arXiv:2302.01811v1 [cs.CR] 3 Feb 2023

CheckedCBox：使用 Checked C 实现增量式空间内存安全的类型引导程序分区（扩展版）

Liyi Li*、Arunkumar Bhattar*†、Le Chang、Mingwei Zhu 和 Aravind Machiry†马里兰大学 †普渡大学

摘要

空间内存安全违规仍是 C 程序面临的一大问题。Checked C 是一种安全的 C 语言方言，通过添加受检指针类型和注释来扩展 C 语言，从而以向后兼容的方式保证空间内存安全，并允许受检指针与常规（未检查）指针类型混用。然而，未检查代码中的漏洞可能会破坏已检查代码的空间安全保证。我们提出CheckedCBox，它通过使用 受污染类型增强 Checked C 类型系统，从而为 Checked C 添加灵活的类型引导程序分区机制，使程序能够灵活地划分为 c 区域和 u 区域，并确保 u 区域代码不会影响 c 区域的空间安全。我们对类型系统进行了形式化，并证明了类型良好的 CheckedCBox 程序具有不崩溃和非暴露属性。我们以可配置的方式实现了 CheckedCBox，从而能够使用现有的沙箱机制（例如，WebAssembly）无缝执行程序分区。对七个程序的评估表明，CheckedCBox能够有效且高效地划分程序，并防止四种已知漏洞。

1 引言

内存破坏，尤其是空间内存破坏所导致的漏洞，仍然是 C 程序面临的一大问题 [3, 43, 45]，尽管人们已付出诸多努力试图加以防范 [41]。包括 CCured [34],Softbound [32],和 ASAN [39], 在内的多项工业界和研究界工作，探索了如何更好地编译 C 程序并自动强制实施空间安全。这些方法都会带来过高的性能开销，因而不适合部署使用。最近，Elliott 等人 [9] 和 Li 等人 [21] 提出并形式化了 Checked C （一种 C 语言的开源扩展），通过引入新的指针类型来确保程序的空间安全，即已检查（c）指针类型。已检查指针以系统级内存字的形式表示，无需“膨胀”元数据 [8], ，同时确保向后兼容性，即开发者可以在同一程序中使用已检查指针和常规（未检查u）指针。然而，正如我们在第 2.2 节中所解释的，未转换或未检查（u）代码可能会破坏c 区域所提供的保证。我们需要确保作为未检查（u）区域的一部分执行的代码，不会导致已检查（c）区域发生安全违规；为此，需要使用程序分区机制 *[38]*。

现有的此类机制并不适用，因为它们基于进程隔离，开销较高，并且是以数据为中心的（第 2.3 节）。但在我们的场景中，我们希望采用一种低开销的以代码为中心的划分，其中 u 区域代码（或函数）应与 c 区域代码隔离（或分区）。我们还希望该技术能够与 Checked C 的保障共存并兼容，从而使包含c 区域代码的分区能够保持空间安全。

*这些作者对本研究的贡献相同。

在此，我们提出了一种由类型指导的、以代码为中心的程序分区方法。具体而言，我们的系统CheckedCBox扩展了 Checked C 的已检查和未检查指针类型——它们分别表示安全和不安全的程序部分——并引入了运行于隔离沙箱机制上的**污染**（t_*）类型；该系统禁止已检查类型实体与未检查类型实体之间的通信，并通过使用带有额外有效性检查的污染类型，强制实现已检查类型与未检查类型之间的通信。

开发者首先将所需的（即未检查的 u）函数以及函数中使用的指针标记为受污染。然后，CheckedCBox将给定程序划分为两个权限不同的分区（u 和 c 区域）：

- u区域（低权限的污染区域，扩展自 Checked C 中的未受检区域）：此分区包含受污染类型（即函数和指针），且只能访问受污染指针和未检查指针。
- c 区域或安全区域（高权限的未污染或已检查区域）：此分区包含其余（未受污染）的代码和数据，并可完全访问c 区域。c 区域中的函数可以调用u 区域中的任意函数，并访问其中的所有数据，但反之则不行；回调函数除外，我们稍后将对此进行讨论。

c 区域代码作为常规程序执行，而u区域分区将在现有的沙箱环境中运行（例如WASM 沙箱），并通过额外的插桩促进c与u 区域中代码之间的通信。

污点类型与特权分区相结合，可强制实现隔离并提供内存安全，而无需将所有未检查的 C 语言代码转换为 Checked C 代码，因为未检查类型可以保留在u 区域中，而c 区域代码可以访问分配在u 区域中的污点类型实体。尽管内存隔离可以防止直接违规，u 区域代码仍可能通过污点指针影响c 区域，从而发动混淆代理攻击 [24, 35], 例如，在污点指针中使用有效的 c 区域地址。我们的编译器通过动态检查确保污点指针指向u 区域地址空间，从而避免这些攻击。这些检查由我们的编译器静态生成。

总而言之，我们的主要贡献有以下三点。

CheckedCBox 类型系统、形式化体系与编译器。我们提出了一种将污染类型与 Checked C 相结合的类型系统，并提供了额外保证——不崩溃和非暴露保证，即，类型良好的CheckedCBox程序绝不会因空间安全违规而崩溃，并且u 区域代码无法直接观察已检查指针的地址。我们扩展 Checked C 编译器以支持该类型系统，并通过扩展 Checked C 形式化体系 [21] 来提供不崩溃和非暴露保证。我们正式证明了与这两项保证相关的定理，并使用基于模型的随机测试[18] 来验证以下内容之间的模拟关系：CheckedCBox

语义及其编译器形式化体系。据我们所知，CheckedCBox 是首个具有程序分区机制的 C（类）语言及编译器形式化体系。

类型引导的程序分区。我们提出一种类型引导的程序分区技术，用于分离已检查和未检查代码区域，并确保上述保证。我们的模块化设计使我们能够使用现有的沙箱技术来强制实施内存和执行隔离，并在CheckedCBox 编译器中实现污点指针。

支持回调至已检查区域，且不暴露 Checked 指针。尽管我们禁止从已检查区域直接访问未检查区域，但有时确实需要进行此类访问。具体而言，当已检查区域希望向未检查区域提供对某些共享的 Checked 数据的访问时，为实现这一点，我们支持在已检查区域中定义回调函数，并允许未检查区域通过函数指针调用这些函数。然而，若未检查区域知晓已检查区域函数的地址，就会违反非暴露保证，并导致其他攻击 [14]。

我们通过间接寻址来解决这一问题。具体来说，未检查区域不会直接访问已检查区域的回调函数，而是通过污点类型的受保护跳板函数访问它们；该函数会将执行导向相应的回调函数。此外，跳板函数本身通过不透明索引而非其虚拟地址进行引用，这通过现有的沙箱化技术实现。

我们通过对七个大型真实世界程序进行分区，评估了 CheckedCBox [1] 的有效性。评估结果表明，CheckedCBox 提供了一种灵活、低开销的程序分区机制，并能保证空间内存安全。

2 背景与动机

这里简要介绍 Checked C 及 CheckedCBox 的设计动机。

2.1 Checked C

Checked C 于 2015 年由微软研究院启动开发，但在 2021 年底分支出来，目前由安全软件开发项目 (SSDP) 积极维护。详情请参阅先前的概述 [9] 和形式化描述 [21]。

已检查指针类型。Checked C 引入了三种已检查指针：

- `_Ptr<T>` (*ptr*) 表示一个指针，该指针要么为空值，要么指向一个类型为 T 的单个对象。
- `_Array_ptr<T>` (*arr*) 表示一个指针，该指针要么为空值，要么指向一个包含 T 个对象的数组。数组宽度由一个 边界表达式定义，后文将对此进行讨论。
- `_NT_Array_ptr<T>` (*ntarr*) 类似于 `_Array_ptr<T>`，不同之处在于，边界表达式定义的是数组宽度的最小值——上界之后还可能存在其他对象，直到空值终止符为止。

两种 *arr* 和 *ntarr* 指针都具有相关联的边界，用于定义该指针所引用的内存范围。声明边界的三种不同方式及其对应的内存范围如下：

`_Array_ptr<|T|> p: count(n)`	$[p,\ p + \text{sizeof}(T) \times n)$
`_Array_ptr<T> p: byte_count(b)`	$[p, p + b)$
`_Array_ptr<T> p: bounds(x, y)`	$[x, y)$

也可以为 *ntarr* 声明边界，但内存范围可以继续向右延伸，直到遇到 空值 终止符（即 空值 不在边界范围内）。

确保空间内存安全。Checked C 编译器会对已检查指针的加载和存储插入检测，以确认指针非空，并进一步确认对 *arr* 和 *ntarr* 指针的访问位于其指定边界内。例如，在代码 `if (n>0) a[n-1] = ...` 中，写入通过地址 α = `a + sizeof(int)×(n-1)` 进行。如果 `a` 的边界为 `count(u)`，插入的检查会在解引用前确认 `a` $\leq \alpha <$ `a + sizeof(int)×u`。检查失败时会抛出异常。通常，LLVM 可以优化掉插入的检查，从而几乎不会产生运行时开销 [7]。

向后兼容性。Checked C 与传统 C 向后兼容，因为所有传统代码都能通过类型检查并成功编译。不过，编译器只会对已检查指针添加上述空间安全检查。代码尚未完全移植时，空间安全保证是有限的。具体而言，只有出现在已检查代码区域（`c` 区域）中的代码才能保证具有空间安全性。`c` 区域可以通过 `checked` 关键字，在文件、函数或单个代码块级别指定。[2] 在 `c` 区域中，不允许使用传统指针和某些不安全习惯用法（例如，可变参数函数调用）。

将 C 语言转换为 Checked C。由于转换需要进行语义推理以及重构等其他修改，因此无法完全自动地将 C 语言代码转换为 Checked C。更多详情请参见附录 A.1。

2.2 u区域不提供安全保证

对于完全转换的程序，Checked C 可提供空间安全保证；也就是说，这些程序只使用已检查类型，不使用常规指针类型。部分注释程序只有在已检查指针不与任何未检查指针通信时，才能享有空间安全保证。例如，在下面的示例中，函数`func` 只使用已检查指针，因此不会发生空间安全违规。然而，其他未转换的代码区域（或不安全区域）可能会影响安全区域中的指针，并破坏某些假设，从而导致漏洞，跨语言攻击 [28]已证明了这一点。

尽管存在无责证明 [21, 36] 适用于 Checked C，但它并未声称 C 语言区域中不会发生空间安全违规，而是声称C 语言区域不会因任何空间安全违规而受到指责。请看以下示例：

```
1  // c region code
2  int func(array_ptr<char> p : count(5)) {
3  ..p[4]..4   }5   // u region code6   ...
7  str = "he";8 ...
9  func(assume_bounds_cast<char>(str, 5));
```

这里，`c` 区域中的函数 `func` 期望指向包含五个元素的缓冲区的指针，但`u` 区域中的代码以包含两个元素的缓冲区调用该函数（第 9 行）。这导致`c`区域发生空间安全违规（`a`），但责任或根本原因当然在`u` 区域（）。此外，由于`c` 和`u` 区域在同一地址空间中执行，例如，缓冲区溢出）`u` 区域中的空间内存破坏可能会导致整个程序崩溃，即使`c`

[1] 我们的实现已在 https://github.com/REDACTED 开源。

[2] 你也可以在已检查区域内指定未检查区域（`u` 区域）。

```
// Entry point
void server_loop(int sock_fd) {
 unsigned b_z;
 struct queue *p;
 ...
 while(1) {
  ...
  p = malloc(sizeof(struct queue) * b_z);
  ...
  if (handle_request(sock_fd)) {
    ...
  }
 }
}

int handle_request(int sock_fd) {
 char buff[MAX_MSG_SIZE];
 int rc = -1;
 size_t r_len;
 r_len = read_msg(sock_fd, buff,
                  MAX_MSG_SIZE);
 if (r_len > 0 &&
     r_len < MAX_MSG_SIZE) {
  switch(buff[0]) {
   case REQ1:
    rc = process_req1(buff, r_len);
    break;
   case REQ2:
    rc = process_req2(buff, r_len);
    break;
   ...
  }
 }
 return rc;
}
```

```
size_t read_msg(int sock_fd, char *msg,
                size_t sz) {
 size_t nr;
 nr = read(sock_fd, (void*)msg, sz);
 ...
 一个 // complex code
 return nr;
}

int process_req1(char *msg, size_t m_l) {
 int rc = -1, i;
 if (m_l > MIN_SIZE) {
   sscanf(msg, "%d", &i);
   msg[i] = ...
 }
 一个 // complex code
 return rc;
}

int process_req2(char *msg, size_t m_l) {
 size_t i = 0, j = 0;
 int rc = -1;
 char anum[MAX_MSG_SIZE] = {0};
 if (msg) {
  while(i < m_l && j < MAX_MSG_SIZE-1) {
   ...
   if (isalnum(msg[i]))
    anum[j++] = msg[i];
   i++;
  }
  rc = process_data(anum);
 }
 return rc;
}
```

清单 1：存在任意内存写入漏洞的简单网络服务器 C 程序片段，漏洞由 标出。

```
// Entry point
void server_loop(int sock_fd) {
 ...

_Array_ptr<struct queue> p : count(b_z) = NULL;
...                                        }

int process_req2(_Array_ptr<char> msg: count(m_l),
                 size_t m_l) {                  ...
char anum _Checked[MAX_MSG_SIZE] = {0};
...                                    }
```

清单 2：清单 1 中的指针已通过手动方式或 3C 等自动化工具标注为受检类型。

区域。我们需要一种隔离机制，以确保作为u 区域的一部分执行的代码不会违反c 区域中的安全保证。

2.3 程序分区

程序分区 [38] 是一种广为人知的技术，用于将程序划分为多个相互隔离的部分或分区。该领域已有大量研究。[1, 5, 22, 23, 42]这些技术大多以数据为中心 [22, 23], , 即由程序数据驱动分区。具体而言，对于程序中的敏感数据，目标是根据某个函数是否能够访问这些敏感数据，将函数划分为两部分或两个分区。这些方法的性能开销主要由封送成本决定，并且取决于敏感数据的使用方式。Glamdring [22] 和 PtrSplit [23], 等最先进方法的开销高达 37% 至 163%，难以接受。

3 概述

图 1 展示了 CheckedCBox 各个组件之间的交互。给定经过适当注释的源程序和沙箱配置，CheckedCBox 会生成一个可执行文件，使所有污染函数和数据都位于沙箱中

```
size_t read_msg(int sock_fd, char *msg,
                size_t sz) _Tainted    {…}

int process_req1(char *msg, size_t m_l) _Tainted
                                                { }
...
```

代码清单 3：代码清单 1 中受污染函数的初始注释。

```
int handle_request(int sock_fd) {
 char buff[MAX_MSG_SIZE] _Tainted ;
 ...
}

int process_req2( _T_Array_ptr<char> msg: count(m_l),
                 size_t m_l) {
 ...
}
```

代码清单 4：根据类型规则为代码清单 1 添加的其他受污染指针注释。

```
// Entry point
void server_loop(int sock_fd) {
 unsigned b_z;
 _Array_ptr<struct queue> p :
                  count(b_z) = NULL;
 ...
 while(1) {
  ...
  p = malloc(sizeof(struct queue) * b_z);
  ...
  if (handle_request(sock_fd)) {
    ...
  }
 }
}

int handle_request(int sock_fd) {
 char buff[MAX_MSG_SIZE] _Tainted;
 int rc = -1;
 size_t r_len;
 r_len = read_msg(sock_fd, buff,
                  MAX_MSG_SIZE);
 if (r_len > 0 &&
     r_len < MAX_MSG_SIZE) {
  switch(buff[0]) {
   case REQ1:
    rc = process_req1(buff, r_len);
    break;
   case REQ2:
    rc = process_req2(buff, r_len);
    break;
   ...
  }
 }
 return rc;
}
```

```
size_t read_msg(int sock_fd, char *msg,
                size_t sz) _Tainted {
size_t nr;
nr = read(sock_fd, (void*)msg, sz);
...
a // complex code
return nr;
}

int process_req1(_T_Array_ptr<char> msg :
                 count(m_l), size_t m_l)
                 _Tainted {

int rc = -1, i;
if (m_l > MIN_SIZE) {
  sscanf(msg, "%d", &i);
  msg[i] = ...
}
// complex code
 return rc;
}

int process_req2(_T_Array_ptr<char> msg :
                 count(m_l),
                 size_t m_l) {
 size_t i = 0, j = 0;
 int rc = -1;
 char anum _Checked[MAX_MSG_SIZE] = {0};
 if (msg) {
  while(i < m_l && j < MAX_MSG_SIZE-1) {
   ...
   if (isalnum(msg[i]))
    anum[j++] = msg[i];
   i++;
  }
  rc = process_data(anum);
 }
 return rc;
}
```

