|  |
| --- |
| **一、创新训练总结报告：**不少于3000字；本页不够，可自行加页。 |
| 1. **参加创新训练项目清单**   傅泽，计算机学院2023级软件工程专业硕士。在读期间，完成Rupta Rust编程语言分析工具的研究与改进的创新训练工作，详情请参见总结报告。  **2、总结报告**  本次创新训练针对已有的Rupta Rust编程语言分析工具进行改进，目的为：增添其输出信息的丰富性和多样性，使之可被用于死代码移除。简而言之，需要额外增添下列功能：   * 命令行参数增加—overall-metadata <filename.json>，用于输出完整的函数调用信息 * 在输出的函数调用信息中，首先标注当前软件包（下称为crate）涉及到的所有crate，其次标注所有函数的信息，最后标注所有函数调用发生的位置。   为实现上述功能，需分为几步分别实现。  2.1 环境配置  本次实践采用Visual Studio Code配合Rust Analyzer插件进行编码工作。在开始工作前，由于Rust Analyzer对rustc\_private组件报unresolve extern crate错误，影响正常的错误诊断，故需要事先进行解决，总结为4步：   1. 给rustup安装新组件，rustup component add rustc-dev 2. 在VS Code的设置中，将rust-analyzer.rustc.source设置为discover 3. 在当前crate包的Cargo.toml中填上这样两行：  * [package.metadata.rust-analyzer] rustc\_private = true  1. 重启Rust Analyzer   2.2 熟悉Rupta的代码结构  作为MIRAI的改进型，Rupta的代码结构和MIRAI具有相当的相似性。恰好笔者具有阅读MIRAI项目源代码的经验，故可迁移至此项目上。以入口函数**src/bin/cargo-pta.rs**  为例，其运行流程图如下图所示。这个源代码是整个项目开始运行的起点。追溯其流程图并细化，很容易就会发现它的运行逻辑十分简单，其重点就在于PTACallbacks的定义，以及rustc\_driver::catch\_fatal\_errors的调用。调用Rustc MIR API的一种办法，是在fenxiqi 代码中初始化一个Compiler对象，并为该对象设置不同阶段的回调函数。命令该Compiler对象进行代码分析获得MIR后，回调函数将会被调用，从而允许开发者对已经编译好的MIR进行访问等操作。PTACallbacks定义于src/pta/mod.rs，因此需要进入这个文件进行分析。    首先来看看PTACallbacks结构体的定义。它只有两个成员：  • options：是AnalysisOptions结构体，用于配置分析过程的参数。  • file\_name：待分析（编译）的文件的路径。  简明易懂！然后我们再来看看它是如何实现rustc\_driver::Callbacks需要的那些回调函数的。   * config回调函数：这个简单，接受一个rustc\_interface::interface::Config参数，将自身的file\_name设置为该参数记载的源文件名。例如src/main.rs。 * after\_analysis回调函数：这个也很简单，接受一个rustc\_interface::interface::Compiler参数和一个rustc\_interface::queries::Queries，对前者进行错误检查（abort\_if\_errors），若无误则对后者执行enter方法。该方法接受一个闭包，该闭包接受一个TyCtxt，并运行self.run\_pointer\_analysis方法。   我们来瞅瞅PTACallbacks::run\_pointer\_analysis。 - 首先创建并启动了一个rupta::util::mem\_watcher::MemWatcher。 - 创建时，尝试获取当前内存占用，并存储到自身。若获取不到，则假设当前内存占用为0。 - 启动时，每隔100毫秒收取一次系统当前内存占用，然后更新最大值。 - 若找不到入口函数则直接退出；否则构造一个rupta::mir::analysis\_context::AnalysisContext。根据指定的PTA分析类型不同（CallSiteSensitive模式和Andersen模式）构造不同的分析器（分别是rupta::pta::context\_sensitive::ContextSensitivePTA和rupta::pta::andersen::AndersenPTA），然后调用它们各自的analyze方法。  前文提及的ContextSensitivePTA分析器结构体的本质是这样定义的：  pub type CallSiteSensitivePTA<'pta, 'tcx, 'compilation> = ContextSensitivePTA<'pta, 'tcx, 'compilation, KCallSiteSensitive>;  也就是说，为ContextSensitivePTA的策略（Strategy）泛型参数填入这个KCallSiteSensitive即可获得一个CallSiteSensitivePTA。上文提及，构造了一个CallSiteSensitivePTA之后马上调用了它的analyze方法。这个方法内容很简单，除了计时以外，就只做了三件事：initialize，propagate和计时结束后的finalize。  