Journal of Computer Applications
计算机应用,2024,44(7):2151 - 2159
ISSN 1001-9081
CODEN JYIIDU
2024-07-10
http://www.joca.cn

文章编号:1001-9081(2024)07-2151-09
DOI:10.11772/j.issn.1001-9081.2023070968

基于源码分析的自动化外部函数接口生成方法

孙　蒴[1],张　伟[1,2,3],冯温迪[1*],张俞炜[4]
(1.北京信息科技大学 计算机学院,北京 100101;
2.北京未来区块链与隐私计算高精尖中心(北京信息科技大学),北京 100101;
3.国家经济安全预警工程北京实验室(北京信息科技大学),北京 100101;　4.北京大学 计算机学院,北京 100871)
(*通信作者电子邮箱wendifeng@bistu.edu.cn)

摘　要:外部函数接口(FFI)是解决一种编程语言调用其他语言函数库的主要方法。针对使用FFI技术时需要大量人工编码的问题,提出自动化外部函数接口生成(AFIG)方法。该方法利用基于抽象语法树的源码逆向分析技术,从被封装的库文件中精准提取出用于描述函数接口信息的多语言融合的统一表示。基于此统一表示,不同平台的代码生成器可利用多语言转换规则矩阵,全自动化地生成不同平台的FFI相关代码。为解决FFI代码生成中的效率低下问题,设计了一种基于依赖分析的任务聚合策略,通过把存在依赖的任务聚合为新的任务,有效消除了FFI代码任务在并行下的阻塞与死锁,从而实现任务在多核系统下的可扩展与负载均衡。实验结果表明:与人工编码相比,AFIG方法减少了FFI开发中98.14%的开发编码量以及41.95%的测试编码量;与现有的SWIG(Simplified Wrapper and Interface Generator)方法相比,在同等任务下可减少61.27%的开发成本;且生成效率随着计算资源的增加呈线性增长。

关键词:外部函数接口;代码生成;依赖消除;并行处理;静态分析
中图分类号:TP311.56　　**文献标志码:**A

Automatic foreign function interface generation method based on source code analysis

SUN Shuo[1], ZHANG Wei[1,2,3], FENG Wendi[1*], ZHANG Yuwei[4]
(1. *Computer School, Beijing Information Science and Technology University, Beijing* 100101, *China*;
2. *Beijing Advanced Innovation Center for Future Blockchain and Privacy Computing (Beijing Information Science and Technology University), Beijing* 100101, *China*;
3. *Beijing Laboratory of National Economic Security Early-warning Engineering (Beijing Information Science and Technology University), Beijing* 100101, *China*;
4. *School of Computer Science, Peking University, Beijing* 100871, *China*)

Abstract: Foreign Function Interface (FFI) is a fundamental method to invoke interfaces provided in other programming languages. Focusing on huge amount of manual coding required when using FFI, an Automatic Foreign function Interface Generation (AFIG) method was proposed. The reverse source code analysis technique based on abstract syntax tree was employed by AFIG to accurately retrieve the multilingual intermediate representation from library binaries, in which function interface information was uniformly described. Based on the representation, the multilingual conversion rule matrix could be utilized by different platform code generators to automatically generate FFI codes for various platforms without handcrafting. To further reduce generation time usage, a dependency analysis-based task aggregation strategy was proposed, by which tasks with dependencies were consolidated as monolithic ones. Hence, blocking and deadlocks were efficiently eliminated, and load balancing and scalability on multi-core systems were achieved, accordingly. Experimental results indicate that AFIG achieves a reduction of 98.14% for FFI developing codes and 41.95% for testing codes compared to manual coding method; under the same task, AFIG further reduces development cost by 61.27% compared to SWIG (Simplified Wrapper and Interface Generator). And the code generation efficiency of AFIG increases linearly with the increase of computing resources.

Key words: Foreign Function Interface (FFI); code generation; dependency elimination; parallel processing; static analysis

收稿日期:2023-07-19;**修回日期:**2023-09-25;**录用日期:**2023-09-25。　**基金项目:**国家重点研发计划项目(2022YFC3320900);北京市教育委员会科研计划项目(KM202311232005);"北京未来区块链与隐私计算高精尖中心"和"国家经济安全预警工程北京实验室"资助。

作者简介:孙蒴(1999—),男,河北任丘人,硕士研究生,主要研究方向:操作系统内核、计算机网络;　张伟(1980—),男,山东临清人,教授,博士,主要研究方向:网络和数据安全、软硬件协同设计;　冯温迪(1994—),男,山东菏泽人,副教授,博士,主要研究方向:移动边缘计算、软件定义网络、网络功能虚拟化;　张俞炜(1994—),男,福建福州人,助理研究员,博士,CCF会员,主要研究方向:人工智能、智能软件测试。

