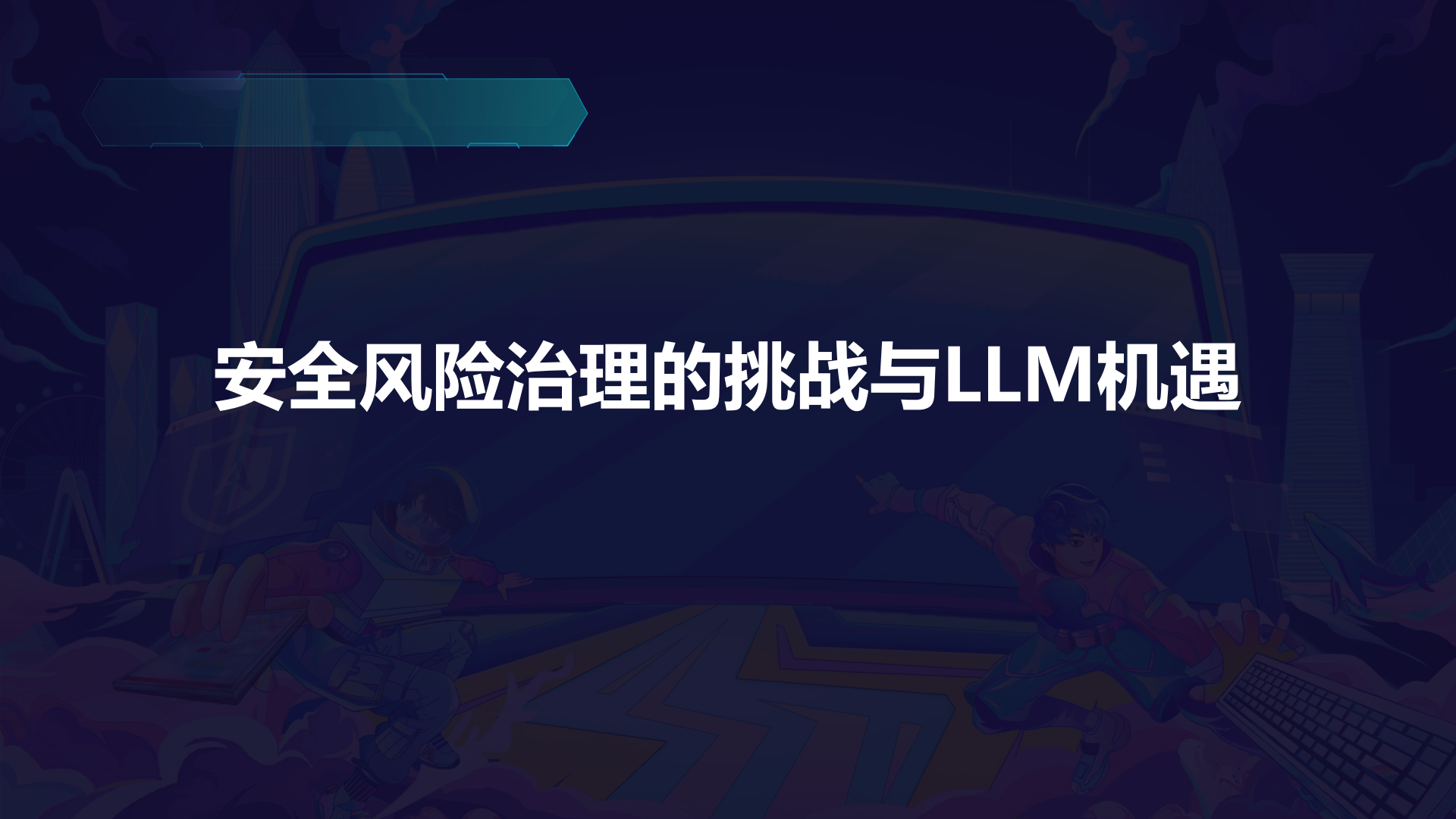
LLM与安全风险治理
端到端解决校验缺失风险代码

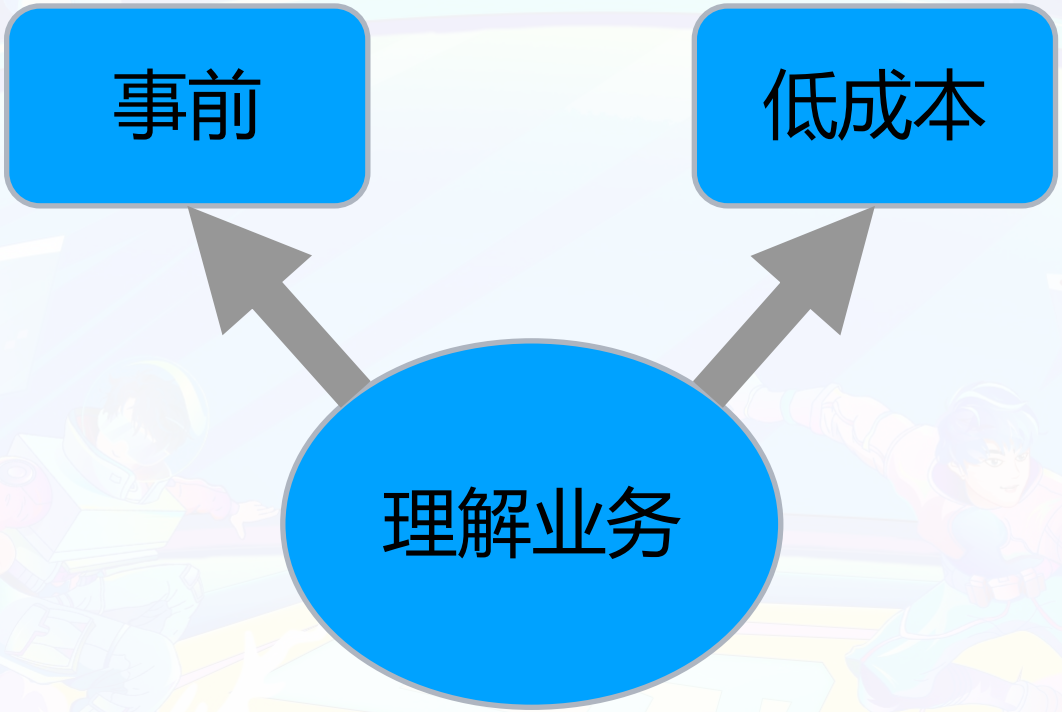
刘彦南
字节跳动无恒实验室安全工程师

目 录

安全风险治理的挑战与LLM机遇



安全风险治理的挑战