由于我们是想从输出的函数调用图入手，查看这个图中的信息是如何一步步被加入的，从而在加入信息的过程中补充加入更多信息，达到获取分析所需信息的目的，因此有必要从函数调用图的生成开始逆向寻找这个往调用图中加信息的过程。  首先通过查看输出的日志（PTA\_LOG=debug cargo pta ...）发现有一行这个：[2024-06-17T13:31:27Z INFO rupta::util::results\_dumper] Dumping call graph...。搜索可知这行INFO是在src/util/results\_dumper.rs文件中定义的dump\_results函数输出的。  知道了生成调用图的位置，我们可以继续往上追踪到src/pta/context\_sensitive.rs的ContextSensitivePTA::finalize方法中。这个方法只干两件事情：输出函数调用图，输出PTA分析统计结果。  继续往上追踪，发现这个finalize方法在impl PointerAnalysis for ContextSensitivePTA的analyze方法中被调用。后者的构成在上文中已经讨论完毕。因此，有必要分析调用图数据结构self.call\_graph是如何在initialize和propagate方法中被修改的了。  2.3 收集必要的信息  在开题报告中，我们说过我们希望我们的分析工具可以给出如下信息：  {  "crates": [  {  "crate\_name": ...,  "manifest\_path": ...  }, ...  ],  "callables": [  {  "belongs\_to\_crate\_idx": ...,  "source\_file\_path": ...,  "line\_number": ...  }, ...  ],  "calls": [  {  "caller\_idx": ...,  "callee\_idx": ...  }, ...  ] }  我们的总体思路是：   1. crate的信息，总体的信息可以用cargo metadata获取，某个函数所属的crate也可以用src/builder/fpag\_builder.rs的FuncPAGBuilder::new中的方法查询到。 2. callables的信息，在src/builder/fpag\_builder.rs中，也是利用FuncPAGBuilder::new中的方法收集完全了，只不过需要和上一步crate信息对上 3. calls中的信息来源也被解决，来自src/pta/context\_sensitive.rs中的ContextSensitivePTA::add\_call\_edge函数，它能知晓调用者和被调用者各自的DefId。   由于MIRAI能够输出函数所在的源代码文件路径，而Rupta没有这个机能，所以需要借鉴一下MIRAI是怎么做这件事情的。  根据之前的调研，MIRAI会收集一个数组，内部的元素长这样：(rustc\_span::Span, (DefId, DefId))，表示在Span中发生了第一个DefId函数调用第二个DefId函数的情况。结果发现可以这样获取函数调用发生的源代码路径：  // loc的类型就是rustc\_span::Span let source\_loc = loc.source\_callsite(); if let Ok(line\_and\_file) = source\_map.span\_to\_lines(source\_loc) {  // line\_and\_file的类型是FileLines  // pub struct FileLines {  // pub file: Lrc<SourceFile>,  // pub lines: Vec<LineInfo>,  //}  // 现在已经可以得知该语句的位置了。 }  由于我们关心的是函数定义发生的位置而不是调用，这里的代码撑死了只能给我们一些方向性的参考。而目前的主要矛盾是：这个rustc\_span::Span的信息到底是在哪一步中收集获得的呢？  经过简单的搜索，我们发现原来上述(rustc\_span::Span, (DefId, DefId))信息是通过CallGraph::CallGraph方法加入到调用图中的，而后者在call\_visitor.rs的第349行被调用了。我们马上直奔那里一探究竟。  CallVisitor::get\_function\_summary中调用了上述加入新调用关系的方法。解读该函数发现信息来源是这样构成的：  调用发生的位置loc来自于CallVisitor实例自身的bv.current\_span，其中bv是个BodyVisitor。  调用者的位置，即第一个DefId来自于CallVisitor实例自身的bv.def\_id。结合MIR的特性很容易明白，实际上MIR中的每个Body就是一个函数。因此bv.def\_id就是当前正在被分析的函数（即调用者caller）的DefId。  被调用者的位置，即第二个DefId来自于函数传入的参数。我们可以暂且不管这个东西。  于是，我们很好奇这个bv中的def\_id是怎么获得的呢？于是跳转到该结构体的定义中一看，原来它的DefId是从构造函数中传进来的，不是自己分析获得的。没事，看看谁调用了BodyVisitor::new呢？