清单 5：清单 1 的最终注释程序，包含受污染和已检查类型。被突出显示的函数将在沙

而且C 语言区域（非沙箱区域）不会受到沙箱中代码的影响。这里，我们从开发者的角度简要介绍CheckedCBox，并在后续章节介绍各个组件的详细信息。

3.1 运行示例

为了说明 CheckedCBox 编译器所需的更改，代码清单 1 展示了一个简单网络服务器的删节版 C 语言代码，该服务器以 server_loop 作为其入口点（）。服务器在循环中运行并调用 handle_request，该函数负责处理网络请求。函数 handle_request 通过套接字读取数据

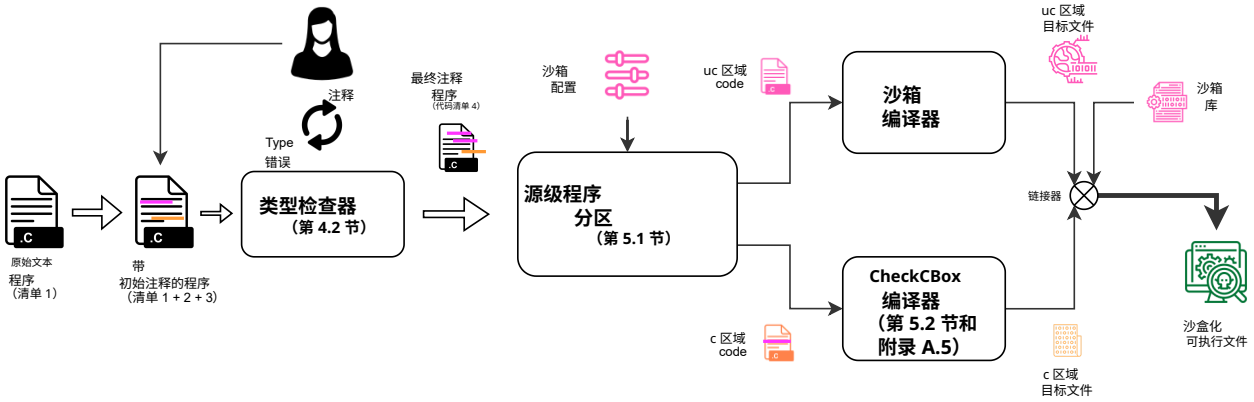


图 1：CheckedCBox 各阶段之间交互的概述。

通过 read_msg 读取数据，并根据第一个字节调用process_req1 或 process_req2 来处理请求。

漏洞。由于将 用作数组 process_req1 中的索引，且未进行任何合理性检查，因此存在任意内存写入漏洞（由 表示）。变量 i可以取任意整数值，因为它是从 msg中解析得到的，而其内容是从套接字读取的，如 read_msg 所示，标记为 。

目标。开发者的目标是对清单 1 中的代码进行分区，使空间内存漏洞不会影响程序的其他部分。理想情况下，开发者可以将全部代码转换为 Checked C，从而实现全面的空间内存安全。然而，转换为 Checked C 可能需要付出相当大的工作量。例如，Checked C 不直接支持 void* 指针。因此，开发者需要将使用 void* 指针的函数转换为泛型版本，这可能会很繁琐。为了解决这一问题，开发者可以尽力进行转换，只标注少量指针，例如，使用自动转换工具3C 进行转换，该工具会像清单 2 所示那样标注少量指针。然而，如第 2.2 节所示，u 区域中的代码也会影响已检查指针的安全性。

3.2 CheckedCBox 注释

为克服上述困难，开发者使用未检查指针标记高风险函数，即，read_msg 和 process_req1作为受污染函数，如代码清单 3 所示，并通过使用CheckedCBox，生成带有已检查类型和污染类型的部分注释程序，即，将列表 2 和 3 叠加到清单 1 之上。根据我们的类型规则（第 4.2 节），最初的污染函数可能需要将其他指针标记为受污染。开发者使用我们的类型检查器来识别所需的其他注释（），并按照代码清单 4 添加这些注释。如代码清单 5 所示，最终得到的良类型程序会与目标沙箱的某些配置参数一起传递给我们的源级程序划分器。

3.3 分区

我们的分区器会将提供的程序拆分为两组源文件，并进行必要的源代码修改，以便它们能够通信

变量： x 整数： $n ::= \mathbb{Z}$ 上下文模式： $m ::=$ c | u指针模式： $\xi ::=$ m | tBound： $b ::=$ $n \mid x + n\beta$ ::= (b, b) 字类型： $\tau ::=$ int | $\text{ptr}^{\xi}\ \omega$ 类型标志： $\kappa ::=$ $nt \mid \cdot$ 类型： $\omega ::=$ $\tau \mid [\beta\ \tau]_{\kappa} \mid \forall\, x.\, \tau \to \tau$ 表达式： $e ::=$ $n : \tau \mid x \mid e + e \mid (\tau) e \mid \tau e \mid$ $\text{ret}(x, n : \tau, e) \mid \text{strlen}(x) \mid \text{malloc}(\xi, \omega) \mid *e \mid *e = e \mid$ 令 $x = e$ 于 $e \mid$ 若 (e) e 否则 $e \mid e(e) \mid$ 未检查(x) $\{e\}$ | 已检查(x) $\{e\}$

图2：CoreChkCBox 语法

包含沙箱化代码。这些源文件集合使用相应的编译器进行编译，以获得相应的目标文件。c 区域目标文件包含用于实施CheckedCBox 保证的必要运行时检查。u 区域目标文件则根据相应的沙箱机制生成。最后，这两个目标文件与必要的沙箱库一同链接，生成最终的可执行文件，从而使所有污染函数都在沙箱（u 区域）中执行，而其余函数则作为标准代码（c 区域）执行。

4 CHECKEDCBOX 形式化

本节介绍 CheckedCBox 的形式化核心模型，称为 CoreChkCBox。我们将介绍其语法、语义、类型系统，以及 CoreChkCBox 的元理论，包括类型健全性、非暴露性和非崩溃定理。

4.1 语法

图2展示了 CoreChkCBox 的语法。

类型语法。从高层来看，我们将类型分为字长值、多字值或函数类型。字长值可以是整数或指针。每种指针类型（$\text{ptr}^{\xi}\ \omega$）都包含一个指针模式注释（ξ，上下文模式与指针模式的区别将在下文简要介绍），它可以是已检查（c）、受污染

(t)、或未检查（u），以及一个类型（ω），表示其所指向的有效值类型。多字值类型（$[\beta\ \tau]_\kappa$）涵盖数组和以空字符结尾的数组，由数组中元素的类型（τ）、数组边界（β）（由数组大小的上界和下界构成（(b_l, b_h)））以及数组标志（κ）构成。边界 b 仅限于整数字面量 n 和表达式 $x+n$。数组指针是否以空字符结尾，由注释 κ 决定；对于以空字符结尾的数组，该注释为 nt，否则为 $\cdot$（书写类型时省略 $\cdot$）。下文展示了 CoreChkCBox 中数组和以空字符结尾的数组的表示示例：

$$\texttt{_t_Array_ptr<}\tau\texttt{>} : \texttt{count}(n) \quad \Leftrightarrow \quad \text{ptr}^{t}\ [(0,n)\ \tau]$$
$$\texttt{_NT_Array_ptr<}\tau\texttt{>} : \texttt{count}(n) \quad \Leftrightarrow \quad \text{ptr}^{c}\ [(0,n)\ \tau]_{nt}$$

为简洁起见，我们用 $\text{ptr}^{c}\ [b\ \tau]$ 表示 $\text{ptr}^{c}\ [(0,\ b)\ \tau]$，因此，上述示例可分别改写为 $\text{ptr}^{c}\ [n\ \tau]$ 和 $\text{ptr}^{c}\ [n\ \tau]_{nt}$。

禁止不安全类型。这些类型的良构性见附录 A.2。此规则可防止构造不安全类型。考虑类型 _t_Array_ptr<_Ptr<int>>，它表示由已检查指针构成的污点数组。在 CoreChkCBox 中，此类型并非良构，因为使用受污染（t）数组时，它可能会在 u 区域中暴露已检查指针的地址。不过，我们可以有一个元素为污点指针的已检查数组，例如，_Array_ptr<_t_Ptr<int>> 是有效类型。函数类型使用依赖函数声明表示，即 $\forall\ x.\ \tau \to \tau$，其中 x 表示一个 int 类型变量列表，这些变量用于绑定 τ 和 τ 中出现的类型变量。下面展示了一个函数指针类型示例：

$$\texttt{_t_ptr<(int)(_t_NT_Array_ptr<}\tau_1\texttt{> : count}(n)\texttt{,}$$
$$\texttt{_t_NT_Array_ptr<}\tau_2\texttt{>: count}(n)\texttt{, int}\ n\texttt{)>}$$
$$\Leftrightarrow \text{指针}^{t}\ (\forall\ n.\ \text{整数} \times \text{指针}^{t}\ [(0,n)\ \tau_2]_{nt} \times \text{指针}^{t}\ [(0,n)\ \tau_1]_{nt} \to \text{整数})$$

函数类型还必须满足良构要求（附录 A.2），这些要求禁止在污点指针中嵌套已检查指针。此外，这些要求还确保 τ 和 τ_ 中的所有变量都由 x_ 限定。

表达式。CoreChkCBox 表达式包括常见表达式，例如加法（$e_1 + e_2$）、指针解引用（* e）和赋值（* $e_1 = e_2$），以及需要特殊处理的表达式，例如静态转换（$(\tau)e$）、动态转换（τe）3、strlen 操作（strlen(x)）、内存分配（malloc(ξ, ω)）、函数调用（$e(e)$）、未检查块（unchecked(x){e}）_ 和已检查块（checked(x){e}）。例如，动态边界转换 dyn_bounds_cast<_Array_ptr<τ>>(e,count(n)) 在 CoreChkCBox 中形式化为 $\text{ptr}^{c}\ [n\ \tau]e$。

我们用带类型的整数字面量 $n\ \tau$（即 int 或 $\text{ptr}^{\xi}\ \omega$）表示，以便将固定地址用作指针。例如，$0{:}\text{ptr}^{\xi}\ \omega$（对于任意 ξ 和 ω）表示空指针。堆分配 malloc(ξ, ω) 包含一个模式标志 ξ，用于在不同区域分配内存：在 c 区域中分配 c 模式指针，或在 u 区域中分配 u 模式和 t 模式指针。我们不允许 ω 为函数类型_（_ $\forall\ x.\ \tau \to \tau$）。已检查表达式和未检查表达式用于划定代码区域。为保证非暴露性质，我们扩展 Checked C 语法，加入 checked(x){e} 和 unchecked(x){e}

[3]在编译时假定，并在运行时验证，参见 [21]

$$\begin{array}{lll} r & ::= & e \mid \texttt{null} \mid \texttt{bounds} \\ E & ::= & \square \mid E + e \mid n{:}\tau + E \mid (\tau)E \mid \langle\tau\rangle E \mid *\,E \mid *\,E = e \\ & & \mid *\,n{:}\tau = E \mid \texttt{let}\ x = E\ \texttt{in}\ e \mid \texttt{ret}(x, n{:}\tau, E) \mid \texttt{if}\ (E)\ e\ \texttt{else}\ e \\ & & \mid E(\overline{e}) \mid n{:}\tau(\overline{E}) \mid \texttt{unchecked}(\overline{x})\{E\} \mid \texttt{checked}(\overline{x})\{E\} \end{array}$$

$$\frac{m = mode(E) \qquad e = E[e'] \qquad (\varphi, \mathcal{H}, e') \longrightarrow (\varphi', \mathcal{H}', e'')}{(\varphi, \mathcal{H}, e) \longrightarrow_m (\varphi', \mathcal{H}', E[e''])}$$

$$\frac{\texttt{u} = mode(E) \qquad e = E[e'] \qquad \tau = type(e')}{(\varphi, \mathcal{H}, e) \longrightarrow_{\texttt{u}} (\varphi, \mathcal{H}, E[0{:}\tau])}$$

图3：CoreChkCBox 语义：求值

代码块，其中¯x 表示允许在代码块外部和内部的指令之间通信的所有变量 e，且其中不能包含已检查指针。ret 由语义在求值 let 绑定时引入；下文将很快对此进行解释。

CoreChkCBox 旨在做到足够简单易用，同时又足够强大，能够表达真实的 CheckedCBox 惯用模式。例如，循环可以编码为递归函数调用。出于篇幅考虑，图2未包含 结构体，但如 [21]所示，系统支持它们。Checked C 中的 C 风格 联合体不具备安全类型检查，因此我们将其省略。尽管 CoreChkCBox 的基础语法与 Checked C 模型 [21], 相似，但为支持污染类型（t 见 ξ）、特殊的堆处理（即malloc(ξ, ω)），以及明确指定指针属于已检查和未检查区域，进行了大量增强。

4.2 类型检查与语义

CoreChkCBox 类型系统是一种流敏感的渐进式类型系统，会在类型检查阶段生成额外的动态检查，并在语义求值阶段执行这些检查。我们的类型检查器限制污染指针和受检指针类型的使用，以确保污染指针不会影响受检类型，同时强制遵循 Checked C 类型规则 [21]。

如图4（标记为 T-X）部分所示，每个类型判断的形式为 $\Gamma;\Theta \vdash_m e : \tau$，表示在类型环境 Γ（将变量映射到其类型）和谓词环境 Θ（将整型变量映射到布尔谓词）下，如果在上下文模式 m中求值，表达式 e 将具有类型 τ，这表明代码位于 m 区域。CoreChkCBox 的操作语义定义为一个小步转换关系，其判断形式为 $(\varphi, \text{H}\ \ e) \longrightarrow_m (\varphi', \text{H}'\ \ r)$，如图3所示。此处，$\varphi$ 是一个将变量映射到值的栈 $n{:}\tau$，而 H 是一个堆，分为两部分（c 和 u 堆区域），每部分都将地址（整数字面量）映射到值 $n{:}\tau$。完整的类型规则以及对 (NT) 数组的特殊处理见附录 A.3 和 A.4。

我们用 $\mathcal{H}(m,\ n)$ 获取堆中位于 n 处的值，并用 $\mathcal{H}(m)[n \mapsto n\prime{:}\tau]$ 将位置 n 更新为 m 堆中的值 $n\prime{:}\tau$。堆绑定可以改变，而栈绑定是不可变的——变量 x 一旦在 φ 中绑定到 $n{:}\ \tau$，该绑定就不会再更新。如前所述，当 τ 是指针类型时，值 $0{:}\tau$ 表示空指针。相应地，$\mathcal{H}(m,\ 0)$ 应始终未定义。该关系通过步进得到一个结果 r，它

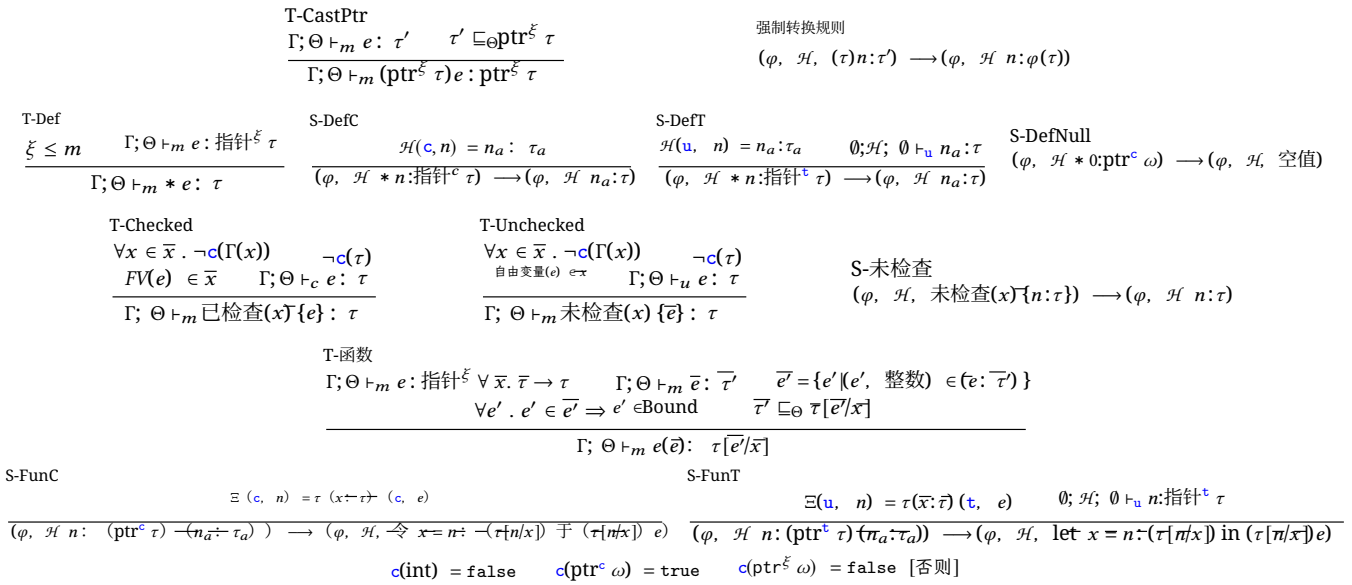


图4：选取的类型检查规则（T-X）和语义规则（S-X）。第一行为强制类型转换操作，第二行为指针解引用，第三行为已检查/未检查代码块，其余行为函数调用。

可以是表达式、空值或越界失败，分别表示归约后立即得到的表达式、空指针解引用或越界访问。这些失败是良好 的结果；停滞状态（无法转换为结果 r的非值表达式）则表示未定义行为。