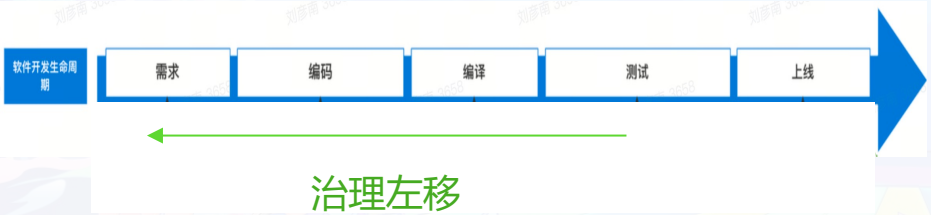


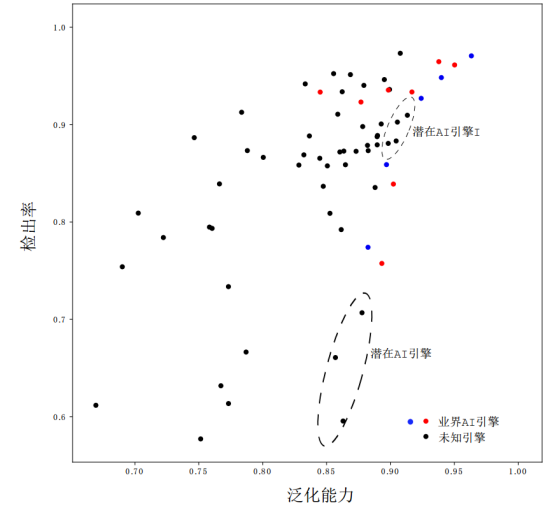
事前：如何比攻击者先行一步

无恒实验室

- 但木桶理论让“攻守效率”不平等，使用相同的方法无法实现目标

1. 风险治理时机左移: 尽早解决风险
2. AI for 安全: “泛化能力”、“检测未知风险”





?