一搜索发现有两处：   * 一处在CallVisitor::create\_and\_cache\_function\_summary中，如果发现被调用者有MIR表示，就新建一个BodyVisitor去分析被调用者的函数调用情况去了。这里def\_id的来源很明了，就是被调用者的def\_id。 * 另一处在CrateVisitor::analyze\_body中，这儿的def\_id仍然是外界传进来的，搜索发现这个analyze\_body方法是在CrateVisitor::analyze\_some\_bodies方法中计算获得的，好家伙终于找到源头了！！   我们重点关注后者的DefId是怎么计算获得的。我们发现有几处不同的计算DefId的方法：   * 通过分析入口函数找到入口函数的DefId * // Get the entry function let entry\_fn\_def\_id = if let Some((def\_id, \_)) = self.tcx.entry\_fn(()) {  def\_id } else {  DefId::local(DefIndex::from\_u32(0)) }; * 这儿的self.tcx的类型是TyCtxt<'tcx>，其来源即为rustc\_driver::Callbacks中after\_analysis方法回调函数中，对其传入的参数queries经处理后调用enter方法时，传递给闭包的第一个参数，也就是说这个tcx是编译器给出的一手信息，未经过MIRAI二次处理。 * 通过遍历HIR的BodyOwners获取各个Body的DefId * for local\_def\_id in self.tcx.hir().body\_owners() {  let def\_id = local\_def\_id.to\_def\_id();  // -- snip --  self.analyze\_body(def\_id); }   至此，我们把如何获得一个函数的DefId的方法梳理完成了。总结起来，大致是如下流程：   1. 从回调函数after\_analysis的参数rustc\_interface::queries::Queries，调用其.global\_ctxt().unwrap().enter(|tcx| {...})方法。 2. 对那个闭包中的tcx，调用迭代器.hir().body\_owners()，每次迭代都能获得一个LocalDefId。 3. 最后使用LocalDefId::to\_def\_id()方法获得DefId。   于是，我们很好奇这个bv中的current\_span是怎么获得的呢？于是我们回归到BodyVisitor的定义中，尝试寻找对self.current\_span的赋值发生在哪里？  第一处赋值发生在BodyVisitor::new方法中，但这次赋值只是给它赋值了一个全0的默认值，没有任何意义：  return BodyVisitor {  // -- snip --  current\_span: rustc\_span::DUMMY\_SP,  // -- snip -- }  除此以外，还有一个BodyVisitor::reset\_visitor\_state方法会将self.current\_span重置为全0值。以上两个对self.current\_span的赋值都不是我们要找的东西。  实际上，真正能够更新这个值的代码在这两个地方：   * BodyVisitor::visit\_statement，它长这样 * fn visit\_statement(&mut self, location: mir::Location, statement: &mir::Statement<'tcx>) {  debug!("env {:?}", self.bv.current\_environment);  self.bv.current\_location = location;  let mir::Statement { kind, source\_info } = statement;  // 其中，source\_info的数据类型是 &rustc\_middle::mir::SourceInfo  self.bv.current\_span = source\_info.span;  // -- snip -- } * BodyVisitor::visit\_terminator，它长这样 * fn visit\_terminator(  &mut self,  location: mir::Location,  kind: &mir::TerminatorKind<'tcx>,  source\_info: mir::SourceInfo, ) {  debug!("env {:?}", self.bv.current\_environment);  self.bv.current\_location = location;  self.bv.current\_span = source\_info.span;  // -- snip -- }   我们以前者为例分析这个SourceInfo的信息是从哪里来的。由函数签名可知这个SourceInfo是从函数参数中的statement提取得来，于是我们想知道这个statement是从哪里来的。