主要操作语义判断 eval- uation 的规则列于图 3 底部。第一条规则接收一个表达式 e，将其分解为一个求值上下文 E 和一个子表达式 e'（使得将 E 中的空位 $\square$ 替换为 e'后得到 e），然后根据计算关系（$\varphi, \mathrm{H}\ \ e'$） $\longrightarrow$（$\varphi, \mathrm{H}\ \ e''$）对 e'求值。该计算关系的规则与类型规则一同列于图 4（标记为 S-X），稍后将对此进行讨论。图 3 中的模式函数根据上下文 E，确定表达式 e' 所在的上下文模式，即区域。在代码清单 3 中，函数调用 handle_request 位于 u 区域，因为它处在未检查函数 server_loop 内。第二条规则描述了 u 区域中可能出现崩溃行为时的异常处理。u 区域中的操作可能会非确定性地崩溃，而 CheckedCBox 沙箱机制会将程序恢复到安全点（$0{:}\tau$），并继续使用现有的程序状态。求值上下文 E 定义了标准的从左到右的求值顺序。

模式、静态强制转换与子类型关系。在 CoreChkCBox 中，类型规则中出现的上下文模式 m 决定了允许进行指针解引用和值赋值的代码区域，这也取决于指针模式。我们定义一个三点格 $\xi_1 \le \xi_2$ [4] 来描述这种许可，其中 $\mathtt{t} \le \xi$和 $m \le m_o$。这意味着，t 指针可以在任意区域中进行解引用和值赋值，而c 和 u 指针只能分别在 c 和 u 区域中执行此类操作。

CoreChkCBox 还提供静态强制转换操作。如图 4 中的规则 T-CastPtr 所述，一个指针类型表达式

[4]在类型规则中，格通常用作 $\xi \le m$ as m 表示上下文模式。

类型为 $\mathtt{ptr}^{\xi_1}\ \tau_1$ 可以被强制转换为另一种指针类型 $\mathtt{ptr}^{\xi_2}\ \tau_2$，条件是 $\mathtt{ptr}^{\xi_1}\ \tau_1$ 是 $\sqsubseteq_\Theta$到 $\mathtt{ptr}^{\xi_2}\ \tau_2$的子类型关系，即$\mathtt{ptr}^{\xi_1}\ \tau_1 \sqsubseteq \mathtt{ptr}^{\xi_2}\ \tau_2$。在 CoreChkCBox中，除了可以将 t 模式的指针强制转换为 u模式的指针外，所有子类型关系都存在于模式相同的两种类型之间，这意味着 ξ_1 和 ξ_2 在上文中的含义基本相同，而上面的模式格（$\le$）与子类型关系无关。

```
//_Ptr<int> x; _t_Ptr<int> y; int *z;
z = (int *)y; // This is okay.
x = (_Ptr<int>)y; // Not allowed.
```

在上面的示例中，t 模式指针可以转换为 u 模式，但不允许将 t 模式转换为 c 模式。完整的子类型关系已在附录 A.2 中描述。请注意，let 语句在CoreChkCBox中是不可变的，因此以下代码无法成立，因为变量x 和 y 在 CoreChkCBox中必须具有相同的类型。

```
//_Ptr<int> x;  _t_Ptr<int> y;
x = y; // Not allowed.
```

指针解引用。指针解引用的类型规则和语义规则（图4中的 T-Def、S-DefC、S-DefT、S-DefNull）体现了 CoreChkCBox 的一项关键特性：类型检查器会指导插入在求值阶段执行的动态检查。类型规则（T-Def）确保指针在正确的区域中以正确的模式使用（$\xi \le m$）。借助编译器插入的动态检查，规则 S-DefNull 确保使用空指针时，CoreChkCBox 能捕获运行时错误。数组类型和指针赋值的类型规则和语义规则见附录 A.3 和 A.4。

规则 S-DefC 和 S-DefT 分别用于 c 和 t 模式下的指针解引用。除了 空值 检查外，在 c 模式下解引用指针时，对受污染的（t）指针进行任何动态堆访问都需要进行 验证 $(\emptyset;\ \mathcal{H};\ \emptyset \vdash_u n_a :\ \tau)$，以确认指针值 n_a在 $\mathcal{H}(m, n_a)$中定义良好且具有正确的类型 τ。

未检查块与已检查块。已检查块或未检查块的执行表示从 c 区域切换到 u 区域，或反向切换；其类型和语义规则见图4。在这种上下文切换中，为了保证已检查（c）指针的不暴露性质，不允许已检查指针跨越不同区域，这由谓词 $\forall x \in \overline{x}.\ \neg c(\Gamma(x))$ 和 $\neg c(\tau)$ 保证，同时还要检查块内容 e 中的所有自由变量都属于 $\overline{x}$。例如，第 5.1.2 节中的 StringAuth 是一个蹦床函数，不允许将已检查指针作为参数或返回值。在下文_T_中使用该函数时，由于 StringAuth 位于 u 区域，因此不能合法地接受任何已检查指针；否则，我们可能会将已检查指针地址暴露给不安全代码区域。在 Checked-CBox 中，我们实际上允许在 StringAuth 内访问已检查指针，因为蹦床函数的函数体位于 c 区域。更多信息见第 5.1.2 节。

依赖函数指针。规则 T-Fun（图4）给出了依赖函数指针应用的类型判断，其中我们用 $\tau[e'/x]$ 表示将类型 τ 中所有整数有界变量 x 替换为绑定表达式 e' 的结果，并用 $\overline{\tau}[e'/x]$ 表示将该替换提升到 τ 中的每个类型。给定一个函数指针类型为 $(\text{ptr}^{\xi}\ \forall x.\ \tau \to \tau)$ 的表达式 e，以及类型为 τ' 的参数 e，应用的结果类型为 $\tau[e'/x]$；前提是对于 τ' 和 $\tau[e'/x]$ 中的每一对 τ' 和 τ''，$\overline{\tau}'$ 都是 τ'' 的子类型。考虑图 5 中的 process_req2 函数，其参数 msg 的类型取决于 m_1。它的函数指针类型为 $\text{ptr}^{t}\ \forall \text{m_1}.\text{int}, [(0, \text{m_1})\ \text{char}]_{nt} \to \text{int}$。在 handle_request 中，调用 process_req2(buff, r_len) 将变量 m_1 绑定为 r_len 调用返回后，m_1的作用域已结束，因此我们需要在最终返回类型中将其替换为r_len，因为其中可能包含m_1。

S-FunC 和 S-FunT 分别定义了 c 模式和 t 模式函数指针的语义。对函数指针的调用 n 会从全局函数存储 Ξ 中检索位于 n内存位置的函数定义。该存储将函数指针映射到函数数据 $\tau(x{:}\tau)\ (\xi, e)$，其中τ是返回类型，$(x{:}\tau)$ 是变量及其类型的参数列表，ξ决定了函数的模式，而 e 是函数体。与 H 类似，全局函数存储 Ξ 也分为两部分（c 和 u 存储区域），每部分都将地址（整数字面量）映射到上述函数数据。规则 S-FunT 定义了受污染的函数调用版本，其验证过程 $\emptyset; H; \emptyset \vdash_u n : \text{ptr}^{t}\ \tau$确保全局存储中的函数定义良好且类型正确。

4.3 元理论

这里，我们讨论 CoreChkCBox 的主要元理论结果：类型健全性（进展性和保持性）、非暴露性以及不会崩溃。这些证明已在我们的 Coq 模型中完成。类型健全性依赖于 [21] 和附录 A.6 中给出的若干 良构性 。下文的进展定理指出，CoreChkCBox 程序总能执行一步。

定理 1（进展性）。对于任意 *CoreChkCBox* 程序 e、堆 H、栈 φ、类型环境 Γ 和变量谓词集 Θ，若它们均为良构、一致（$\Gamma;\Theta \vdash \varphi$ 且 $H \vdash \varphi$），并且类型正确（$\Gamma;\Theta \vdash_c e : \tau$，其中某个 τ），则以下情况之一成立：

- e是一个值（$n : \tau$）。
- 存在 φ', H', r，使得 $(\varphi, H, e) \longrightarrow_m (\varphi', H', r)$。

关于c 和u 区域，有两种保持性。检查保持性指出，归约步骤会保持正在归约的程序的类型和一致性；而未检查保持性指出，发生在u 区域的任何求值都不会影响c 模式堆。

定理 2（检查保持性）。对于任意 *CoreChkCBox* 程序 e、堆 H、栈 φ、类型环境 Γ 和变量谓词集 Θ，若它们均为良构、一致（$\Gamma;\Theta \vdash \varphi$ 且 $H \vdash \varphi$）并且类型正确（$\Gamma;\Theta \vdash_c e : \tau$，其中某个 τ），且存在 φ'、H'和 e'，使得 $(\varphi, H, e) \longrightarrow_c (\varphi', H', e')$，则 H'与 H 在 c 区域上一致（$H \triangleright H'$），并且存在 Γ'和 τ'，它们是良构的、在 c 区域上一致（$\Gamma';\Theta \vdash \varphi'$ 且 $H' \vdash \varphi'$）并且类型正确（$\Gamma';\Theta \vdash_c e : \tau'$），其中 $\tau' \sqsubseteq_\Theta \tau$。

定理 3（未检查保持性）。对于任意 *CoreChkCBox* 程序 e、堆 H、栈 φ、类型环境 Γ 和变量谓词集 Θ，若它们均为良构且类型正确（$\Gamma;\Theta \vdash_c e : \tau$ 对某个 τ），并且存在 φ'、H'和 e'，使得 $(\varphi, H, e) \longrightarrow_u (\varphi', H', e')$，则 $H'(c) = H(c)$。

利用上述定理，我们首先证明非暴露性定理： u 区域中的代码无法观察到有效的已检查（c）指针地址。

定理 4（非暴露性）。对于任意 *CoreChkCBox* 程序e、堆 H、栈 φ、类型环境 Γ 和变量谓词集 Θ，若它们均为良构且类型正确（$\Gamma;\Theta \vdash_c e : \tau$ 对某个τ），并且存在 φ'、H'和 e'，使得 $(\varphi, H, e) \longrightarrow_u (\varphi', H', e')$，且 $e = E[\alpha(x)]$，并且 $mode(E) = u$，其中 $\alpha(x)$ 是包含变量 x的某个表达式（既未检查也未检查），因此它不是已检查指针。

现在我们陈述主要结果：不崩溃，这表明良类型程序永远不会陷入停滞状态（表达式 e 是一个非值，无法继续求值[5]）。

定理 5（不崩溃）。对于任意 *CoreChkCBox* 程序e、堆 H、栈 φ、类型环境 Γ和变量谓词集Θ，若它们良构且一致（$\Gamma;\ \Theta \vdash \varphi$ 且 $H \vdash \varphi$），并且m_ℓ类型良好（$\varphi;\ \Theta \vdash_c e : \varphi\ \tau$对于某个$(m_1 \tau)\varphi_1$，且存在$\varphi_{m_2} H_i$、$e_i$和 $\ldots \longrightarrow_{m_k} \varphi_k\ _k\ r\ \ r$ 对应于 $i \in [1, k]$，使得$(_, H)\ (_, H_,)\ (_, H_,)$，那么它永远不会陷入停滞状态。

5 实现

如第 3 节所述，我们的 CheckedCBox 实现包含两个主要组件，即源级程序划分器（CheckMate）和编译器。给定一个类型良好的CheckedCBox 程序和沙箱配置，CheckMate 会执行源到源转换，将程序划分为两部分（即，两组源文件）——分别对应于 c 和 u 区域的已检查源文件和污染源文件。我们的编译器会转换已检查源文件，并添加必要的动态指令。污染源文件需要使用目标沙箱编译器进行编译。最后，这两组目标文件会与任意沙箱库一起链接，生成能够强制实施我们安全保证的最终可执行文件。

[5]请注意，边界和空值并不是停滞表达式——它们表示程序因运行时检查失败而终止。尝试访问$\mathcal{H}n$，但$\mathcal{H}$在 n处未定义的程序会陷入停滞，并违反空间安全。

我们为每个沙箱一次性创建沙箱库。该库抽象了沙箱特定的细节，并提供统一的接口（头文件），供 c 区域使用。例如，_SBX_() 会获取一个指向目标沙箱的不透明指针。我们将以 WebAssembly（WASM [4]）作为目标沙箱来介绍我们的实现。我们还会对该实现进行形式化，并在附录 A.5 中给出一个模拟定理。

5.1 CheckMate

CheckMate 主要以 C 语言实现，是一款 Clang 前端工具（3K 源代码行数）。不过，我们使用一个小型 OCAML 程序（680 源代码行数）来移除注释，并使代码能够由沙箱编译器编译（第 5.1.3 节）。

5.1.1 其他函数限定符。除了 _Tainted限定符（用于标记函数所属的u 区域）之外，我们还提供了几个其他限定符，使开发者能够提供更多信息并简化分区过程。具体来说，我们还提供三个限定符：_Callback、_Mirror和_TLIB。

_Callback：开发者应使用此限定符标记回调函数，也就是位于 c 区域、且可从污染区域调用的函数。CheckMate 会插入适当的沙箱依赖机制，以实现这一功能（5.1.2）。

_Mirror：此限定符允许将相应函数同时复制到c 区域和u 区域，从而能够处理某些在两个区域中都会调用的简单实用函数。例如，在我们对 Parsons 的评估中，append_string 在我们的 Parsons 评估中，_wasm 的调用方来自这两个区域。

```
_Mirror int append_string(_TPtr<char> buf,
const char* appendStr : itype(_Nt_array_ptr<const char>),
_TPtr<char> buf_start, size_t buf_len) {
/* Qualifier Rules:
1.) No access to global data NOT marked "const"
2.) Callees must be _Tainted or _Mirror         */.....}
```

限定符 append_字符串与 _Mirror 会在两个区域中复制该函数，因此允许在append_字符串 中将参数传递给 appendStr ，使其在 u 和 c 区域中分别作为未检查指针或已检查指针。由此避免了过度污点标记带来的复杂性，因为 appendStr 在 c 区域中无需标记为受污染，也无需通过回调来访问 append_字符串 ，该字符串位于 u 区域中。“_Mirror”会在编译时强制执行控制流和数据流语义规则，确保解析镜像函数在 u 区域中编译所需的所有变量和函数调用依赖项。*_TLIB*：此限定符放宽了对库函数的类型检查规则

，使开发者能够在C 语言区域中自由使用该函数。

```
// First, manually check the memory is in tainted region.
// if yes, then call strncpy.
if (!is_mem_in_range(t_str, t_str + n, SBX_LOW(), SBX_HIGH()))
  handle_violation();
// our type checker ignores this because
// the _TLIB annotation below.
strncat(dst, t_str, n);
```

```
- extern char *strncat (char *__restrict __dest,
+ _TLIB extern char *strncat (char *__restrict __dest,
const char *__restrict __src, size_t __n); // In the header file
}
```

传递污点指针t_str 给未限定的 strncat 是不允许的，除非具备额外的 u 区域实现

用于 strncat。如果用户确认 t_str 具有适用于 strncat的正确缓冲区大小，则她可以使用 strncat 标注为 _TLIB，这样一来，t_str就可以作为已检查指针参数处理；此类注释会放宽对其调用中所有参数的类型检查。值得注意的是，CheckedCBox 不会强制实施任何语义来确保 _TLIB 在 c 区域中实现的函数不会修改内存；因此，使用 _TLIB 要求用户意识到内存地址泄漏的风险。

5.1.2 生成C 语言区域源代码分区。我们将所有未受污染的函数复制到C 语言区域源文件中，并进行以下修改，以便与U区域（即沙箱化代码）交互。我们为每个沙箱创建一次库。该库封装了特定于沙箱的细节，并提供统一的接口（头文件），供C 语言区域使用。例如，_SBX_() 会获取一个指向目标沙箱的不透明指针。

处理对受污染函数的调用：在 c 区域中，我们还需要修改对受污染函数的调用，因为这些函数在沙箱（独立地址空间）内执行，因此无法像普通函数一样调用。然而，修改每个受污染函数的调用点既繁琐，也需要精确的指针分析 [31]，以处理通过函数指针进行的间接调用。

我们通过间接寻址来处理这一问题：我们不修改调用点，而是修改受污染函数的函数体，使其调用沙箱中的相应函数。例如，我们修改 process_req1(清单 5) 中c 区域内受污染函数的函数体，如下所示：

```
int process_req1(char *msg, size_t m_l) _Tainted {
- int rc = -1, i;- if (m_l > MIN_SIZE) {- ...
+ return w2c_process_req1(msg, m_l);}
```

这样可以确保，所有对受污染函数 process_req1 的调用（包括通过函数指针进行的调用）都会被重定向到沙箱。

处理_回调限定符：如第 5 节所述 .1.