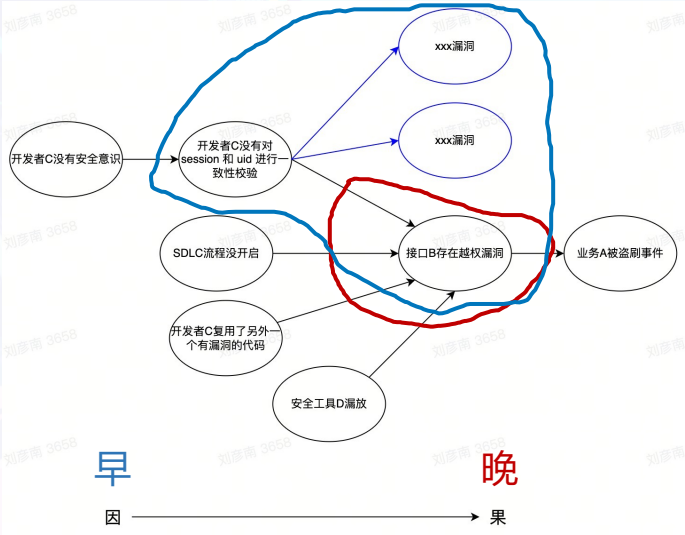
为什么能缓解木桶理论?

提升发现效率

风险形成逻辑：风险代码与漏洞

无恒实验室

- 安全风险其实是一个过程而非瞬时状态，暨多层脆弱性构成的因果链路最终导致了危害的发生。




```go
func GetOrder(ctx *gin.Context, req *OrderReq) (*OrderResp, error) {
    if req.OrderId == nil {
        logs.CtxError(ctx, "OrderId is nil")
        return nil, common.NewError(common.ReqError, "OrderId is nil")
    }

    orderResp, err := order.GetOrder(req.OrderId)
    if err != nil {
        logs.CtxError(ctx, "Get Order error")
        return nil, err
    }
    if orderResp == nil {
        logs.CtxError(ctx, "Order is nil")
        return nil, common.NewError(common.RespError, "Order is nil")
    }

    if orderResp.GetAccount().Id != GetUserId(ctx) {
        logs.CtxError(ctx, "The current user does not have permission to obtain
        return nil, common.NewError(common.AuthError, "The current user does not
    }
    return orderResp, nil
}
```

代码1: 缺少if-condition校验代码（17-20行）会导致水平越权漏洞

（为方便本文阐述构造的测试代码，不存在于模型数据集中）

左移的核心是解决更深层成因让治理风险更高效

缺失校验（风险代码）≠越权漏洞（漏洞）

治理深层风险代码的成本

- 修复成本成为瓶颈

1. 风险代码实例数 >> 漏洞数

及时且自动的修复风险来降低成本

   - 风险代码是漏洞的必要条件
   - 难以压缩：解决时机前移、确认 ‘可利用性’ 和 ‘危害’ 难度大

2. 现有面向漏洞的修复流程成本高昂

   - 漏洞让“修复”与“发现”两部分割裂

关键挑战：理解业务

- 业务场景定义了安全风险成因与危害，以‘校验缺失’风险代码为例
  1. 从业务代码中识别校验需求
     - 业务场景类型、函数功能语义
     - 变量结构定义、上下文函数定义
     - 支撑校验的Common Sense/客观关系知识
     - 值域/类型/模式校验需求
  2. 理解历史上不同校验需求对应的实现方式
     - 交易、评论、订单、UG
     - 组件、中间件、自定义代码、三方函数