追踪visit\_statement方法可知其在visit\_basic\_block中被调用，而后者的运行逻辑大概是这样的：  pub fn visit\_basic\_block(  &mut self,  bb: mir::BasicBlock,  terminator\_state: &mut HashMap<mir::BasicBlock, Environment>, ) {  let mir::BasicBlockData {  ref statements,  ref terminator,  ..  } = &self.bv.mir[bb];  let mut location = bb.start\_location();  let terminator\_index = statements.len();   if !self.bv.check\_for\_errors {  while location.statement\_index < terminator\_index {  self.visit\_statement(location, &statements[location.statement\_index]);  check\_for\_early\_return!(self.bv);  location.statement\_index += 1;  }  // -- snip --  }  // -- snip -- }  用文字描述这个过程就是：  这个函数接收了一个基本块，bb: rustc\_middle::mir::BasicBlock，并利用之从self.bv.mir中索引到了该基本块的信息，其中就包含了该基本块中的所有语句组成的数组statements。  声明一个可变变量location，初始化为该基本块的起始位置。  利用上述变量进行索引，调用前文提及的self.visit\_statement遍历该基本块中的所有语句，方法就是statements[location.statement\_index]。  这就引出了另一个问题：self.bv.mir又是从哪里来的？通过阅读代码知道这个东西只在BodyVisitor的构造函数中发生过唯一一次赋值，而这个构造函数的唯一参数就是一个BodyVisitor，由此这个问题就转变为了：BodyVisitor的mir成员是从哪里来的？  这个问题在BodyVisitor::new中得到了解答，这个构造函数接收一个Body的DefId然后构造一个BodyVisitor实例，而这个Body的mir成员则是从tcx中获得的：  pub fn new(  crate\_visitor: &'analysis mut CrateVisitor<'compilation, 'tcx>,  def\_id: DefId,  // -- snip -- ) -> BodyVisitor<'analysis, 'compilation, 'tcx> {  let tcx = crate\_visitor.tcx;  // --snip --  let mir = if tcx.is\_const\_fn\_raw(def\_id) {  tcx.mir\_for\_ctfe(def\_id)  } else {  let def = rustc\_middle::ty::InstanceDef::Item(def\_id);  tcx.instance\_mir(def)  };  // --snip -- }  由此我们可以知道，BodyVisitor::mir可以通过给定一个TyCtxt和DefId唯一确定，其确定算法即为上述代码。虽然不知道它的实际含义，但是照猫画虎还是比较简单的。  还有一个问题没解决：这个bb又是怎么来的？这就必须追踪BlockVisitor::visit\_basic\_block的调用链了。经过搜索，发现是这样的：    其中左边那个check\_for\_errors分支的调用有两处，而且这两处都长成一个样子：  fixed\_point\_visitor.bv.check\_for\_errors(  &fixed\_point\_visitor.block\_indices,  &mut fixed\_point\_visitor.terminator\_state, );  显然装着一堆bb的block\_indices是从FixedPointerVisitor那边搞来的。  右边那个FixedPointVisitor::visit\_basic\_block的bb来源也是一样，最终都指向了FixedPointerVisitor的block\_indices成员。于是问题就转变成了：这个成员是在哪里赋值的？结果在FixedPointerVisitor::new中发现了端倪：这个构造函数接受一个BodyVisitor，并直接  let dominators = body\_visitor.mir.basic\_blocks.dominators(); let (block\_indices, loop\_anchors) = get\_sorted\_block\_indices(body\_visitor.mir, dominators);  后边那个函数只是对基本块做了一下拓扑排序而已，本质上bb的来源就是BodyVisitor::mir::basic\_blocks罢了。而BodyVisitor::mir的来源，上文已经分析过了。  归总一下，如何获得一条语句的Span信息：   * 首先获得函数的DefId。结合queries...enter(|tcx| {...})回调函数给的tcx参数，可以获得该函数的MIR，记为mir。 * 直接从mir.basic\_blocks获取该函数所包含的全部基本块。 * 对每一个基本块bb，利用mir[bb]获取其包含的语句数组statements，并对每个语句stmt调用let mir::Statement { kind, source\_info } = statement;解包获得source\_info信息。 * 最后，利用source\_info.span获得语句的位置。   进一步地，可以从Span信息获得源文件路径和在文件中的行号信息。  // loc的类型就是rustc\_span::Span let source\_loc = loc.source\_callsite(); if let Ok(line\_and\_file) = source\_map.span\_to\_lines(source\_loc) {  // line\_and\_file的类型是FileLines  // pub struct FileLines {  // pub file: Lrc<SourceFile>,  // pub lines: Vec<LineInfo>,  //}  // 现在已经可以得知该语句的位置了。 }  Rupta和MIRAI都没有非常仔细地收集有关Crate的信息，MIRAI的CrateVisitor也未能提供任何帮助。  我们的目标是：在浏览所有函数的时候，都得知道这个函数属于具体的哪个Crate，这个Crate的Cargo.toml文件在哪里（以此指代该Crate的路径）。  经过搜索，以下代码可以实现这样的功能，它位于src/builder/fpag\_builder.rs的FuncPAGBuilder::new函数中。  // 整一份当前上下文的拷贝。 let cur\_tcx = acx.tcx.clone(); // cur\_tcx: TyCtxt<'tcx> // 获取一些关于当前函数DefId和所属crate的信息 let def\_id\_of\_func = func\_ref.def\_id.clone(); let crate\_index\_num = def\_id\_of\_func.krate; // 有crate的名字，但是没有版本号 let crate\_name = cur\_tcx.crate\_name(crate\_index\_num); // 当前编译会话里能找到函数所在的文件的信息 let cur\_session = acx.tcx.sess; let source\_map = cur\_session.source\_map(); let span = cur\_tcx.def\_span(def\_id\_of\_func); let file = source\_map.lookup\_source\_file(span.lo()); // 找到了这个函数定义在哪个文件里头！！！！ let filename = file.name.clone(); // filename的类型是rustc\_span::FileName，它是个枚举。这里极大概率出现的是Real类型。 // Real类型也是个枚举，在此处最常见的两种Real枚举类型是Remapped和LocalPath。 // Real(Remapped { local\_path: Some("/home/endericedragon/.rustup/toolchains/nightly-2024-02-03-x86\_64-unknown-linux-gnu/lib/rustlib/src/rust/library/core/src/ops/range.rs"), virtual\_name: "/rustc/bf3c6c5bed498f41ad815641319a1ad9bcecb8e8/library/core/src/ops/range.rs" }) // Real(LocalPath("/home/endericedragon/playground/example\_crate/fastrand-2.1.0/src/lib.rs")) // 枚举的其他类型均定义于rustc\_span/src/lib.rs let file\_path = match filename {  FileName::Real(real\_file\_name) => match real\_file\_name {  RealFileName::LocalPath(path\_buf) => {  get\_cargo\_toml\_path\_from\_source\_file\_path\_buf(path\_buf)  }  RealFileName::Remapped {  local\_path: path\_buf\_optional,  virtual\_name: \_virtual\_path\_buf, // 我们不关心虚拟路径，直接弃用  } => {  if let Some(path\_buf) = path\_buf\_optional {  get\_cargo\_toml\_path\_from\_source\_file\_path\_buf(path\_buf)  } else {  String::from("Virtual")  }  }  },  \_ => String::from("Other"), }; println!("crate\_name: {}, crate path: {:?}", crate\_name, file\_path);  // -- snip -- /// 和真正的文件系统交互，从源代码文件逐层向上查找直至找到第一个Cargo.toml，以定位该Crate的路径。 