1，带有这些限定符的函数可以从 u 区域调用。请看下面的StringAuth 函数，它通过访问已检查数据来检查所提供的用户输入 usertoken 是否经过身份验证。由于需要从 u 区域调用该函数，因此将其注释为 _回调。

```
_Callback _TPtr<char> StringAuth(
    _T_Array_Ptr<const char> usertoken : count(len),
    size_t len) {
...
// Checks whether usertoken is authenticated
/*
 These functions will be restricted to only accept
 tainted parameters.
*/
...
}
```

这些回调函数只能使用污染参数，因为它们将从受污染区域调用。

对于每个此类函数，我们都会创建一个对应的蹦床函数，作为该回调函数的入口点，如下所示：

```
+ unsigned int _T_StringAuth(void* sandbox,
+               unsigned int arg_1, unsigned int arg_2) {
+   // Perform necessary Type-conversion of arguments.

+   // uname <- conver arg_1+   // len <- arg_2
+   ret = StringAuth(uname, len);
```

```
+   // ret_val <- ret
+   return ret_val;
+ }
```

蹦床函数处理来自沙箱的调用（因此还需要额外的参数沙箱），执行必要的指针参数转换，最终调用回调。

我们还添加了用于向沙箱注册该蹦床函数的代码。WASM 沙箱的注册函数如下所示：

```
+ void registerCallback_StringAuth(void){
+ //callback function signature {ret <- int, arg_1 <- int, arg_2 <- long}
+ int ret_param_types[] = {0, 0, 1};
+ // 2 <- arg count, 1 <- ret count
+ __StringAuth__C_ = _SBXREG_((void*)_T_StringAuth,2,1, ret_param_types);
+ }
```

该注册函数为蹦床函数创建一个不透明句柄，并允许 u 区域使用相应的句柄调用蹦床函数。

最后，我们修改污染函数的函数体，使其包含对沙箱中污染函数实现的间接调用。不过，我们不再直接从参数列表中传递回调函数指针，而是传递生成的蹦床句柄 `__StringAuth__C_`，如下所示：

```
_Tainted _TPtr<char> StringProc(_TPtr<_TPtr<const char>> user_input,
_TPtr<_TPtr<char>(_TPtr<const char> input, size_t len)>StringAuth) {
-  ...
-  //complex Function Body
- return StringAuth(one_past_start, string_len);
+ return w2c_StringProc(_SBX_(), (unsigned int)string, __StringAuth__C_);

}
```

5.1.3 生成 u 区域源分区。 我们首先复制所有标记为_受污染 和 _Mirror的函数和变量，以创建 u 区域分区。接下来，我们将所有污染类型替换为相应的 C 类型，从而将该分区转换为标准 C。

例如：_T_数组_ptr<char>将替换为char *

在u_区域中，清单 5 里的readmsg 函数签名将修改如下：

```
size_t read_msg(int sock_fd, char* msg,
-                            size_t sz) _Tainted {
+                            size_t sz) {
```

5.2 编译器

CheckedCBox 编译器包含类型检查器（第 4.2 节）以及处理污点指针（即来自沙箱）并执行运行时检查所需的插桩。我们在现有 Checked C 编译器之上，将编译器实现为一个库（1.8K 源代码行数）。我们的类型检查器会验证约束（图4），并将违规情况显示为错误，使开发者能够添加注释。例如，对于清单 5 中的代码，我们的类型检查器会发出错误，指出“*buffer* buff 已受污染，但它分配在c区域中”。因此，开发者必须进行以下更改，才能在沙箱中分配*buff*：

```
- char buff[MAX_MSG_SIZE] _Tainted;
+ _T_Array_ptr<char> buff: count(MAX_MSG_SIZE)
+    = sbx_allocate(MAX_MSG_SIZE);
...
+ sbx_free(buff)
return rc;
```

对于所有受污染指针访问，我们的插桩都会添加一项检查（在运行时执行），以确认地址是否位于沙箱的有效范围内。下面展示了该检查的一个示例：

```
_TPtr<int> t = ...
+ if (!is_in_range(t, SBX_LOW(), SBX_HIGH()))
+  handle_violation();
*t = 1;
```

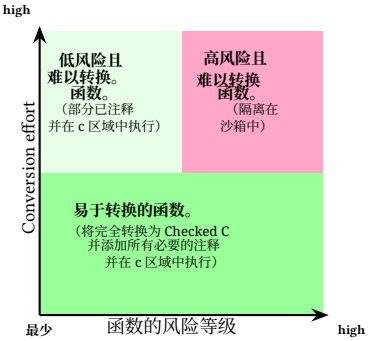


图 5：使用 CheckedCBox 进行灵活分区。

为便于说明，我们使用 C 语言展示所添加的检查。不过，我们的插桩作用于 LLVM IR。

区域 u 的源文件将使用沙箱编译器进行编译，即 WASI-SDK（ [44]）。鉴于 WASM 沙箱使用 32 位地址（而 c 区域使用 64 位地址），我们的插桩还会添加必要的指针转换，以便透明地访问复合数据类型，即 struct中带有指针的结构体。

5.3 保障平衡与开销

将所有未经过检查的函数标记为受污染，可能会导致大量向沙箱的切换，从而带来不必要的开销。例如，在我们的示例中，清单 5 中将 read_msg 函数标记为受污染，会导致系统每次读取消息时都要切入和切出沙箱，这可能效率低下。

然而，正如第 2.1 节所述，已检查指针可以与普通指针并用，因此 C 语言 区域 也可以 包含普通（未检查）指针。这使程序分区方式更加灵活，例如，只将少数高风险的未检查函数标记为受污染，而不是将所有这类函数都标记为受污染。对于清单 5 中的示例，我们可以只将函数 process_req1标记为受污染，从而仅在收到相应的（高风险）请求时才进入沙箱。

通过这种方式，该系统可以用作一种通用的程序分区技术，在开销与安全保障之间取得平衡。图 5 展示了这种替代用法的一般指导原则。出于安全考虑，最好为所有函数标注受检类型。然而，如果难以转换所有函数，那么仅将某些漏洞概率较高的高风险函数标记为受污染可能更为实际，同时仍能兼顾安全保障与开销。为了在实现这种通用用法的同时不违反CheckedCBox的安全保证，我们增加了一项限制：不允许将污点类型转换为通用类型。具体来说，我们禁止在C 语言 区域中进行以下类型转换：

```
//_t_Ptr<int> y; int *z; ∅
z = (int *)y; // Not allowed.
```

6 评估

我们从以下三个方面评估 CheckedCBox：

- **转换工作量：**需要开发者投入多少工作量，为程序的部分代码添加注解，使其能够在 CheckedCBox 中运行?

- **性能开销：** 使用 CheckedCBox 会带来多少性能开销（包括运行时开销和内存开销）？
- **安全影响：** CheckedCBox 提供的隔离在防止安全漏洞方面有多有效？

6.1 数据集

面向网络的程序（如服务器）会直接与外部输入交互，通常需要处理复杂数据类型，因此更容易受到安全问题的影响。我们主要关注网络服务器，因为它们最能受益于我们的分区方法。我们使用 WebAssembly (WASM) 作为目标沙箱。因此，我们还要求选定的程序能够在 WASM 沙箱中编译。我们选取了能够以较少工作量通过 WASM 编译器编译的网络服务器。此外，我们还选取了 Checked C 团队 [30], 建议的几个独立程序，这些程序适合用于评估对 Checked C 的修改。表 1 展示了纳入评估数据集的程序。

ID	程序	描述	Size (源代码行数)
1	ProFTPD	高性能 FTP 服务器	556 K
2	MicroHTTPD	简易 HTTPD 服务器	122 K
3	UFTPD	基于用户数据报协议的 FTP 服务器	3 K
4	LibPNG（png2pnm 和 pnm2png）	在 PNG 和 PNM 之间进行转换的程序	76 K
5	TinyBigNum	多精度整数实现	1.6 K
6	Parsons	JSON 解析库	3.1 K

表 1：评估数据集。

程序	分区方法	指针已注释	代码行，位于 u 区域（沙箱）	Time 已用（小时）	CVEs 已防止
ProFTPD	*TP*	6	N/A	1	CVE-2010-4221 (✓)
png2pnm pnm2png	*TP*	248	N/A	8	CVE-2018-144550 (✓)
MicroHTTPD	*TP*	139	450	3	N/A
UFTPD	TP_r	146	90	3	CVE-2020-14149 (✓) CVE-2020-5204 (✓)
TinyBigNum	*CTP*	69	30	2	N/A
Parsons	*CTP*	364	800	5	N/A

表 2：转换工作量与安全影响汇总。

6.2 实验设置

所有实验均在一台配备 40 GB 内存、搭载 6 核 Intel i7-10700 H 处理器的机器上进行，运行 Ubuntu 20.04.3 LTS。我们使用 WASM 作为目标沙箱，并采用与近期工作类似的配置 [33]；并使用 Valgrind 的“massif”内存分析器 [40] 测量内存使用量，并将应用程序的峰值堆使用量视为其内存消耗。我们使用 `clock()` API（来自 POSIX 的 `<time.h>` 和 Linux 的 `time` 命令，通过已用时钟周期之差来测量运行时间；每项测量均重复十次，并取平均值作为最终结果。

6.3 转换工作量

CheckedCBox（第 5.3 节）的灵活性使应用程序能够以多种方式进行分区，从而实现不同程度的开销和保障。我们探讨以下三种方式：

Checked C 与受污染分区（*CTP*）：这是最全面的用例，其中 c 区域分区包含完全添加了注释的 Checked C 代码，而u 区域包含一个或多个污染函数。这为c 区域中的代码提供完整的空间安全保障，包括与u 区域的隔离。

仅受污染分区（*TP*）：这是不使用 Checked C 的通用分区用例（第 5.3 节）。这与 *CTP*类似，但c区域代码不使用任何 Checked C 注解。此方式只提供隔离保证，不提供空间安全保障。

仅使用受污染指针（TP_r）：在此用例中，我们仅使用受污染指针，所有代码均位于c 区域。这适用于仅需数据隔离的情形，开发者可能只想将某些数据（例如，用户输入）与程序的其余部分隔离开来。如第 5.2 节所述，通过受污染指针进行的所有访问都会经过动态检查，以确保每次访问都处于沙箱内。此方式可提供部分空间安全：通过确保任何涉及受污染指针的空间违规都不会影响 c 区域。

6.3.1 转换方法。我们使用上述三种方法之一对数据集中的每个程序进行了分区。

我们的目标是将高风险输入处理例程与应用程序代码隔离开来。我们手动分析每个应用程序的源代码，以确定哪些函数处理输入数据以及如何处理。我们还会查看此前报告的漏洞，以识别高风险输入处理例程。我们根据输入数据的处理方式，选择上述方法之一进行分区。表 2 总结了我们的数据集分区情况。对于 `ProFTPD`，TP_r采用该方法，并将输入数据标记为受污染。因此，根据类型规则，还需要将另外五个指针标记为受污染。这总共需要 6 条指针注释。按照u 区域的用法，TP_r该方法中没有代码，只使用受污染的指针。我们对 `LibPNG`(`png2pnm` 和 `pnm2png`) 也采用相同的方法。不过，由于LibPNG的内部结构复杂，这种情况下我们必须注释更多指针（248 个）。

对于 `MicroHTTPD` 和 `UFTPD`，*TP*采用该方法，并将所有直接处理输入的方法标记为受污染，因此将其移至沙箱，同时将几个中间指针标记为受污染。对于 `TinyBignum` 和 `Parsons`，我们遵循 *CTP*并将所有输入处理例程标记为受污染，然后将其置于沙箱中。其余非沙箱化代码均使用 Checked C 类型进行完整注释，并置于 C 语言区域中。

我们通过使用相应的测试套件进行验证，确保分区程序保留了预期功能。

6.3.2 转换工作量。倒数第二列显示了对应用程序进行分区所需的小时数。平均而言，每次分区需要 ~ 3.5 小时。不过，具体耗时取决于应用程序的复杂程度和指针的使用情况。尽管绝对耗时较长，但对每个应用程序而言，分区工作只需进行一次。我们首先注释函数，然后迭代修复类型检查器报错。大部分时间（80%）都花在运行类型检查器上。类型检查器遇到第一个错误后便会停止，且不会提供有关其他需要修复的变量的信息。例如，在以下代码中：

```
_TPtr<int> y = ...;

int *z;int *x;∅
 z = y;x = z;
```

类型检查器只会针对第一个赋值显示错误。然而，要正确修复这个错误，我们需要同时注释 x 和 y。如果N需要注释指针，那么在最坏情况下，我们可能需要运行类型检查器 N次，每次额外注释一个

指针。我们计划在未来的工作中通过实现自动转换流程来解决这个问题。

6.4 性能开销

近期研究 [15] 表明，作为 WASM 沙箱一部分执行的代码会产生显著的运行时开销，即 ~200%。为了更好地了解我们的运行时开销，我们首先对 CheckedCBox 中与沙箱相关的其他操作进行微基准测试。

6.4.1 微基准测试。图 6 展示了我们的微基准测试 g
结果。我们对以下操作进行了测量：
WASM 沙箱中的内存访问（SBX_m）：沙箱中的所有内存访问都需要由沙箱运行时进行额外验证，因此会产生运行时开销。我们在循环中执行 10 万次内存访问（读和写），测量沙箱内部的执行时间，并与作为常规程序执行时的时间进行比较。结果（图 6）表明，与普通程序相比，WASM 沙箱中的内存访问会产生 156.6% 的开销。这与近期研究 [15] 的观察结果一致。

沙箱往返（SBX_{RT}）：我们测量在c 区域与沙箱之间往返所需的时间（u区域），并将其与常规函数调用和返回进行比较。我们创建了如下空操作函数：

```
void noop() { return; }
```

我们将此空操作函数放入沙箱中，并测量调用它并从中返回所需的时间：

```
s = clock(); sandbox_noop(); e = clock();
```

我们将其与空操作函数位于c 区域时的常规调用耗时进行比较。

如图 6 所示，我们产生了 ~ 400%的开销。这也与先前工作的性能报告一致33] [19, 。

这是意料之中的，因为进出沙箱需要进行上下文切换，其开销高于常规函数调用（即call 和ret 指令）。

在c 区域中的污点指针访问（TP_c）：如第 5.2 节所述，我们需要执行指针转换，将沙箱中特定表示形式的污点指针转换为原始地址。此外，我们的插桩还会在运行时检查所有污点指针是否都位于沙箱地址范围内，然后再访问它们。我们通过比较污点指针访问与常规指针访问的耗时，来测量这部分额外开销。

如图 6 所示，在C 语言区域中访问污点指针会产生 34% 的开销，这是由于额外的验证检查所致；这些检查要求每次内存访问都执行两条比较指令。

6.4.2 数据集上的开销。图 7 中第一组柱状图展示了分区程序的开销。运行时开销与在沙箱中花费的执行时间以及 C 语言区域与沙箱之间的转换次数成正比，这与微基准测试实验中观察到的沙箱执行开销一致（图 6）。表 2 展示了分区程序中沙箱内的代码行数。对于 ProFTPD，我们只使用了污点指针，沙箱中没有代码，也没有进出沙箱的转换。因此，开销低于 4.2%。对于 UFTPD 和 MicroHTTPD，我们仅将解析消息并返回待执行操作的请求处理程序（规模相对较小）置于沙箱中。服务器只会