理解业务逻辑和历史实现方式

Motivation: 通过LLM端到端解决风险代码

无恒实验室

理解业务逻辑和历史实现方式

从风险代码更加高效治理风险

及时且自动的修复风险来降低成本

- LLM的代码理解和分析能力：LLM模型本身掌握通用的代码语法、文本语义分析、甚至是简单数据流分析，同时具备多种代码理解能力。
- LLM对校验需求推断能力：LLM涌现的逻辑演绎推理能力，可结合历史代码从代码语义、注释语义、数据流等多角度信息合理推断校验需求。
- LLM的代码生成能力：多种LLM代码补全应用证明了其理解开发者意图的能力，可利用其代码生成能力实现校验代码生成补全、实现自动修复。

Show Case

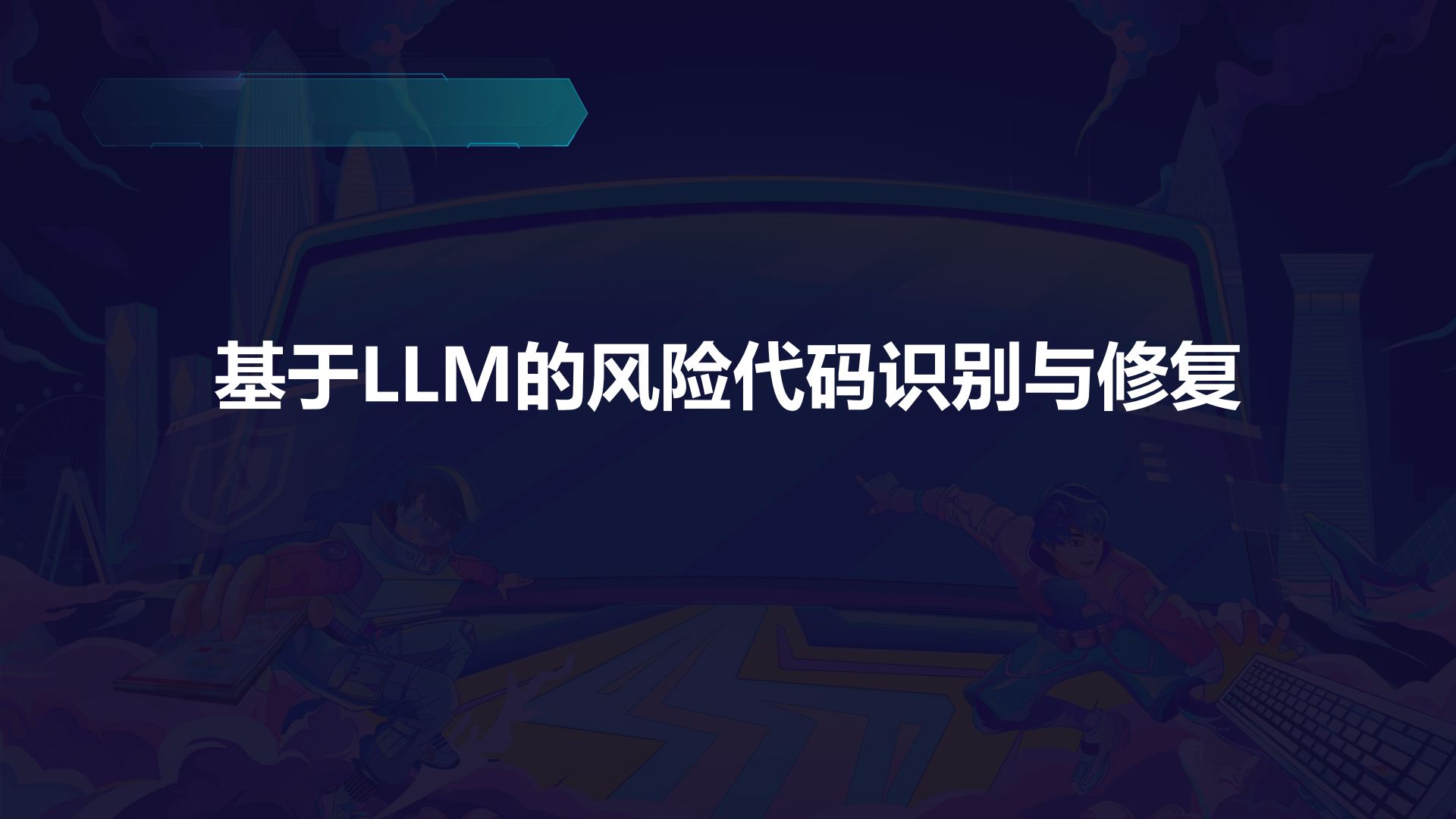
```
Q:<function>``func GetOrder(ctx *gin.Context, req *OrderReq) (*OrderResp, error) {
    if req.OrderId == nil {
        logs.CtxError(ctx, "OrderId is nil")
        return nil, common.NewError(common.ReqError, "OrderId is nil")
    }

    orderResp, err := order.GetOrder(req.OrderId)
    if err != nil {
        logs.CtxError(ctx, "Get Order error")
        return nil, err
    }
    if orderResp == nil {
        logs.CtxError(ctx, "Order is nil")
        return nil, common.NewError(common.RespError, "Order is nil")
    }
    return orderResp, nil
}``
<validate>

---- 以下为Risky Code LLM补全结果----

A: if orderResp.UserId != common.GetUserID(ctx) {
   logs.CtxError(ctx, "No permission to get order, OrderID:%v, UserID:%v", req.OrderId, c
   return nil, common.NewError(common.RespError, "No permission to get order")
}
```

