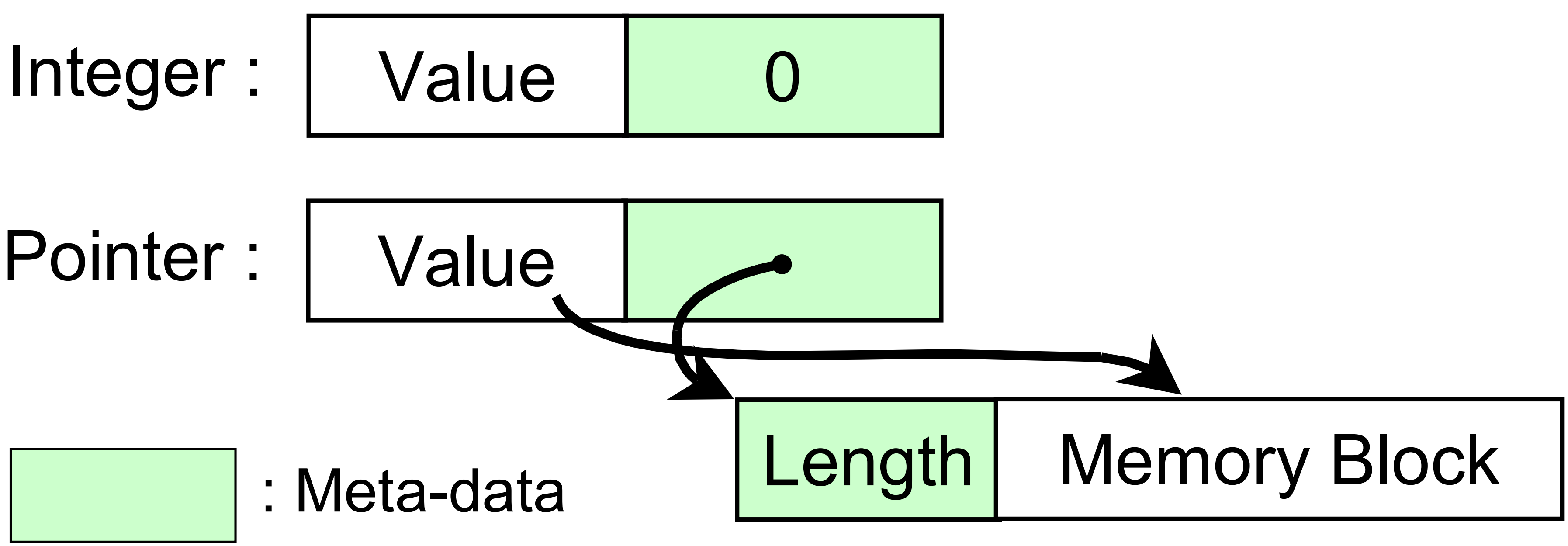
- Attach to every memory block its length