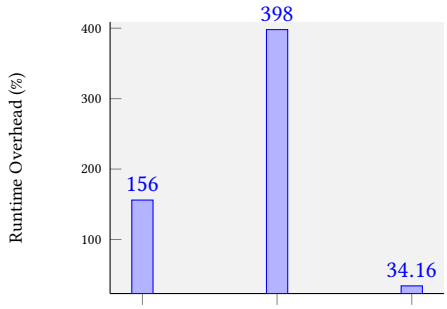


图 6：CheckedCBox 微基准测试。

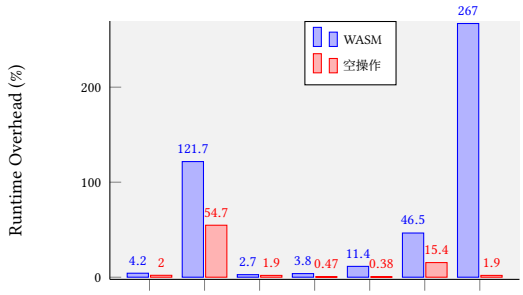


图 7：分区程序的运行时开销。

在特定请求到来时调用这些处理程序，因此与沙箱的转换次数较少。不出所料，这些情况下的开销也较低。对于 TinyBigNum 和 Parsons，由于沙箱区域中的代码相对较多，开销较高。在这两个应用中，我们将频繁使用的解析函数放入沙箱，导致发生大量沙箱转换，其中大多数转换发生在循环中。对于 pnm2png和 png2pnm，情况略有不同：我们将整个 PNG 结构标记为受污染，因此每次访问 PNG 结构体时都会进行动态检查。总之，我们的结果表明，运行时开销在很大程度上取决于沙箱。

仅 CheckedCBox 的开销。为了验证沙箱对整体程序运行时的影响，我们进行了空操作沙箱实验。我们的目标是测量仅由 CheckedCBox 引入的运行时开销。我们通过跳过沙箱化来进行此实验。具体而言，我们将完全带注释的程序作为常规应用程序运行，不进行任何分区。不过，我们使用 CheckedCBox 编译器编译带注释的程序，该编译器会添加相关的运行时检查（第 5.2 节）。我们修改了针对污点指针的插桩，使其检查指针是否有效（而非是否处于沙箱边界内）；这会添加与沙箱化情形相同数量的检查，但比较值会有所不同。

在整个数据集上评估采用 NO-OP 沙箱的 CheckedCBox 时，我们观察到其开销显著低于

图 7 所示的 WASM 沙箱。因此，与其使用的沙箱相比，CheckedCBox 本身造成的开销要低得多。对于 TinyBigNum，开销较高，为 54.7%。我们的分析表明，这是因为我们对主输入缓冲区进行了污点标记，而该缓冲区会在循环中处理。这导致每次循环迭代都需要进行额外检查，从而造成更高的开销。另一个原因是，在当前实现中，我们在所有优化过程完成后才进行插桩，因此插桩没有得到任何优化。我们计划将插桩移至优化过程之前，并利用这些优化过程进一步优化，以降低运行时开销。

6.4.3 内存开销。所有程序都有固定的内存开销（~81 KB），主要用于沙箱，以及用于创建沙箱和其他辅助函数的少量变量。然而，与原版 Checked C 类似，CheckedCBox 本身不会增加任何内存开销，因为编译受污染的指针时不会附带任何元数据。

6.5 安全影响

表 2 的最后一列列出了所有具有污染类型或被隔离在u 区域中的函数所涉及的空间安全漏洞。我们在带注释的程序中重新引入了这些错误，并检查这些错误是否能由相应的崩溃输入或利用程序（如果有）触发。我们还手动验证了这些错误是否能被CheckedCBox触发或阻止。正如预期，所有漏洞都被CheckedCBox阻止了。

符号✅和✅分别表示该漏洞是否由我们的动态插桩检测到，或是否被隔离在沙箱中。这表明CheckedCBox 提供了一种有效机制来防止空间安全漏洞。

7 局限性与未来工作

尽管 CheckedCBox , 具有很高的有效性,

沙箱依赖： CheckedCBox 假设沙箱可用，因此继承了相应沙箱的所有局限性。例如，程序应能够由沙箱编译器编译。此外，如第 6.4.2 节所示，分区应用程序的性能主要取决于沙箱。不过，我们的实现并不依赖于某个特定沙箱，而且很容易扩展到其他沙箱。未来，我们将把实现扩展到其他沙箱。

注释工作量： 目前，所有污点注释都必须手动完成，并且必须满足我们的类型检查器规则（第 4.2 节）。根据沙箱化函数的复杂程度、其参数复杂度以及它对其他函数的依赖关系，这项工作可能会很繁琐。我们计划开发一个自动注释工具：给定初始注释（代码清单 3），该工具将根据我们的类型规则自动添加所有必需的注释（代码清单 4）。

8 相关工作

许多先前的研究都探讨了 C 语言语义的形式化，包括 CompCert [2, 19], Ellison 和 Rosu [10], Kang 等人 [17], 以及 Memarian 等人 [26, 27], , 但它们并未直接关注如何强制实现空间安全。

空间安全 C 形式化。若干先前的研究 [21] 对旨在确保空间安全的 C 语言转换或 C 语言方言进行了形式化。这些研究与 CheckedCBox 的区别将在第 1 节和第 3 节中介绍。Hathhorn 等人[13] 扩展了 Ellison 和 Rosu [10] 的形式化，提出了一种能够检测空间安全违规（以及其他形式的未定义性）的语义；该研究着重于错误查找，而非编译程序以使用这种语义。

CCured [34] 和 Softbound [32] 通过程序转换，为普通 C 实现了空间安全语义。与 CoreChkCBox 类似，这两个系统的操作语义都会为指针添加边界注释。CCured 会将等价于数组指针的指针编译为“胖”指针，而 SoftBound 会将边界元数据编译到单独的哈希表中，因此能够保留二进制兼容性，但检查成本更高。CheckedCBox 利用静态类型信息，无需附加于指针的元数据即可进行边界检查。Cyclone [12, 16] 是一种旨在确保内存安全的 C 语言方言；其指针类型与 CCured 的类似，而其形式化定义 [12] 侧重于确保时间安全。Deputy [6, 46] 是另一种旨在避免使用胖指针的安全 C 方言，其形式化定义 [6] 直接通过编译来定义其语义。以上工作均未涉及程序分区和沙箱机制。

程序分区机制。CheckedCBox 中未检查代码区域与已检查代码区域的分离，代表了一种隔离机制，旨在确保作为未检查区域一部分执行的代码不会破坏已检查区域中的安全保证。这通常称为程序分区 [38], , 该领域已有大量研究 [1, 5, 22, 23, 42] 。这些技术大多是以数据为中心的 [22, 23],, 即由程序数据驱动分区。例如，给定程序中的敏感数据，目标是根据函数能否访问这些敏感数据，将函数划分为两部分或两个分区。这些方法的性能开销主要由编组成本决定，并取决于敏感数据的使用方式。当前最先进方法 [22, 23]的开销高得难以接受，介于 37% 至 163% 之间。RLBox [33], 将类型检查器与沙箱机制结合，以更好地实现程序分区机制。它们允许在不同代码区域之间共享污点指针。不过，RLBox 类型系统主要基于 C 语言++ 模板，因此很难或无法直接应用于 C 程序。此外，该技术没有形式化保证，只能隔离完整的库，而不能隔离任意函数。CheckedCBox 是一个通用的形式化验证系统，能够以更强的保证对任意函数进行分区。

9 结论

我们提出了 CheckedCBox，这是一种结合 Checked C 的类型引导的程序分区机制，用于实现增量式空间内存安全。我们的系统支持将程序中的任意函数分区到沙箱中，从而使这些函数不会影响程序的其余部分。我们形式化了类型系统和语义，并证明了多种可确保空间内存安全的安全性质。我们将 CheckedCBox 实现为一个交互式流水线，使开发者能够以交互方式对应用程序进行分区。我们对七个程序的评估表明，CheckedCBox 能够让应用程序的交互式分区以相对较少的工作量完成。

参考文献

[1] Andrea Bittau、Petr Marchenko、Mark Handley 和 Brad Karp。2008年。Wedge：将应用程序拆分为低权限隔离区。USENIX 协会。[2] Sandrine Blazy 和 Xavier Leroy。2009年。C 语言 Clight 子集的机械化语义。《自动推理杂志》43，第3期（2009年），263–288。https://doi.org/ 10 1007 10817 009 9148 3。/ [3] 2019 s - - - BlueHat。. 内存破坏仍然是最普遍的安全漏洞。https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/。访问日期：2020-02-11。[4] Jay Bosamiya、Benjamin Lim、Bryan Parno 和 Formally Verified。2020年。将 WebAssembly 作为中间语言，用于可证明安全的软件沙箱化。[5] David Brumley 和 Dawn Song。2004年。Privtrans：自动对程序进行分区以实现权限分离。载于 USENIX安全研讨会，第57卷。[6] Jeremy Condit、Matthew Harren、Zachary Anderson、David Gay 和 George C. Necula。2007年。用于低级编程的依赖类型。载于欧洲编程研讨会（ESOP '07）论文集。[7] Junhan Duan、Yudi Yang、Jie Zhou 和 John Criswell。2020年。使用 Checked C 重构 FreeBSD 内核。载于2020年 IEEE 网络安全开发会议（SecDev）论文集。[8] Gregory J Duck 和 Roland HC Yap。2016年。使用低胖指针进行堆边界保护。载于第25届国际编译器构造会议论文集。ACM，132–142。[9] Archibald Samuel Elliott、Andrew Ruef、Michael Hicks 和 David Tarditi。2018年。Checked C：通过扩展使 C 安全。载于2018年 IEEE 网络安全开发会议（SecDev）。 53 60 10 1109 2018 00015 – 。https://doi.org/ 。/ [10] 2012 SecDev。. Chucky Ellison 和 Grigore Rosu。. 具有应用的 C 语言可执行形式语义。载于第39届 ACM SIGPLAN-SIGACT 编程语言原理年度研讨会论文集（美国宾夕法尼亚州费城）（POPL '12）。ACM，美国纽约州纽约，533–544。https://doi.org/10.1145/2103656. 2103719[11] Matthias Felleisen、Robert Bruce Findler 和 Matthew Flatt。2009年。使用 PLT Redex 的语义工程（第1版）。MIT 出版社。[12] Dan Grossman、Greg Morrisett、Trevor Jim、Michael Hicks、Yanling Wang 和 James Cheney。2002年。Cyclone 中基于区域的内存管理。载于 PLDI。[13] Chris Hathhorn、Chucky Ellison 和 Grigore Roşu。2015年。定义 C 语言的未定义性。SIGPLAN Not. 50，第6期（2015年6月），336–345。https://doi.org/10.1145/ 2813885.2737979[14] Christophe Hauser、Jayakrishna Menon、Yan Shoshitaishvili、Ruoyu Wang、Giovanni Vigna 和 Christopher Kruegel。2019年。Sleak：自动化地址空间布局去随机化。载于第35届年度计算机安全应用会议论文集。190–202。[15] Abhinav Jangda、Bobby Powers、Emery D Berger 和 Arjun Guha。2019年。没那么快：分析 {WebAssembly} 与本机代码的性能。载于2019年 USENIX 年度技术会议（USENIX ATC 19）。107–120。[16] Trevor Jim、Greg Morrisett、Dan Grossman、Michael Hicks、James Cheney、和 Yanling Wang。2002年。Cyclone：C 语言的一种安全方言。载于 USENIX 年度技术会议。USENIX，美国加利福尼亚州蒙特雷，275–288。[17] Jeehoon Kang、Chung-Kil Hur、William Mansky、Dmitri Garbuzov、Steve Zdancewic 和 Viktor Vafeiadis。2015年。一种支持整数指针转换的形式化 C 内存模型。SIGPLAN Not. 50，第6期（2015年6月），326–335。https: //doi.org/ 10 1145 2813885 2738005 。/ 。Leonidas Lampropoulos 和 Benjamin C. [18] Pierce。2018年。QuickChick：Coq 中基于性质的测试。电子教材。1.0版。http://www.cis.upenn. edu/~bcpierce/sf。[19] Xavier Leroy、Andrew W. Appel、Sandrine Blazy 和 Gordon Stewart。2012年。CompCert 内存模型，第2版。研究报告 RR-7987。INRIA。26页。https://hal.inria.fr/hal-00703441[20] Liyi Li、Yiyun Liu、Deena Postol、Leonidas Lampropoulos、David Van Horn 和 Michael Hicks。2022年。Checked C 形式模型（扩展版本）。https://arxiv.org/abs/2201.13394。[21] Liyi Li、Yiyun Liu、Deena L. Postol、Leonidas Lampropoulos、David Van Horn 和 Michael Hicks。2022年。Checked C 形式模型。载于计算机安全基础研讨会（CSF）论文集。[22] Joshua Lind、Christian Priebe、Divya Muthukumaran、Dan O'Keeffe、Pierre-Louis Aublin、Florian Kelbert、Tobias Reiher、David Goltzsche、David Eyers、Rüdiger Kapitza 等。2017年。Glamdring：针对 intel {SGX} 的自动应用程序分区。载于2017年 USENIX 年度技术会议（USENIX ATC 17）。285–298。[23] Shen Liu、Gang Tan 和 Trent Jaeger。2017年。PtrSplit：在自动程序分区中支持通用指针。载于2017年 ACM SIGSAC 计算机与通信安全会议论文集。2359–2371。[24] Aravind Machiry、Eric Gustafson、Chad Spensky、Christopher Salls、Nick Stephens、Ruoyu Wang、Antonio Bianchi、Yung Ryn Choe、Christopher Kruegel 和 Giovanni Vigna。2017年。BOOMERANG：利用可信执行环境中的语义鸿沟。. 载于 NDSS。[25] Aravind Machiry、John Kastner、Matt McCutchen、Aaron Eline、Kyle Headley 和 Michael Hicks。2022年。通过 3C 将 C 转换为 Checked C。《ACM 编程语言汇刊》6，OOPSLA1（2022年），1–29。