fn get\_cargo\_toml\_path\_from\_source\_file\_path\_buf(file\_path: PathBuf) -> String {  let mut path = file\_path;  while let Some(parent) = path.parent() {  if parent.join("Cargo.toml").exists() {  return parent.to\_path\_buf().to\_string\_lossy().into();  }  path = parent.to\_path\_buf();  }   unreachable!() }  Rupta和MIRAI都提供了绘制函数调用图的功能，说明他们均有数据结构存储函数调用关系。我们抽丝剥茧，看看具体是怎么实现的。  首先是输出.dot文件的地方。    通过观察上述函数的源代码，不难发现有个表征函数调用图中“函数调用关系”的结构体CallGraphEdge，而函数调用图的边恰好就是用来体现函数之间调用关系的！  继续阅读，发现这个CallGraphEdge是对任意实现了::graph::call\_graph::CGCallsite trait的类型的简单包装。不过这不是最重要的，最重要的是我们发现了两个方法，这两个方法对于在调用图中增加一条边是有用的，即定义在CallGraph上的：   * pub fn get\_callees(&self, callsite: &S) -> HashSet<F>方法 * pub fn add\_edge(&mut self, callsite: S, caller\_id: F, callee\_id: F) -> bool方法。   这儿的泛型参数详细为：F: CGFunction、S: CGCallSite。  其中，后者真正进行了边的创建、边编号EdgeIdx的申请和更新调用图中的信息等事务。根据后者的调用情况，我们可以给出结论：这些调用边并非在全部计算完成之后再加入调用图，而是一边计算一边加入调用图的。因此，想要知道函数的调用信息，有两个做法：   1. 修改CallGraphEdge，使得它能容纳我们想要的信息（caller、callee的唯一标识，并且尽可能直观） 2. 新增数据结构，记录我们想要的信息（比较麻烦，没必要，不推荐）   然而存在一个问题：函数调用图中，是用FuncId或者CSFuncId指代某个函数的。但是在MIR中，是用DefId指代某个函数。这中间一定存在某种转换关系，即我们想将FuncId转换为DefId。这要怎么做呢？  根据Rupta代码的启示，确认了可以这样做：  // 已知acx: AnalysisContext let func\_ref = acx.get\_function\_reference(func\_id); let func\_def\_id = func\_ref.def\_id;  利用上述原理，很容易改造::pta::context\_sensitive::ContextSensitivePTA::add\_call\_edge为如下的样子：  fn add\_call\_edge(&mut self, callsite: &Rc<CSCallSite>, callee: &CSFuncId) {  let caller = callsite.func;  if !self.call\_graph.add\_edge(callsite.into(), caller, \*callee) {  return;  }  // 利用acx把FuncId转换为DefId，这样函数的所有信息都能知道  let caller\_ref = self.acx.get\_function\_reference(caller.func\_id);  let caller\_def\_id = caller\_ref.def\_id;  let callee\_ref = self.acx.get\_function\_reference(callee.func\_id);  let callee\_def\_id = callee\_ref.def\_id;  println!("{:?} --> {:?}", caller\_def\_id, callee\_def\_id);  // 以下部分掌管比较细化的边，例如从实参指向形参的边，  // 和从返回值指向存储返回值的变量的有向边，  // 我们可以暂时不管。  let new\_inter\_proc\_edges = self.pag.add\_inter\_procedural\_edges(self.acx, callsite, \*callee);  for edge in new\_inter\_proc\_edges {  self.inter\_proc\_edges\_queue.push(edge);  } }  函数的调用双方在add\_call\_edge方法中全都知道了。但是函数调用发生在哪个文件的第几行中呢？  首先，注意到add\_call\_edge函数中传入的参数callsite: &Rc<CSCallSite>，其内有一个字段location: rustc\_middle::mir::Location，看名字应该是能知道这个call site的具体位置的。但是，如何利用这个信息进行查找呢？  