Bounds checks are now possible



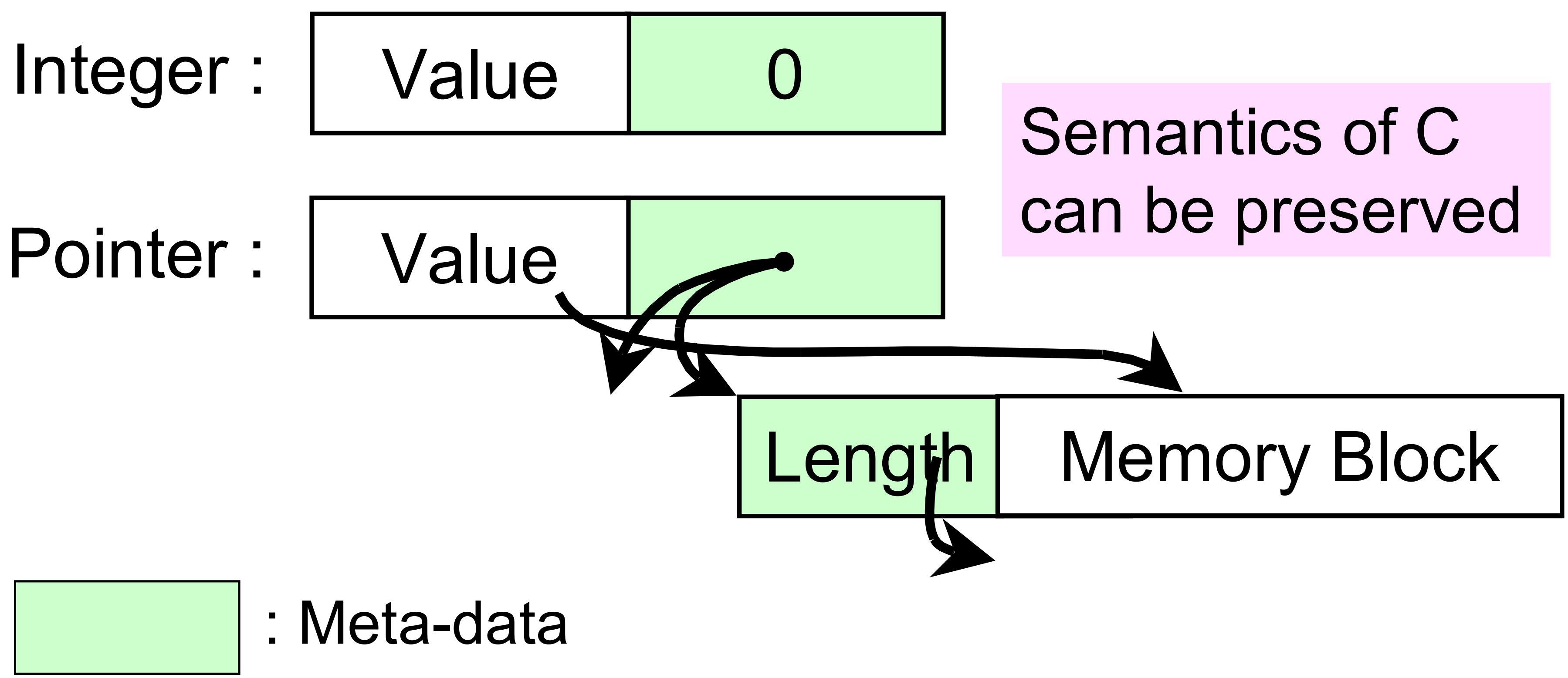
Transformation: Step 3 / 3

- Since integers are doubled in size, memory blocks should also be doubled



Transformation: Step 3 / 3

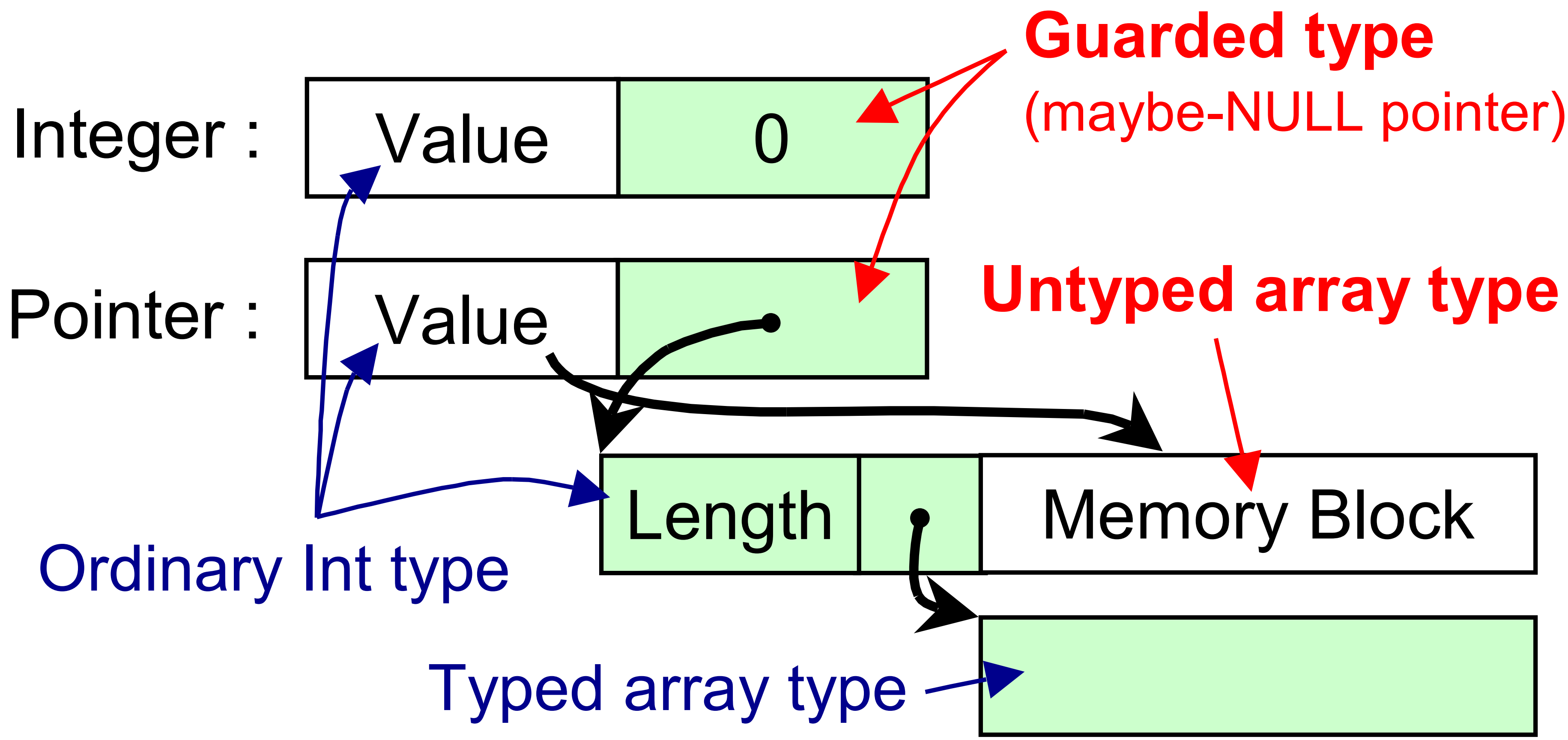
- Since integers are doubled in size, memory blocks should also be doubled