[26] Kayvan Memarian、Victor B. F. Gomes、Brooks Davis、Stephen Kell、Alexander Richardson、Robert N. M. Watson 和 Peter Sewell。2019年。探索 C 语义与指针来源。《ACM编程语言汇刊》3，POPL，文章 67（2019年1月），32 页。https://doi.org/10.1145/3290380[27] Kayvan Memarian、Justus Matthiesen、James Lingard、Kyndylan Nienhuis、David Chisnall、Robert N. M. Watson 和 Peter Sewell。2016年。深入 C 的内部：阐释事实标准。《SIGPLAN通讯》51，6（2016年6月），1–15。https://doi.org/ 10 1145 2980983 2908081 。/。 [28] Samuel Mergendahl、Nathan Burow 和 Hamed Okhravi。2022年。跨语言攻击。（2022年）。[29] Denis Merigoux、Nicolas Chataing 和 Jonathan Protzenko。2021年。Catala：法律编程语言。arXiv预印本 arXiv:2103.03198（2021年）。[30] 微软。2019年。Checked C评估基准。https://github.com/microsoft/checkedc/wiki/Benchmarks-for-evaluating-Checked-C。访问日期：2020年10月27日。[31] Ana Milanova、Atanas Rountev 和 Barbara G Ryder。2002年。在存在函数指针的情况下进行精确的调用图构建。载于《第二届 IEEE 源代码分析与操作国际研讨会论文集》。IEEE，155–162。[32] Santosh Nagarakatte、Jianzhou Zhao、Milo M.K. Martin 和 Steve Zdancewic。2009年。SoftBound：面向 C 语言的高度兼容且完整的空间内存安全。载于《第 30 届 ACM SIGPLAN 编程语言设计与实现大会论文集》（爱尔兰都柏林）（PLDI '09）。计算机协会，纽约州纽约，美国，245–258。https://doi.org/10.1145/ 1542476.1542504[33] Shravan Narayan、Craig Disselkoen、Tal Garfinkel、Nathan Froyd、Eric Rahm、Sorin Lerner、Hovav Shacham 和 Deian Stefan。2020年。在 Firefox渲染器中改造细粒度隔离。载于《第 29 届 USENIX安全研讨会》（USENIX Security 20）。USENIX 协会，699–716。https://www.usenix.org/conference/usenixsecurity20/presentation/narayan [34] George C. Necula、Jeremy Condit、Matthew Harren、Scott McPeak 和 West- ley Weimer。2005年。CCured：遗留软件的类型安全改造。《ACM编程语言与系统汇刊》（TOPLAS）27，3（2005年）。[35] Vineet Rajani、Deepak Garg 和 Tamara Rezk。2016年。论访问控制、能力、二者的等价性以及混淆代理攻击。载于《2016 IEEE 第 29 届计算机安全基础研讨会》（CSF）。IEEE，150–163。[36] Andrew Ruef、Leonidas Lampropoulos、Ian Sweet、David Tarditi 和 Michael Hicks。2019年。使用 checked c 逐步实现安全。载于《安全与信任原理国际会议》。施普林格，Cham，76–98。[37] Andrew Ruef、Leonidas Lampropoulos、Ian Sweet、David Tarditi 和 Michael Hicks。2019年。使用 Checked C 逐步实现安全。载于《安全与信任原理》，Flemming Nielson 和 David Sands（编）。施普林格国际出版，Cham，76–98。[38] Sean Rul、Hans Vandierendonck 和 Koen De Bosschere。2009年。迈向自动化程序分区。载于《第 6 届 ACM 计算前沿会议论文集》。89–98。[39] Konstantin Serebryany、Derek Bruening、Alexander Potapenko 和 Dmitry Vyukov。2012年。AddressSanitizer：快速的地址健全性检查器。载于《2012年 USENIX年度技术会议论文集》。[40] Julian Seward、Nicholas Nethercote 和 Josef Weidendorfer。2008年。Valgrind 3.3——GNU/Linux 应用程序的高级调试与分析。Network Theory Ltd。[41] Dokyung Song、Julian Lettner、Prabhu Rajasekaran、Yeoul Na、Stijn Volckaert、Per Larsen 和 Michael Franz。2019年。SoK：面向安全的消毒。载于《2019年 IEEE安全与隐私研讨会论文集》（S&P）。[42] Gang Tan 等。2017年。软件基础隔离的原理与实现技术。《隐私与安全基础与趋势》1，3（2017年），137–198。[43] CVE趋势。2021年。C VE趋势。https://www.cvedetails.com/vulnerabilities-by- types.php。访问日期：2020年10月11日。[44] WASI SDK 2022年。wask-sdk 开发者。wasi-sdk。https://github.com/ WebAssembly/wasi-sdk。[45] Bin Zeng、Gang Tan 和 Úlfar Erlingsson。2013年。Strato：面向低级内联引用监视器的可重定向框架。载于《第 22 届 USENIX安全会议论文集》。[46] Feng Zhou、Jeremy Condit、Zachary Anderson、Ilya Bagrak、Rob Ennals、Matthew Harren、George Necula 和 Eric Brewer。2006年。SafeDrive：使用基于语言的技术实现安全且可恢复的扩展。载于《第 7 届操作系统设计与实现研讨会》（OSDI'06）。USENIX 协会，西雅图，华盛顿。

A 附录

A.1 将 C 转换为 Checked C

Checked C 的安全保证伴随着某些限制。例如，如下所示，Checked C 程序不能在边界表达式中使用取地址变量，因为边界关系可能由于通过指针进行的潜在修改而不成立。

```
...
_Array_ptr<int> p : count (n) = NULL;
✖..,&n,.
```

因此，将现有 C 程序转换为 Checked C 可能需要重构，例如，在不改变其功能的情况下，从上述程序中删除 `&n`。根据程序的复杂程度，这可能需要付出相当大的努力 [7]。近期，Machiry 等人开发了 3C [25]，尝试通过添加适当的指针注释来自动将程序转换为 Checked C。然而，正如 3C 中所述，完全自动化的转换不可行，并且需要开发者手动转换某些代码区域。

A.2 良构性与子类型

$$m \vdash \mathsf{int} \qquad \frac{\xi \wedge m \vdash \tau \qquad \xi \leq m}{m \vdash \mathsf{ptr}^{\xi}\ [\beta\ \tau]_{\kappa}} \qquad \frac{\xi \wedge m \vdash \tau \qquad \xi \leq m}{m \vdash \mathsf{ptr}^{\xi}\ \tau}$$

$$\frac{\xi \wedge m \vdash \tau \qquad \xi \leq m \qquad FV(\overline{\tau}) \cup FV(\tau) \subseteq \overline{x}}{m \vdash \mathsf{ptr}^{\xi}\ (\forall\, \overline{x}.\ \overline{\tau} \to \tau)}$$

$$\mathtt{t} \wedge \mathtt{c} = \mathtt{u} \qquad \xi \wedge \mathtt{u} = \mathtt{u} \qquad \mathtt{c} \wedge m = m \qquad m_1 \wedge m_2 = m_2 \wedge m_1$$

图 8：嵌套指针的良构性

$$\Gamma \vdash n \qquad \frac{x : \mathsf{int} \in \Gamma}{\Gamma \vdash x + n} \qquad \frac{\Gamma \vdash b_l \qquad \Gamma \vdash b_h}{\Gamma \vdash (b_l, b_h)} \qquad \Gamma \vdash \mathsf{int}$$

$$\frac{\Gamma \vdash \beta \qquad \Gamma \vdash \tau}{\Gamma \vdash \mathsf{ptr}^{m}\ [\beta\ \tau]_{\kappa}} \qquad \frac{\Gamma \vdash \tau}{\Gamma \vdash \mathsf{ptr}^{m}\ \tau} \qquad \frac{T \in D}{\Gamma \vdash \mathsf{ptr}^{m}\ \mathsf{struct}\ T}$$

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \mathsf{ptr}^{m}\ \tau} \qquad \frac{\forall \tau_i \in \overline{\tau}.\Gamma[\forall x \in \overline{x}.\, x \mapsto \mathsf{int}] \vdash \tau_i \qquad \Gamma[\forall x \in \overline{x}.\, x \mapsto \mathsf{int}] \vdash \tau}{\Gamma \vdash \forall\, \overline{x}.\ \overline{\tau} \to \tau}$$

图 9：类型与边界的良构性

图 8 定义了嵌套指针的良构性，确保没有受污染的（t）指针包含已检查的（c）元素字段。具体而言，对于嵌套指针 $\mathsf{ptr}^{\xi_1}(\ldots\mathsf{ptr}^{\xi_2}\ \tau\ldots)$，$\xi_2 \leq \xi_1$。图 9 定义了类型中边界的良构性，要求任何边界都必须是 `int` 类型，并且在 Γ 中有实例。类型中出现的所有绑定变量都必须是类型环境 Γ 的实例，或是函数指针中出现的绑定变量。

边界不等式与等式：

$$n \leq n' \Rightarrow n \quad \leq_{\Theta} n'$$
$$n \leq n' \Rightarrow x + n \leq_{\Theta} x + n' n \leq n' \wedge \Theta(x)$$
$$= \mathsf{ge_0} \Rightarrow n \quad \leq_{\Theta} x + n' \Theta(x)$$
$$= \mathsf{eq}\ b \wedge b + n \leq_{\Theta} b' \Rightarrow x + n \leq_{\Theta} b' \Theta(x)$$
$$= \mathsf{eq}\ b \wedge b' \leq_{\Theta} b + n \Rightarrow b' \quad \leq_{\Theta} x + n$$
$$b \leq_{\Theta} b' \wedge b' \leq_{\Theta} b \Rightarrow b \quad =_{\Theta} b'$$

类型等价：

$$\mathsf{int} \quad =_{\Theta} \mathsf{int}$$
$$\omega =_{\Theta} \omega' \Rightarrow \mathsf{ptr}^{\xi}\ \omega \quad =_{\Theta} \mathsf{ptr}^{\xi}\ \omega'$$
$$\beta =_{\Theta} \beta' \wedge \tau =_{\Theta} \tau' \Rightarrow [\beta\ \tau]_{\kappa} \quad =_{\Theta} [\beta'\ \tau']_{\kappa}$$
$$\text{条件}(x, \tau \to \overline{\tau}, y, \quad') \Rightarrow \quad .\ \quad \quad .$$
$$\tau' \to \tau \qquad \forall\, x\ \ \tau \to \tau =_{\Theta} \forall\, y\ \ \tau' \to \tau'$$

子类型：

$$\tau =_{\Theta} \tau' \Rightarrow \tau \qquad \sqsubseteq_{\Theta} \tau' \text{指针}^{\mathtt{t}}\ \tau \qquad \sqsubseteq_{\Theta}$$
$$\text{指针}^{\mathtt{u}}\ \tau 0 \leq_{\Theta} b_l \wedge b_h \leq_{\Theta} 1 \Rightarrow \text{指针}^{m}\ \tau \qquad \sqsubseteq_{\Theta} \text{指针}^{m}\ [(b_l, b_h)$$
$$\tau] b_l \leq_{\Theta} 0 \wedge 1 \leq_{\Theta} b_h \Rightarrow \text{指针}^{m}\ [(b_l, b_h)\ \tau] \quad \sqsubseteq_{\Theta} \text{指针}^{m}\ \tau$$
$$b_l \leq_{\Theta} 0 \wedge 1 \leq_{\Theta} b_h \Rightarrow \text{指针}^{m}\ [(b_l, b_h)\ \tau]_{nt} \sqsubseteq_{\Theta} \text{指针}^{m}\ \tau$$
$$b_l \leq_{\Theta} b'_l \wedge b'_h \leq_{\Theta} b_h \Rightarrow \text{指针}^{m}\ [(b_l, b_h)\ \tau]_{nt} \sqsubseteq_{\Theta} \text{指针}^{m}\ [(b'_l b'_h)\ \tau]$$
$$b_l \leq_{\Theta} b'_l \wedge b'_h \leq_{\Theta} b_h \Rightarrow \text{指针}^{m}\ [(b_l, b_h)\ \tau]_{\kappa} \sqsubseteq_{\Theta} \text{指针}^{m}\ [(b'_l b'_h)\ \tau]_{\kappa}$$
$$\tau' \sqsubseteq_{\Theta} \tau \wedge \tau \sqsubseteq_{\Theta} \tau' \Rightarrow \text{指针}^{\xi}\ \forall\, x.\ \tau \to \tau \quad \sqsubseteq_{\Theta} \text{指针}^{\xi}\ \forall\, x.\ \tau' \to \tau'$$
$$n' + n = add(n', n)\ (x + n') + n = x + add(n', n)\ \text{条件}(x, \tau, y, \tau') \ = \exists z\ .\ x \cdot$$
$$\cup\, z \wedge y \cup z \wedge \text{大小}(x) \ = \text{大小}(y) \ = \text{天小}(z) \wedge \tau[z/x] = \tau'[z/x]$$

图 10：类型相等与子类型关系

在 CoreChkCBox 中，类型相等 $\tau =_{\Theta} \tau'$ 是一种类型构造等价关系，它由图 10 中（NT-）数组指针类型的有界相等（$=_{\Theta}$）以及两个函数类型的 α 等价定义；也就是说，如果 $\beta =_{\Theta} \beta'$ 和 $\tau =_{\Theta} \tau'$，则两个（NT-）数组指针类型 $[\beta\ \tau]_{\kappa}$ 和 $[\beta'\ \tau']_{\kappa}$ 等价；如果我们能找到一个与 x 和 y 长度相同的变量列表 z，分别将其替换到 $\tau \to \tau$ 和 $\tau' \to \tau'$ 中的 x 和 y，且替换结果相等，则两个函数类型 $\forall\, x.\ \tau \Rightarrow \tau$ 和 $\forall\, y.\ \tau' \to \tau'$ 等价。

T-CastPtr 规则允许将类型为 τ' 的表达式转换为已检查指针，条件是 $\tau' \sqsubseteq \mathsf{ptr}^{\mathtt{c}}\ \tau$。这一子类型关系

$$\sqsubseteq \qquad 10 \qquad \tau =_{\Theta} \tau \ \Rightarrow$$

见图 Fig.,

$$\tau \sqsubseteq_{\Theta} \tau$$

并建立在类型相等（,,）之上。众多规则确保了该关系具有传递性。其中大多数规则处理数组指针类型之间的类型转换。规则（$0 \leq b_l \wedge b_h \leq 1 \Rightarrow \mathsf{ptr}^{m}\ \tau \sqsubseteq \mathsf{ptr}^{m}\ [(b_l, b_h)\ \tau]$）允许将单例指针视为数组指针，其中包含 $b_h \leq 1$ 和 $0 \leq b_l$。当输出类型具有子类型关系（$\tau \sqsubseteq_{\Theta} \tau'$）且参数类型具有逆向子类型关系（$\tau' \sqsubseteq_{\Theta} \tau$）时，两个函数指针类型具有子类型关系（$\mathsf{ptr}^{\xi}\ \forall\, x.\ \tau \to \tau \ \sqsubseteq_{\Theta} \mathsf{ptr}^{\xi}\ \forall\, x.\ \tau' \to \tau'$）。 — –

图 10 所示的子类型关系涉及依赖界限，也就是可能引用变量的边界。为了判定图 10 中的前提 $b \leq_{\Theta} b'$，我们需要一种能够考虑这些变量可能取值的判定过程。此过程会考虑由类型判断跟踪的 Θ，以及当前的栈快照 φ（在类型保持证明中进行子类型判定时）。

由于边界表达式可能包含变量，要确定诸如 $b_l \leq_{\Theta} b'_l$ 这样的假设，需要推理这些变量可能取的值。类型系统使用 Θ 使此类推理更加精确。Θ 是从变量 x 到方程谓词 P 的映射，其形式为 $P ::= \mathsf{ge_0} \mid \mathsf{eq}\ \ b$。它将

变量映射到在类型检查过程中记录的方程。如果 Θ 将 x 映射到 ge_0，这意味着 $x \geq 0$；eq b 表示 x 在当前上下文中等价于边界值 b，例如规则 T-LetInt 和 T-RetInt 中 e_2 的类型判断。

$\sqsubseteq$ 由 Θ参数化，它为有界变量提供允许值的范围；因此，可以证明更多的 $\sqsubseteq$ 关系。例如，规则 T-LetInt 为变量 x插入谓词 eq e_1。假设 e_1 等于 0，当 x 在类型 $\text{ptr}^{c}\ [(0,x)\ \text{int}]_{nt}$ 中用作类型变量时，如果我们知道 xeq 0，则可以证明子类型关系 $\text{ptr}^{c}\ [(0,x)\ \text{int}]_{nt} \sqsubseteq \text{ptr}^{c}\ [(0,0)\ \text{int}]_{nt}$。

为在依赖类型中捕获绑定变量，Checked C 子类型关系（$\sqsubseteq$）由受限栈快照 $\varphi|_\rho$和谓词映射 Θ参数化，其中 φ是一个栈，而 ρ是一组变量。$\varphi|_\rho$表示将 φ的域限制为变量集 ρ。显然，我们有如下关系：$\varphi|_\rho \subseteq \varphi$。$\sqsubseteq$由 $\varphi|_\rho$参数化，指的是当我们比较两个边界 $b \leq_\Theta b\prime$时，实际上通过使用 $\varphi|_\rho$中的可能值解释 b和 $b\prime$中的变量，执行 $\varphi|_\rho\ (b) \leq_\Theta \varphi|_\rho\ (b\prime)$。我们为两个受限栈快照 $\varphi|_\rho$和 $\varphi\prime|_\rho$定义子集关系 $\preceq$:

定义 1（栈快照子集）。给定两个 $\varphi|_\rho$和 $\varphi\prime|_\rho$，$\varphi|_\rho \preceq \varphi\prime|_\rho$，当且仅当对于 $x \in \rho$和 y，$(x,y) \in \varphi|_\rho \Rightarrow (x,y) \in \varphi\prime|_\rho$。

对于任意两个受限栈快照 $\varphi|_\rho$和 $\varphi\prime|_\rho$，满足$\varphi|_\rho \preceq \varphi\prime|_\rho$，我们有以下 Checked C 中的定理（已在 Coq 中证明）：

定理 6（栈快照定理）。给定两个类型 τ和$\tau\prime$，两个受限栈快照 $\varphi|_\rho$和 $\varphi\prime|_\rho$，若 $\varphi|_\rho \preceq \varphi\prime|_\rho$，且$\tau \sqsubseteq \tau\prime$在 $\varphi|_\rho$的参数化下成立，则 $\tau \sqsubseteq \tau\prime$在 $\varphi\prime|_\rho$的参数化下成立。