- 将原校验语句移除之后（不在训练集中），模型能够生成语义类似的校验语句，原代码为`orderResp.GetAccount().Id`，模型生成的为`orderResp.UserId`

- 说明模型在“语义层面”进行补全，成功识别到代码来自“获取订单信息”场景，关联‘需要对订单的属主与当前登录用户进行校验’需求，并生成校验代码

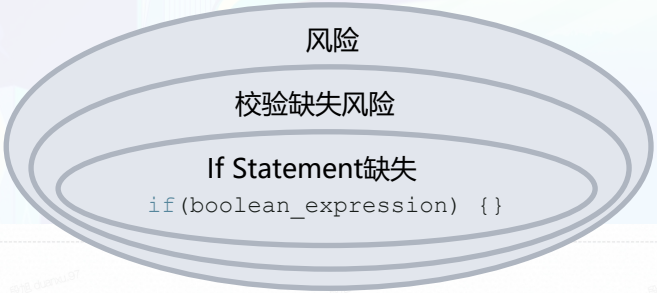
Risky Code LLM可在语义层面理解业务场景、分析校验需求并补全校验语句

基于LLM的风险代码识别与修复



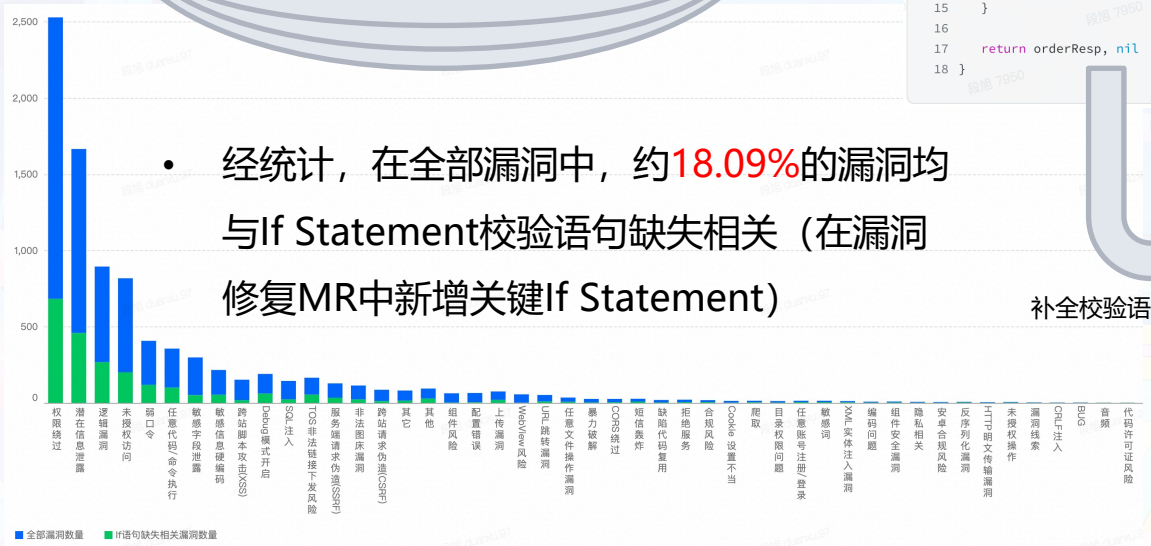
任务目标

- 任务目标：对代码中缺失的If语句进行补全




```go
func GetOrder(ctx *gin.Context, req *OrderReq) (*OrderResp, error) {
    if req.OrderId == nil {
        logs.CtxError(ctx, "OrderId is nil")
        return nil, common.NewError(common.ReqError, "OrderId is nil")
    }

    orderResp, err := order.GetOrder(req.OrderId)
    if err != nil {
        logs.CtxError(ctx, "Get Orc
        return nil, err
    }
    if orderResp == nil {
        logs.CtxError(ctx, "Order
        return nil, common.NewError
    }

    return orderResp, nil
}
```

没有对订单属主鉴权，
导致可以越权获取任意订单



- 经统计，在全部漏洞中，约18.09%的漏洞均与If Statement校验语句缺失相关（在漏洞修复MR中新增关键If Statement）

补全校验语句

```go
func GetOrder(ctx *gin.Context, req *OrderReq) (*OrderResp, error) {
    if req.OrderId == nil {
        logs.CtxError(ctx, "OrderId is nil")
        return nil, common.NewError(common.ReqError, "OrderId is nil")
    }

    orderResp, err := order.GetOrder(req.OrderId)
    if err != nil {
        logs.CtxError(ctx, "Get Order error")
        return nil, err
    }
    if orderResp == nil {
        logs.CtxError(ctx, "Order is nil")
        return nil, common.NewError(common.RespError, "Order is nil")
    }

    if orderResp.GetAccount().Id != GetUserId(ctx) {
        logs.CtxError(ctx, "The current user does not have permission to
        return nil, common.NewError(common.AuthError, "The current user
    }
    return orderResp, nil
}
```