Semantics of C can be preserved

Typing

- This structure can be typed in $CTAL_0$



Supported Features

- Casts between integers and pointers
  - Using fat integers
- Arrays, structures, unions
  - By treating them as untyped arrays
- Function pointers
- Dynamic memory allocation (`malloc`)

Preliminary Experiment

- We have successfully compiled ...
  - Simple insertion sort program
  - glibc's quick sort function
    - Heavily uses function pointers
  - Huffman-code compressor
  - xvgif (GIF image decoding library)
    - Successfully detected and prevented a known buffer overflow bug
  - etc.

Outline

- Background
- Our Language: $CTAL_0$
- Implementation of Compiler
- **Related Work**
- Conclusion & Future Work

Related Work

- CCured [1], Fail-Safe C [2]
  - Ensure memory safety of C programs
  - Good runtime performance
  - Source-to-source translators; there is no safety guarantee on assembly code

[1] G.C.Necula et al., *CCured: Type-safe retrofitting of legacy software*, TOPLAS '05.
[2] Y.Oiwa et al., *Fail-safe ANSI-C compiler: An approach to making C programs secure*, ISSS '02.

Related Work

- Typed Assembly Language [1]
  - Basis of our language $CTAL_0$
  - Mainly aimed at compiling from ML-like type-safe functional languages
- TALx86 [2]
  - TAL for Intel IA-32 architecture
  - Mainly aimed at compiling from Popcorn (an imperative, safe language)

[1] G.Morrisett et al., *From system F to typed assembly language*, POPL ’98.
[2] G.Morrisett et al., *TALx86: A realistic typed assembly language*, WCSSS ‘99

Outline

- Background
- Our Language: $CTAL_0$
- Implementation of Compiler
- Related Work
- **Conclusion & Future Work**

Conclusion

- We have proposed a new typed assembly language $CTAL_0$, based on TAL
  - Guarantees memory safety at assembly-code level
- We have implemented an experimental compiler from C to $CTAL_0$
  - Supports free intermixing of integers and pointers, arrays, structures, unions, and function pointers

Future Work

- Improve compiler implementation
  - Optimization and static analysis to remove redundant dynamic checks
  - Binary compatibility with existing libraries
- Enrich $CTAL_0$'s type system
  - Support explicit memory deallocation
  - Support linking of object files

Fin.