显然，对于每个 $\varphi|_\rho$，我们都有 $\emptyset \preceq \varphi|_\rho$。类型检查阶段是一个编译时过程，因此在类型检查阶段，$\varphi|_\rho$是 $\emptyset$。证明类型保持需要栈快照，因为边界表达式中的变量会被求值消去。

A.3 其他类型规则和字面量有效性检查

这里我们展示图 11 中其他 CoreChkCBox 操作的类型规则。规则 T-Mac 处理 malloc 操作。其中有一项良构性检查，要求 ω 中可能出现的有界变量必须属于 Γ 的定义域（见图 9）。这与类型环境的良构性假设（定义 A.1）类似。规则 T-Add 处理子项为整数表达式的二元操作。其他规则说明如下。

指针访问。T-AssignArr 规则检查数组赋值操作，并返回指向对象的类型。针对其他对象类型的指针，规则也类似；例如，T-Assign 规则将一个值赋给非数组指针内存位置。条件 $\xi \leq m$ 确保已检查指针和未检查指针只能分别在已检查区域和未检查区域中解引用；类型规则不尝试判断访问是否在边界内，而是将此类检查留给语义处理。T-Ind 规则适用于指针算术。为简单起见，在 Checked C 的形式化定义中，我们不允许任意指针算术。唯一允许的指针算术操作是图 11 中规则 T-Ind 和 T-IndAssign 所示的形式。谓词 $\tau\prime \sqsubseteq_\Theta \tau$ 要求被赋值的值是指针类型的子类型。T-IndAssign 规则是一种扩展的赋值操作，用于处理带指针算术的数组/NT-数组指针赋值。

字面常量有效性。规则 T-ConstU 和 T-ConstC 描述程序中字面量的类型假设。$\text{c}(\tau)$判定未受检区域中的字面量指针不能具有受检类型，这体现了一项假设：程序员无法猜出受检指针地址，并在 CheckedCBox 的未受检区域中使用它。规则 T-ConstC要求通过静态验证过程验证字面量指针，这与第 4.2 节中的动态验证过程类似。

验证过程 $\Theta; \mathcal{H}; \sigma \vdash_m n: \tau$检查（图 12）用于验证字面量 $n: \tau$，其中$\mathcal{H}(m)$ 是该字面量所在的初始堆，而 σ 是一个假定为已检查的字面量集合。要检查函数指针的有效性，还需要全局函数存储区 $\Xi(m)$。有效的函数指针应出现在正确的存储区域（c 或u），且该地址存储的函数具有正确的类型。图 12 中的最后一条规则描述了非函数指针的有效性检查，其中指针范围（[0,大小(ω))中的每个元素都应类型良好。

已检查指针会像规则 T-ConstC 那样在类型步骤中检查有效性，而污点/未检查指针在类型检查期间不会进行此类检查。如上所述，污点指针会在动态执行期间通过有效性检查进行验证。

如果字面量的类型是整数、未检查指针或空指针，则该字面量类型正确，如图 12 中最上方的三条规则所示。然而，如果它是已检查指针 $\text{ptr}^{c}\ \omega$，我们就需要确保它在堆中指向的内容具有适当的所指类型（ω），并递归地确保通过这种方式可访问的任何字面量指针也都是类型良好的。图中的底部规则对此进行了说明：对于指针范围 $[n, n+\text{大小}(\omega))$ 内的每个内存位置 $n+i$，其中 大小 会返回其参数的大小，该内存位置 $\mathcal{H}(n+i)$ 处的值也必须类型良好。然而，由于堆快照可能包含循环结构（这会导致类型推导无限延伸），我们使用作用域 σ，在检查指针所指内容的类型时，假定原始指针类型良好。中间规则随后访问该作用域以闭合递归，保持推导有限，这与 Ruef 等人 [37] 的方法相同。

Let 绑定。规则 T-Let 和 T-LetInt 为 let 表达式确定类型，该表达式也允许类型依赖。具体而言，对 let 求值的结果可能具有引用其某个绑定变量的类型（例如，如果结果是带有变量定义边界的已检查指针）。此时，一旦该变量离开作用域，我们就必须将其替换掉（T-LetInt）。注意，我们限制表达式e_1 在语法上必须匹配 Bounds 表达式 b 的结构（见图 2）。当 x 的类型为 int 时，规则 T-RetInt 为 ret 表达式确定类型。ret 不会出现在源程序中，而是在对 let 绑定求值时由语义引入（图 13 中的规则 S-Let）。

A.4 其他语义规则

图 13 展示了 CoreChkCBox 的其余语义规则。本小节将解释其中几个具有代表性的规则。

已检查与受污染指针操作。规则 S-Var 会从栈中加载 x的值 φ。规则 S-DefArrayC 和 S-DefArrayT 分别对 c 和 t 数组指针进行解引用。在 CoreChkCBox 中，数组与 NT-array 解引用的区别在于范围

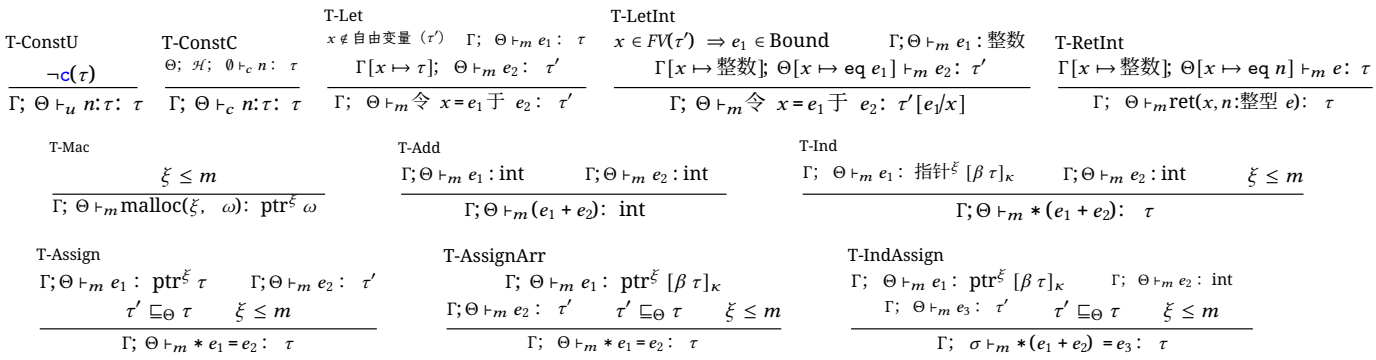


图 11：其余 CoreChkCBox 类型规则（扩展图 4）

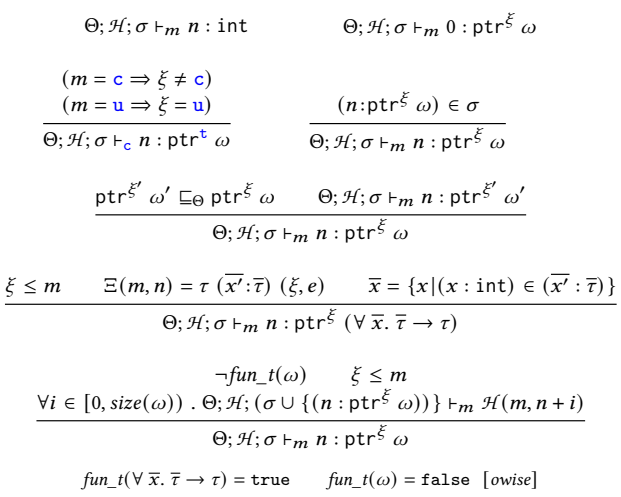


图 12：常量的验证/类型规则

的 0 位于 $[n_l, n_h)$ 不 $[n_l, n_h]$，这意味着无法对数组中位于上界位置的元素进行解引用。规则 DefArrayBound 和 DefNTArrayBound 描述了解引用操作的错误情况。如果我们对数组/NT-array 指针进行解引用，且模式为 c （或 t），0 必须位于以下范围内： n_l至 n_h（这表示解引用未越界）；否则，系统会产生 边界 错误。显然，对数组/NT-array 指针进行解引用时，如果 $n \le 0$，也会发生 空值状态转换。

规则 S-Malloc 和 S-MallocBound 描述了 malloc 的语义。给定一个不包含自由变量的有效类型 ω_a，对于特定模式 ξ，alloc 函数会返回一个地址，该地址指向已分配空间的第一个位置，且该空间的大小等于 ω_a 的大小；此外还会返回一个新的堆快照 H′，用于标记此次新分配的已分配空间。如果 ξ 为 c，则新的 H 分配位于 c 区域；而 t 和 u 模式的分配则会在 u 区域中创建新的空间。malloc 会转换为地址 n，其类型为 $\text{ptr}^{\xi}\ \omega_a$，并得到更新后的新堆。如果 ω_a 是数组/NT-数组类型 $[(n_l, n_h)\ \tau]_\kappa$，malloc 也可能转换为边界错误，并且或者 $n_l \neq 0$，或者 $n_h \le 0$。当有界变量求值为不符合要求的边界常量时，就可能发生这种情况。

规则 S-AssignArrC 和 S-AssignArrT 用于赋值操作。只要 0（解引用的位置）位于指针注释指定的边界内，且严格小于上界，S-AssignArrC 就会向数组赋值。规则 S-AssignArrT 与 S-AssignArrC 类似，但适用于受污染的指针。对受污染指针进行任何动态堆访问都需要进行验证。执行这种验证等同于对图 12 中的指针常量进行字面量类型检查。

Let 绑定。语义使用特殊的 ret 形式管理变量作用域。S-Let 求值得到一个配置，其表达式为 $\text{ret}(x, \varphi(x)\ e))$。它会将 x 的旧值保存在 ret 的栈中，并用 let 绑定的新值更新 φ 中 x 的值。在将 e 求值为一个值 $n{:}\tau$ 后，我们会将 φ 的值恢复为 ret 的栈中存储的值 μ，这一点在规则S-Ret中说明。

A.5 编译形式化

将 Checked C 编译为 Clang/LLVM 时，主要的微妙之处在于捕获指针字面量上的注释，这些注释用于跟踪数组边界信息，并在 S-DefArrayC 和 S-AssignArrC 等规则的前提中使用，以防止空间安全违规。Checked C 编译器 [21]插入了额外的指针检查，以验证指针非空且边界未超出限制。后者通过引入额外的影子变量来存储（NT-）数组指针的边界信息来实现。

在 CheckedCBox中，上下文模式和指针模式决定指针所指向的特定堆/函数存储区，即 c 指针指向已检查区域，而 t 和 u 指针指向未检查区域。未检查区域关联着一种沙箱机制，用于对潜在的内存故障进行异常处理。在编译后的 LLVM 代码中，模式不同时，指针访问操作的语法也不同。图 14 列出了编译器实现（CBox，代表 CheckedCBox）和形式化（Core，代表 CoreChkCBox）中解引用操作（$*\,x$）编译后的不同语法。列表示不同的指针模式，行表示不同的上下文模式。例如，当 t模式指针位于 c模式区域中时，我们会将解引用操作编译为沙箱指针访问函数（sand_get(x)），以访问 CheckedCBox

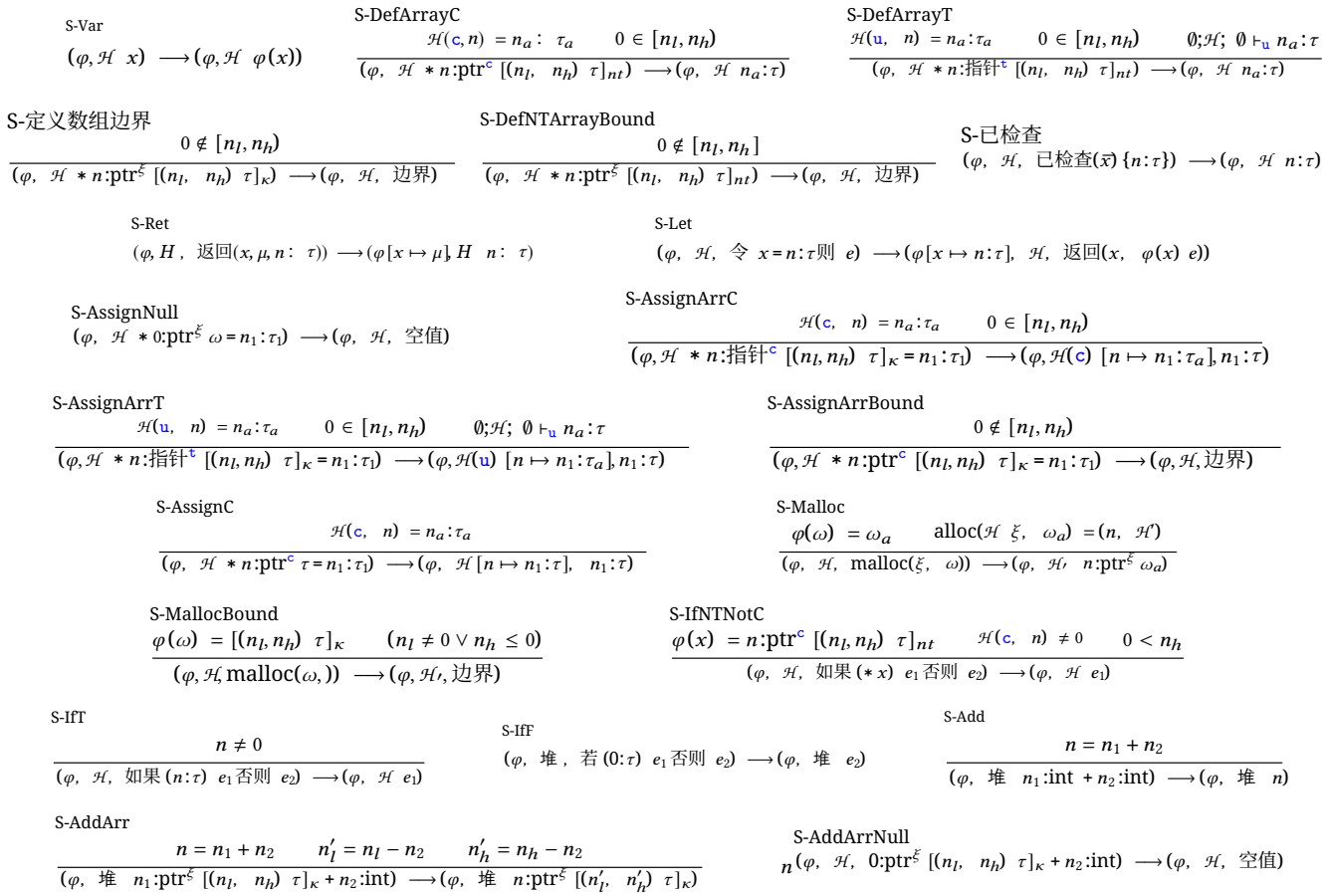


图 13：其余 CoreChkCBox 语义规则（扩展图 4）

	c	t	u
	CBox / Core	CBox / Core	CBox / Core
c	$*x$/ $*(c, x)$	sand_get (x) / $*(u, x)$	×
u	×	$*x$/ $*(u, x)$	$*x$/ $*(u, x)$

图 14：解引用的编译目标

实现中。在 CoreChkCBox中，我们在现有的 $*x$操作（在 LLVM 中）之上创建了一种新的解引用数据结构：$*(m, x)$。如果模式为 c，则访问已检查堆/函数存储区；否则，访问未检查的堆/函数存储区。

本节展示 CoreChkCBox 如何处理指针模式、模式切换和函数指针编译；由于 Checked C 包含注释的擦除，因此不会损失表达能力 [21]。关于编译器形式化，我们提出一种编译算法，将 CoreChkCBox 转换为 CoreC。CoreC 是一种没有元数据注释的无类型语言，它代表了我们在 LLVM 上构建的中间层，可简化编译。在 CoreC 中，解引用、赋值、malloc 和函数调用的语法分别为：$*(m, e)$、$*(m, e) = e$、$\text{malloc}(m, \omega)$ 和 $e(e)$。该算法阐明了如何实现编译

在实际的 CheckedCBox 编译器中，同时略去了许多重要细节（CoreC 与 LLVM IR 有许多不同之处）。

编译通过扩展 CoreChkCBox 的类型判断定义如下：

$$\Gamma; \Theta; \rho \vdash_m e \gg \dot{e} : \tau$$

现在有一个 CoreC 输出 $\dot{e}$ 和一个输入 ρ，它将每个（非类型）数组指针变量映射到其模式，并将每个变量p 映射到一对 影子变量，用于保存 p的最新上界和下界。这些边界可能与 p 的类型中的边界不同，这是由于边界扩宽所致。[6]