补全鉴权校验，当请求订单不属于
当前登录用户时，阻止获取订单

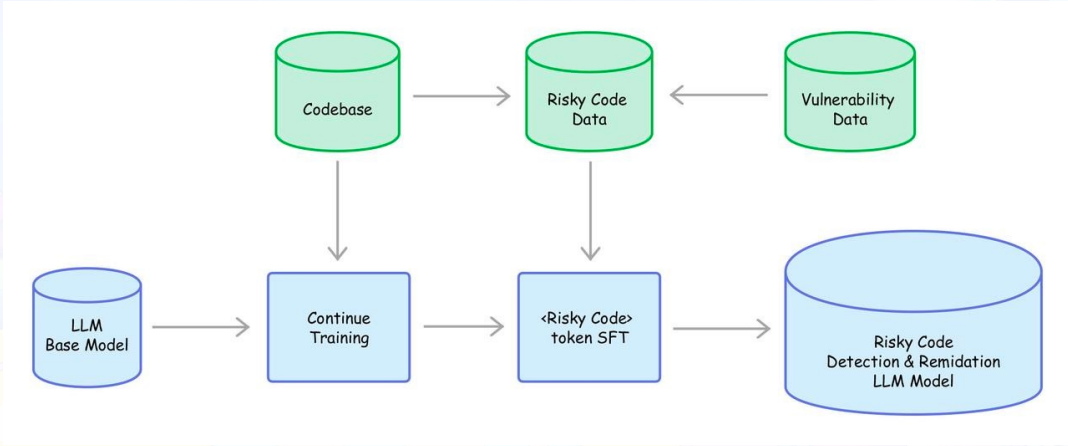
整体框架

1. Pretraining：使用具备基础代码能力的开源LLM模型作为Base Model。

2. Continue Training：结合公司Codebase代码，对Base Model进一步Continue Training，填补开源LLM模型数据分布与公司代码分布存在的差异。

3. Supervised Fine-Tune（SFT）：基于Codebase和历史漏洞数据，对模型进行指令微调，使模型了解“当前的任务/指令为补全校验语句”，以及“在何种代码分布下需要补全校验语句”。

已知前提：
风险代码形成逻辑与
业务场景关系紧密

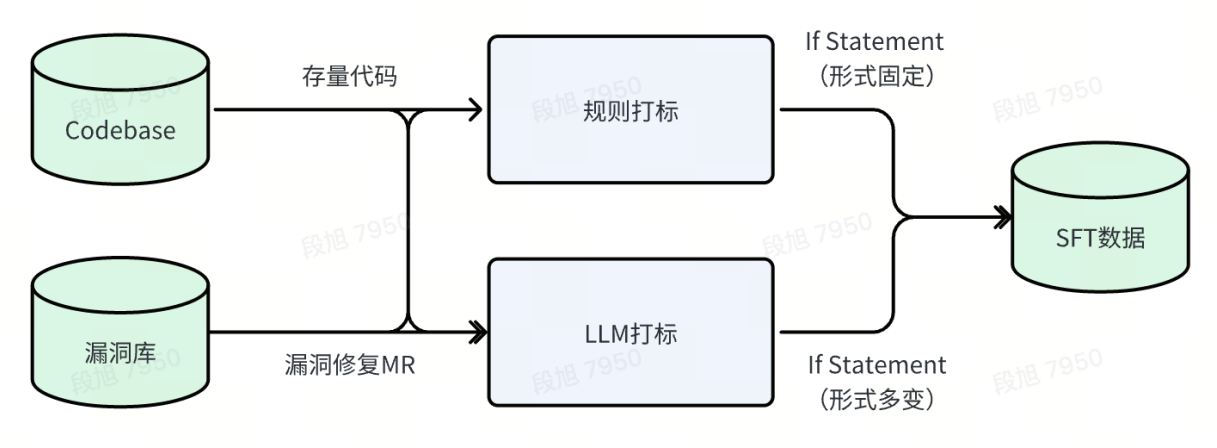


Supervised Fine-Tune

数据格式：采用了FIM（Fill in Middle）的训练模型，通过引入Special Token，将函数方法上下文输入到模型中，模型需要输出该方法缺失的关键校验，具体格式如下：

```
Model Input: <bos><risk_check_context>[context]<risk_check_validation>
Model Output: <risk_check_result>[completed check]<eos>
```

→ 计算Loss



规则打标：AST抽取出If Statement，通过简单规则过滤除关键的校验。打标出来的数据形式较为固定

LLM打标：部分关键校验形式多变，单纯规则无法识别（例如调鉴权语句再通过If判断结果，关键信息应是鉴权函数，而不是If语句），此类数据通过LLM识别。

Supervised Fine-Tune

- 在实践过程中，我们发现LLM有时会无中生有、强行编造一些结构体的成员变量，导致生成错误的校验语句。例如，如果`orderResp`中的属主是某个`Group`，同`Group`中的用户可以相互查看订单，那么对`orderResp.UserId`校验就是错误的。

- 此时可以在Prompt中补充`orderResp`的成员变量，从而生成正确的校验语句：