根据搜索结果，办法如下：  // 这就是callsite参数的location字段 let call\_location = callsite.location; // 由于函数调用一定是发生在caller里，因此 // 需要获得caller的mir let caller\_mir = self.acx.tcx.optimized\_mir(caller\_def\_id); // 利用mir获得callsite的位置 // let call\_span = caller\_mir.source\_info(call\_location).span; // ! 之前的call\_span的获得方法有问题，现在好了 let call\_block = &caller\_mir.basic\_blocks[call\_location.block]; let call\_span = if call\_location.statement\_index < call\_block.statements.len() {  call\_block.statements[call\_location.statement\_index].source\_info.span } else {  call\_block.terminator().source\_info.span }; // 为获得行号信息，需要一个source\_map let source\_map = self.acx.tcx.sess.source\_map(); // match一下两种情况，Ok就是又有文件路径又有行号，Err就是只有文件路径（估计还是虚拟路径） match source\_map.lookup\_line(call\_span.lo()) {  Ok(source\_and\_line) => {  let source\_file = source\_and\_line.sf;  // 别忘记，这儿的行号和列号全是从0开始的  let line\_number = 1 + source\_and\_line.line;  println!(  "Callsite: {:?} calls {:?} at {:?} line {}",  caller\_ref.to\_string(),  callee\_ref.to\_string(),  source\_file.name,  line\_number  );  }  Err(\_) => () }  2.4 合理输出信息  先回顾一下我们需要的信息，以及它们分别分布在什么地方。   1. crate的信息，总体的信息可以用cargo metadata获取，某个函数所属的crate也可以用src/builder/fpag\_builder.rs的FuncPAGBuilder::new中的方法查询到。 2. callables的信息，在src/builder/fpag\_builder.rs中，也是利用FuncPAGBuilder::new中的方法收集完全了，只不过需要和上一步crate信息对上 3. calls中的信息来源也被解决，来自src/pta/context\_sensitive.rs中的ContextSensitivePTA::add\_call\_edge函数，它能知晓调用者和被调用者各自的DefId。   Rupta输出信息的总出口位于src/util/results\_dumper.rs的dump\_result函数中，函数签名如下：  pub fn dump\_results<P: PAGPath, F, S>(  acx: &AnalysisContext,  call\_graph: &CallGraph<F, S>,  pt\_data: &DiffPTDataTy,  pag: &PAG<P>, ) where  F: CGFunction + Into<FuncId>,  S: CGCallSite + Into<BaseCallSite>,  <P as PAGPath>::FuncTy: Ord + std::fmt::Debug + Into<FuncId> + Copy {  // --snip -- }  其函数签名中的acx很难不引起注意，它的类型是AnalysisContext，内含许多分析工作所必须的数据结构（例如TyCtxt）。它还存储了分析结果输出的路径，例如其中的：  if let Some(pts\_output) = &acx.analysis\_options.pts\_output {  info!("Dumping points-to results...");  dump\_ci\_pts(acx, pt\_data, pag, pts\_output);  // dump\_pts(pt\_data, pag, pts\_output); }  很显然，pts\_output就是运行rupta时传入的命令行参数中，指定的PTS输出路径。同时，函数还接收了一大堆参数（call\_graph, pt\_data）等。这些才是真正输出到调用图中去的信息。它们是谁给的呢？原来是ContextSensitivePTA：  impl<...> ContextSensitivePTA<...> {  pub fn finalize(&self) {  // dump call graph, points-to results  results\_dumper::dump\_results(self.acx, &self.call\_graph, &self.pt\_data, &self.pag);   // dump pta statistics  let pta\_stat = ContextSensitiveStat::new(self);  pta\_stat.dump\_stats();  } }  很显然，是在分析过程中，ContextSensitivePTA把分析结果存储于自身，然后在结束分析时调用了dump\_results进行分析结果的存储的。  