我们在 PLT Redex 中形式化了这一判断的规则，[11], 遵循并扩展了针对 CoreChkCBox 的 Coq 开发。为了确信编译正确，我们使用 Redex 的基于属性的随机测试支持，证明编译到 $\dot{e}$ 的结果会模拟 e，对于所有 e均成立。

方法。这里我们通过示例说明编译规则，使用类 C 语法；完整规则见 [20]。每条规则最多执行三项任务：(a) 将 e 转换为 A-范式；(b) 插入动态检查和界限扩宽表达式；以及

[6]由于下界从不扩大，下界影子变量并非必需；我们将其纳入以保持一致。

```
1  int deref_array(n : int,
2      p :指针c [(0,n) 整数]nt,
3      q :指针t [(0,n) 整数]nt) {
4   /* ρ(p) = p_lo,p_hi,p_m */
5   /* ρ(q) = q_lo,q_hi,q_m */
6     * p;7      * q = 1;8   }9 ...
10 /* p0 : 指针c [(0,5) 整数
]nt */11  /* q0 : 指针t [(0,5) 整
数]nt */
12 deref_array(5, p0, q0);
```

```
1  deref_array(int n, int* p, int * q) {
2   //m is the current context mode
3   let p_lo = 0; let p_hi = n;
4   let q_lo = 0; let q_hi = n;
5   /* runtime checks */
6   assert(p_lo ≤0 && 0 ≤p_hi);
7   assert(p != 0);8     *(mode(p) ∧ m,p);
9   verify(q, not_null(m, q_lo, q_hi)
10            && q_lo ≤0 && 0 ≤q_hi);
11   *(mode(q) ∧ m,q)=1;12   }13   ...
14 deref_array(5, p0, q0);
```

图15：依赖函数的编译示例

(c) 根据模式生成正确的指针访问表达式。A-范式转换很简单：通过将子表达式的结果存储到临时变量中来处理复合表达式，如下例所示。

```
let y=(x+1)+(6+1)   →   let a=x+1;
                        let b=6+1;
                        let y=a+b
```

这简化了对子表达式所产生的副作用的管理。接下来的两个编译步骤更值得关注。我们根据不同的 CoreChkCBox 操作来阐述这两个步骤。

指针访问与模式。对于每个指针声明，如果该指针是（NT-）数组，我们首先分配两个影子变量，用于跟踪可能因指针算术和 NT 数组边界扩展而改变的下界和上界。每个 c-mode NT-数组指针变量都与其存储中的类型信息相关联。此外，我们还会插入边界检查和空指针检查，例如图15中的第 6 行和第 7 行。此外，在形式化表示中，每次使用受污染的指针之前（图15中的第 9 行和第 10 行），都会插入一个类似于图 12 的验证步骤，用于检查指针在堆中是否已定义（非_空值），以及是否满足空间安全性。谓词 not_null 检查指针范围内（p_lo 和 p_hi）的每个元素是否都已在堆中定义。编译后的解引用（*(mode(p)∧m,p)）和赋值（*(mode(q)∧m,q)=1）操作中的模式，是根据指针模式与当前上下文模式的交操作（∧）指针模式（p）与当前上下文模式（m）。

已检查块与未检查块。在 CheckedCBox 的实现中，未检查块和已检查块会编译为由沙箱提供的上下文切换函数。unchecked(x){e} 会编译为 sandbox_call(x, e)，其中我们调用沙箱来执行表达式 e，并传入参数 x。checked(x){e} 会编译为 callback(x, e)，其中我们在沙箱内回调已检查块代码 e。在 CheckedCBox 中，我们采用一种积极的执行方案，直接从编译后的汇编代码中获取指针地址，以触发 _Callback。在形式化描述中，为简洁起见，我们依靠类型系统来保证上下文切换，而不创建额外的函数调用。

函数指针与调用。函数指针的管理方式与普通指针类似，但我们会插入检查，以确认函数存储区中的指针地址非空，并检查其类型表示是否正确；这适用于c 和t 模式的指针 [7]。函数调用的编译方式（编译为(m, e)(e)）与图 14 中指针访问操作的处理方式类似。对于依赖函数调用的编译，图15 提供了参考。请注意，数组指针p 的边界不会作为参数传递，而是根据p的类型进行初始化——参见图顶部原始CoreChkCBox 程序的第 4 行。生成代码的第3 行将下界设为0，并将上界设为n。

A.6 约束与元理论

这里，我们首先给出第 4.3 节所需的一些良构性和一致性定义，然后给出 CoreChkCBox 编译器的模拟定理。类型健全性依赖于以下几种良构性：

定义 *A.1*（类型环境良构性）。如果 Γ 中作为类型边界提及的每个变量，都由 Γ 中具有 int 类型的变量界定，则类型环境 Γ 是良构的。

定义 *A.2*（堆良构性）。对于每个 m，如果 (i) $\mathcal{H}(m, 0)$ 未定义，并且 (ii) 对于 $\mathcal{H}(m)$ 值域中的所有 $n{:}\tau$，类型 τ 不包含自由变量，则堆 $\mathcal{H}$ 是良构的。

定义 *A.3*（栈良构性）。如果对于 φ 值域中的所有 $n{:}\tau$，类型 τ 都不包含自由变量，则栈快照 φ 是良构的。

我们还需要引入一致性的概念，将归约步骤前后的堆环境，以及类型环境、谓词集和栈快照联系起来。

定义 *A.4*（栈一致性）。类型环境 Γ、变量谓词集 Θ 和栈快照 φ 是一致的（记作$\Gamma; \Theta \vdash \varphi$），当且仅当对于每个变量 x，若 $\Theta(x)$ 有定义，则存在某个 τ，使得 $\Gamma(x) = \tau$，且 $\varphi(x) = n{:}\tau'$存在某个 n, τ'，其中 $\tau' \sqsubseteq_\Theta \tau$。

定义 *A.5*（经检查的栈-堆一致性）。栈快照 φ与堆$\mathcal{H}$一致$\mathcal{H} \vdash \varphi$——记作 x， $\varphi(x) = n{:}\ \tau$对于模式$(\tau) =$c 意味着 $\emptyset$；$\mathcal{H}(c)$；$\emptyset \vdash_c n{:}\ \tau$。

[7] c 模式的指针在开始时检查一次， t 模式的指针则在每次使用时检查

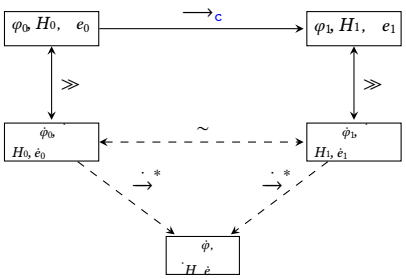


图16：CoreChkCBox 与 CoreC 之间的模拟

定义 *A.6*（已检查堆-堆一致性）。堆$\mathcal{H}_\mathcal{C}$与$\mathcal{H}$一致——记作$\mathcal{H} \triangleright \mathcal{H}_\mathcal{C}$——如果对于每个常量 n，$\emptyset \quad \emptyset \vdash_c n \quad \tau \qquad \emptyset \quad \emptyset \vdash_c n \quad \tau$ ；$\mathcal{H}$；：意味着 ;$\mathcal{H}_\mathcal{C}$;：。

我们在为 CoreChkCBox 开发的 PLT Redex 模型中形式化了编译过程和模拟定理（见第 4.1 节），然后尝试利用 Redex 对随机测试的支持来证伪它。Redex 允许我们将编译指定为逻辑规则（即类型检查的扩展），再以算法方式执行这些规则，从而自动测试模拟是否成立。这个过程揭示了编译过程和定理陈述中的若干错误。我们最终计划在 Coq 模型中证明模拟定理。

我们使用记号 $\gg$ 表示栈和堆的擦除——右侧与左侧相同，但移除了类型注释：

$$\mathcal{H} \gg \dot{\mathcal{H}}$$

$$\varphi \gg \dot{\varphi}$$

此外，当 Γ；$\emptyset \vdash \varphi$ 和 φ 良构时，我们用 $(\varphi,\ \mathcal{H}\ e) \gg_m (\dot{\varphi},\ \dot{\mathcal{H}}\ \dot{e})$ 表示 $\varphi \gg \dot{\varphi}$、$\mathcal{H} \gg \dot{\mathcal{H}}$ 以及 Γ；Θ；$\emptyset \vdash_m e \gg \dot{e}$：$\tau$ 分别对于某个 τ 成立。记号中省略了 Γ，因为 φ 的良构性及其相对于 Γ 的一致性意味着 e 必须在 φ 下封闭，因此可以从 φ 恢复 Γ。最后，我们用 $\cdot\rightarrow^*$ 表示 CoreC 的归约关系的传递闭包。与 CoreChkCBox 不同，CoreC 的语义不区分已检查区域和未检查区域。

图 16 概述了模拟定理。[8] 该模拟定理的表述方式与 Merigoux等人 [29]的表述方式类似。

通常的模拟性质会将图中的中间部分和底部部分替换为以下内容：

$$(\dot{\varphi}_0, \dot{\mathcal{H}}_0, \dot{e}_0) \dot{\rightarrow}^* (\dot{\varphi}_1, \dot{\mathcal{H}}_1, \dot{e}_1)$$

相反，我们使用关系 ~关联两个擦除配置，该关系只要求这两个配置最终规约到相同的状态。

定理 7（模拟（~））。对于 *CoreChkCBox* 表达式 e_0、栈 φ_0、φ_1 以及堆快照 H_0、H_1，若 $H_0 \vdash \varphi_0$、$(\varphi_0$、H_0、$e_0) \gg_c (\dot{\varphi}_0$、$\dot{H}_0$、$\dot{e}_0)$，并且存在某个 r_1 使得 $(\varphi_0$、H_0、$e_0) \rightarrow_c (\varphi_1$、$H_1$、$r_1)$，则以下事实成立：

- 若存在 e_1 使得 $r = e_1$ 且 $(\varphi_1, H1, e_1) \gg (\dot{\varphi}_1, \dot{H}1, \dot{e}_1)$，则存在某个 $\dot{\varphi}$、$\dot{H}$、$\dot{e}$，使得 $(\dot{\varphi}_0, \dot{H}0, \dot{e}_0) \cdot\rightarrow^* (\dot{\varphi}, \dot{H}\ \dot{e})$，且 $(\dot{\varphi}_1, \dot{H}1, \dot{e}_1) \cdot\rightarrow^* (\dot{\varphi}, \dot{H}\ \dot{e})$。
- 若 r_1 = 为边界或空值，则有 $(\dot{\varphi}_0, \dot{H}0, \dot{e}_0) \cdot\rightarrow^* (\varphi_1, \dot{H}1, r_1)$，其中 $\varphi_1 \gg \dot{\varphi}_1$，$H_1 \gg \dot{H}1$。

由于我们的随机生成器不会生成未检查表达式（其行为可能未定义），因此我们只能测试模拟定理关于已检查代码的部分。这一限制意味着无需陈述模拟定理的另一个方向，即 e_0 陷入停滞的情形，因为定理 1 保证，如果 e_0 在检查模式下类型良好，就永远不会进入停滞状态。

当前版本的 Redex 模型已经针对 21500 个深度小于 12 的表达式进行了测试。每个表达式都可能规约多个步骤；我们会在每一对相邻步骤之间测试模拟，以涵盖范围更广的程序，尤其是具有非空堆的程序。

A.7 其他程序评估

这里介绍其他程序评估的描述。**Parsons**。Parsons 在两个变体 parsons_wasm 和 parsons_tainted 中都进行了全面注释。parsons_wasm 将大部分输入解析函数移入沙箱，同时将所有指针标记为污点指针。这些沙箱化函数通过 RLBOX 的回调机制进行间接调用，与已检查区域交互。不过，在 parsons_tainted 中，我们没有将任何函数移入沙箱，但仍将所有指针标记为污点指针。测试套件本身包含 328 项测试，全面检验 JSON 解析器的功能。两个分支的基准测试结果均通过执行测试套件连续 10 次，计算 CheckedCBox 与 generic-C/checked-C 变体之间的平均差异得出。由于 WebAssembly 的性能限制，parsons_wasm 与 checked-C 和 generic-C 相比，预期分别产生 200% 和 266% 的运行时开销。然而，将 parsons_tainted 与 checked-C 比较时，CheckedCBox 的速度更快，因为与 checked-C 对已检查指针执行的运行时边界检查相比，CheckedCBox 本身对污点指针执行的运行时插桩开销更低。此外，我们观察到的平均峰值内存仅为 9.5 KiB，而非预期的 82 KiB 开销，这是因为 Valgrind 不会将分配给污点指针的 WASM 阴影内存计入其中。

LibPNG。CheckedCBox 对 libPNG 的改动范围很窄，首先封装了 CVE-2018-144550，并修复了 compare_read() 中的缓冲区溢出。此外，我们还将 Lib-png 中涉及对用户输入图像数据进行读取、写入和图像处理（交错、像素内处理等）的部分标记为受污染。也就是说，图像字节的行会被读入污点指针，而该行_字节的污点状态会在整个程序中传播。我们的所有改动也适用于 png2pnm 和 pnm2png 可执行文件。为评估 png2pnm，我们对测试脚本运行 10 次取平均值；该脚本会在 libpng 的 pngsuite 中的 52 个 PNG 文件上运行 png2pnm。为测试 pnm2png，我们对 pnm2png 将一个 52 MB、5184×3456 像素的大型 PNM 图像文件转换为 PNG 的过程运行 10 次并取平均值。Valgrind 报告称，转换后的 CheckedCBox 代码堆空间消耗较低，这是因为沙箱的影子内存所占用的堆空间未计入统计。

[8] 为便于阅读，图中省略了 $\dot{e}_1$ 求值为 边界 或 空值 的可能性。

因此，在评估 pnm2png 时，CheckedCBox 的堆内存消耗低了 52 MB，因为整幅图像都加载到了影子内存中。

MicroHTTPD。MicroHTTPD 展示了将程序转换为 CheckedCBox 在实际操作中面临的困难。我们对该程序的转换旨在通过沙箱化来防护内存漏洞 CVE-2021-3466 和 CVE-2013-7039。CVE-2021-3466 被描述为一种由缓冲区溢出引起的漏洞：未受保护的“memcpy”会将数据复制到一个结构体指针（struct MHD_PostProcessor pp）中，而该指针被强制转换为字符缓冲区（char *kbuf = (char *) &pp[1]）。我们的修改需要将该指针标记为受污染，以确保“memcpy”安全。但这要么需要编组该结构体指针（及其子结构体指针成员）所指向的数据，要么需要将对该结构体指针的每处引用都标记为受污染，进而还需要将该结构体的每个指针成员都标记为受污染。结构体指针之间的数据编组并不容易，而且由于其指针成员在空间上具有非线性特征，因此需要编写大量编组代码；字符指针则没有这个问题。这与我们的转换目标不符，因为我们的目标是尽可能减少改动。因此，CheckedCBox 未处理上述 CVE。针对 CVE-2013-7039，我们将该函数的用户输入数据参数标记为污点指针；为尽量减少转换改动，我们没有在这些函数中传播污点。考虑到在错误被发现之前不可能对其进行沙箱化这一时间顺序上的限制，以及程序员的一般直觉，我们将许多核心内部函数（例如 MHD_str_pct_decode_strict_() 和 MHD_http_unescape()）移入了沙箱。

Tiny-bignum。由于规模小且结构简单，我们选择对 Tiny-bignum 进行全面的 CheckedCBox 修改。此外，由于使用 sprintf() 时存在内存不安全问题，已将 bignum_to_string() 移至沙箱中。

ProFTPD：。对 ProFTPD 的 CheckedCBox 修改规模较小，并且通过将传递给不安全函数 "pr_netio_telnet_gets()" 的用户输入标记为受污染，精确地封装了 CVE-2010-4221。尽管我们传播了上述函数参数的污点状态，但按照第 6.2 节所述方法测得的运行时开销仍然很小，因为该指针的数据流图规模较小，因此需要注释的指针也较少。

UFTPD。对 UFTPD 的 CheckedCBox 修改旨在将 CVE-2020-14149 和 CVE-2020-5204 纳入沙箱。CVE-2020-14149 被记录为 handle_CWD() 中的空指针解引用，在 2.12 之前的版本中可能导致拒绝服务，因此我们需要将此函数置于沙箱中。CVE-2020-5204 被记录为 handle_PORT() 中由 sprintf() 导致的缓冲区溢出漏洞，因此同样需要将此函数置于沙箱中。尽管我们本可以只将存在故障的指针标记为受污染，但我们希望使修改更加通用。