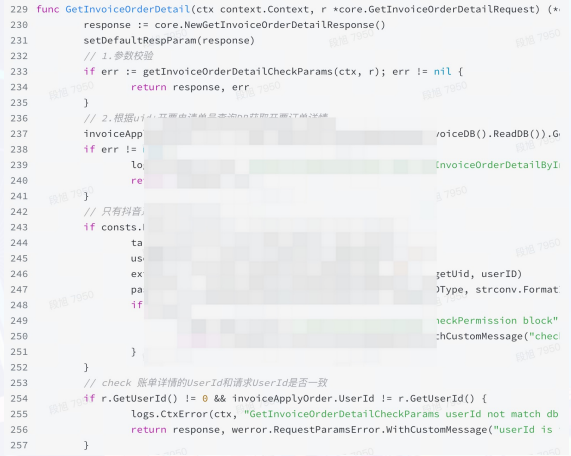
```
if orderResp.GroupId != common.GetGroupID(ctx)
```

```
Q:<context>``type OrderResp struct {
    GroupId         int
    OrderId         int
    OrderStatus     int
    OrderContent    string
}``
<function>``func GetOrder(ctx *gin.Context, req *OrderReq) (*OrderResp, error) {
   if req.OrderId == nil {
      logs.CtxError(ctx, "OrderId is nil")
      return nil, common.NewError(common.ReqError, "OrderId is nil")
   }

   orderResp, err := order.GetOrder(req.OrderId)
   if err != nil {
      logs.CtxError(ctx, "Get Order error")
      return nil, err
   }
   if orderResp == nil {
      logs.CtxError(ctx, "Order is nil")
      return nil, common.NewError(common.RespError, "Order is nil")
   }
   return orderResp, nil
}``
<validate>

---- 以下为Risky Code LLM补全结果----

A: if orderResp.GroupId != common.GetGroupID(ctx) {
  logs.CtxError(ctx, "GroupId not match, req:%v, orderResp:%v", req, orderResp)
  return nil, common.NewError(common.RespError, "GroupId not match")
}
```