那么，为了输出函数调用信息和所属crate信息，可以将上述信息直接放进AnalysisContext中。由于Rupta的几乎每个分析有关的函数都会以一个AnalysisContext作为第一个参数，因此在这里存储结果是相对容易实现的。  于是，增添rupta的代码，最终改动情况如下：   1. 新建了模块info\_collector，在其中定义了CrateMetadata和FuncMetadata两个结构体，前者唯一标识一个crate，后者唯一标识一个函数。 2. 在AnalysisContext中新增了一个func\_metadata: HashSet<FuncMetadata>字段，存储FuncPAGBuilder计算获得的所有FuncMetadata。 3. FuncPAGBuilder::new中计算获得构造FuncMetadata所需的所有信息，构造后者并加入AnalysisContext.func\_metadata中。 4. src/info\_collector/mod.rs中，用serde给FuncMetadata等结构体实现了了Serialize trait。 5. src/util/results\_dumper.rs的dump\_results函数中，增加了输出func\_metadata的语句。   在存储及输出函数及其所属Crate的过程中，每个FuncMetadata都会存储一个CrateMetadata结构。然而，一个crate中大概率有不止一个函数，这意味着相同内容的CrateMetadata会在数个FuncMetadata中存储多次。这样显然十分浪费内存。  一种想法自然是：开个数组存CrateMetadata，而只在FuncMetadata中存储这个CrateMetadata在数组中的下标。但是这个数组同时需要具有去重的功能，因为不同函数可以属于同一个Crate。  基于上述需求，设想并了一个结合HashMap和Vec的新数据结构VecSet，它的定义长这样：  pub struct VecSet<T: Eq + Hash> {  // 真正存储数据的数组  data: Vec<Rc<T>>,  // 记录每个数据项在数组中的下标，用于去重  included: HashMap<Rc<T>, usize>, }  使用Rc<T>，可以有效避免同一份数据项存储两遍的问题。经过测试，使用Rc<T>的VecSet比未使用Rc<T>的朴素版本能节省将近一半的内存用量（1608KB 减小到 868KB）。  和输出函数信息类似，如法炮制：   1. 在info\_collector中定义CallSiteMetadata结构体，包含调用者和被调用者的DefId，以及调用发生所在的文件在文件系统中的路径、文件中的具体行号。同时为它实现std::hash::Hash trait和serde的Serialize trait。 2. 在AnalysisContext中新增了一个callsite\_metadata: HashSet<CallSiteMetadata>。 3. rupta::pta::context\_sensitive::ContextSensitivePTA::add\_call\_edge计算获得一次函数调用的具体信息，并以此构造CallSiteMetadata，加入acx中。 4. 在result\_dumpers中输出这些信息。   2.5 增添命令行参数  Rupta的命令行参数解析模块脱胎于Clap，因此只需找到定义Parser的部分，添加选项后再将信息存储入AnalysisOptions结构中即可。Parser是这样定义的：  let parser = Command::new("rupta")  .no\_binary\_name(true)  .override\_usage(RUPTA\_USAGE)  .version(version())  .arg(Arg::new("entry-func-name")  .long("entry-func")  .takes\_value(true)  .help("The name of entry function from which the pointer analysis begins."))  .arg(Arg::new("…")  只需添加一个.arg()方法的调用即可方便地添加命令行参数。笔者最终增添了如下命令行参数：  .arg(Arg::new("overall-metadata-output")              .long("dump-overall-metadata")              .takes\_value(true)              .help("Dump overall metadata in JSON format to the output file."))  并为AnaylsisOptions增添pub overall\_metadata\_output: Option<String>,  字段，将该命令行参数的信息存储其中。  至此，Rupta改造创新训练完毕。相关文档和代码详见github.com/Endericedragon/rupta。  **研究生签名：傅泽**  **2024 年11月18日** |