在Prompt内丰富上下文信息可缓解幻觉（hallucination）问题

Supervised Fine-Tune

[context]构成：给定待扫描函数，则该函数的[context]包含：该函数代码 + 该函数Callee + 该函数中对象的成员变量 + 成员方法，四部分拼接在一起



当前函数代码

Callee函数代码

成员方法

成员变量

实验结果与分析

实验结果

细粒度评价指标：

- BLEU：机器翻译领域的常见指标，衡量参考文本和生成文本在1-gram至n-gram的相似性；
- 加权BLEU：在BLEU基础上提高编程语言关键字的权重；
- AST相似性：衡量模型生成代码与原代码的语法相似性，即参考代码和生成代码中相同AST子树的比例；
- DFG相似性：衡量模型生成代码与原代码的语义相似性，即参考代码和生成代码中相同DFG边的比例；
- 校验变量的Jaccard相似度：IF语句布尔表达式中被校验变量的Jaccard相似度；

Codebleu：加权平均

- **参数量、Continue Training以及在Callee、成员变量、成员方法均能够带来有效提升**

模型	参数量	codebleu	bleu	加权bleu	AST相似性	校验变量Jaccard相似度
Customized	1.7b	0.49	0.45	0.45	0.58	0.40
Codegen	6b	0.51	0.48	0.50	0.56	0.53
Starcoder	15b	0.58	0.55	0.57	0.65	0.55

模型	训练方式	codebleu	bleu	加权bleu	AST相似性	校验变量Jaccard相似度
Codegen 6b	SFT	0.49	0.47	0.49	0.54	0.50
Codegen 6b	Continue Training - SFT	0.51	0.48	0.50	0.56	0.53

模型	Context构成	codebleu	bleu	加权bleu	语法匹配程度	校验变量Jaccard相似度
Codegen 6b	No Addtional Context	0.58	0.55	0.56	0.66	0.63
Codegen 6b	with Addtional Context	0.60	0.57	0.59	0.68	0.65

实验结果

无恒实验室

- **上传漏洞、SSRF等“校验形式单一”的传统web洞，及涉及到鉴权的逻辑漏洞具有较高的准确性**

漏洞类型	bleu	codebleu	加权N-gram	语法匹配程度	校验变量Jaccard相似度
服务端请求伪造(SSRF)	0.78	0.85	0.79	0.92	0.77
上传漏洞	0.78	0.85	0.78	0.93	0.79
跨站请求伪造(CSRF)	0.75	0.83	0.75	0.93	0.73
潜在信息泄露	0.78	0.83	0.78	0.91	0.75
未授权访问	0.77	0.85	0.77	0.93	0.76
逻辑漏洞	0.69	0.79	0.70	0.84	0.67
权限绕过	0.75	0.82	0.76	0.90	0.76
敏感字段泄露	0.71	0.79	0.72	0.85	0.70
SQL注入	0.68	0.76	0.68	0.81	0.71
跨站脚本攻击(XSS)	0.34	0.49	0.37	0.49	0.31
非法图床漏洞	0.37	0.52	0.39	0.54	0.33

测试数据来源：

历史漏洞数据：筛选历史漏洞修复MR中包含新增校验语句的漏洞。将Merge之前的代码作为`[context]`，将MR中新增的校验语句作为`[completed check]`；


Q&A


THANKS