0 引言

在互联网时代，软件开发迭代速度快，功能变更频繁[1]。为避免重复“造轮子”、提升开发效率，软件系统开发者常使用公共函数库来快速实现相关功能[2]。尽管函数库通常以应用编程接口（Application Programming Interface，API）的形式提供功能，但它无法保证提供与开发者所使用的编程语言相同的版本。为此，允许一种编程语言调用另一种编程语言的外部函数接口（Foreign Function Interface，FFI）技术[3]应运而生。在实际开发中，FFI技术常被用于帮助Java或Python等语言程序调用C语言版本的驱动程序[4]。然而，FFI在实际使用中却存在着重重挑战。这其中最主要的原因在于FFI的使用需要开发者手动编写一些FFI相关代码，其中包括：重写函数接口的转换代码、为使转换代码能够适配到软件系统的适配代码以及与这些代码相关的测试代码。手动编写这些代码极大增加了开发人员的工作量，且由于FFI相关代码有特定的语法格式要求，由此提升了使用FFI技术的门槛[5]。

为改善这些问题，现有工作提出了以通过辅助工具降低FFI相关代码编码难度，例如在FFI开发过程中提供静态语法检查[6]与错误追踪功能[7]从而为手动编码提供辅助。另一方面，外部函数接口生成器（Foreign Function Interface GENerator，FFIGEN）[8]技术应运而生。其中，Beazley[9]提出的SWIG（Simplified Wrapper and Interface Generator）采用半自动模式生成FFI代码并延续至今[10]，ctypesgen[11]可对脚本语言进行自动分析与生成，但因脚本语言的特殊性使其模型无法运用到高级语言上。现有FFIGEN通过生成FFI相关代码来减少人工编码量，但未能自动地协助各类语言的FFI开发。首先，受限于代码分析能力，现有FFIGEN无法抽取外部接口的参数性质（输入、输出或同时包含两者），因而开发人员需要相应地进行手动标注；其次，现有的FFIGEN仅能自动生成FFI转换代码，这些转换代码并不能够直接被软件系统使用，仍需要进一步适配；最后，现有的FFIGEN无法生成对应的测试代码，并且自动化测试的现有研究[12]因规则问题无法应用于FFI领域中。综上所述，因现有研究的自动化程度不足，从而业界通常只能采用手动编码的模式进行FFI开发[4,13]。

为此，本文提出基于源码分析的自动化外部函数接口生成（Automatic Foreign function Interface Generation，AFIG）方法。AFIG方法的核心包括基于抽象语法树的源码逆向分析系统、多语言转换规则矩阵，以及多语言的融合的统一表示。逆向分析系统利用多语言转换规则矩阵自动识别出外部接口中的函数信息、依赖关系和参数性质，并将它们转换为多语言融合的统一表示，然后将此统一表示转换成为目标FFI相关代码，此过程无需任何人工编码。此外，为进一步提升AFIG方法在处理大规模软件库中生成FFI相关代码的性能，本文提出了基于负载均衡与任务聚合的面向FFIGEN的代码生成加速算法。该算法消除了代码生成过程中的任务依赖关系，并实现了代码生成任务的细粒度划分。实验结果表明，该加速算法可使AFIG方法的代码生成性能随计算资源增加呈线性增长。

综上所述，本文的主要工作包括：

1）对FFI相关代码和现有研究的分析与总结，提出实现自动化生成FFI相关代码的AFIG方法。该方法解决了现有研究的自动化程度不足的问题，从而提升了需要外部函数调用的软件开发效率。

2）实现了以AFIG方法为核心的原型系统，并在该系统上进行实验。实验结果表明：在相同的数据集下，AFIG方法可减少FFI开发中98.14%的开发编码量和41.95%的测试编码量，而现有的SWIG方法仅能减少27.31%的开发编码量并且无法减少测试编码量。实验结果表明AFIG在减少人工成本方面有显著优势。

3）为解决AFIG方法在大规模数据量下的生成速度问题，提出了基于依赖关系分析的任务聚合与负载均衡策略，消除了FFI代码生成任务中的依赖关系，允许代码生成任务被细粒度划分和调度，通过对任务的并行加速，代码生成性能可随计算资源的增加呈线性增长。

1 背景知识

本章以图1为例，阐述FFI技术的相关知识及术语。

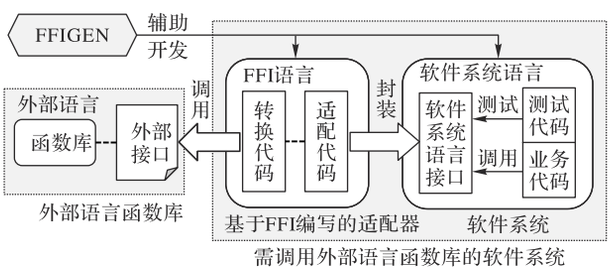


图1 基于FFI实现不同编程语言间的调用

Fig. 1 Invocation across programming languages based on FFI

1）外部语言：指开发软件所使用的编程语言之外的语言。

2）外部接口文件：简称外部接口，指使用外部语言编写的功能函数声明的载体文件，例如C语言中的头文件（.h）。

3）外部函数接口技术：指可实现一种编程语言调用另一种编程语言程序的技术。FFI技术通过在运行时动态加载外部函数库至内存，并提供与函数库的交互方法，从而实现两种语言之间的相互调用。不同语言提供了不同的FFI技术用于不同语言间交互，例如：JNA（Java Native Access）[14]允许Java程序调用C语言函数库，从而扩展了Java的功能；ctypes[15]允许Python程序调用C语言函数库。

4）FFI开发过程：指利用FFI技术，在一种语言（软件开发语言）需要调用以另一种语言提供的API接口时的开发过程。例如，使用Java语言调用C语言编写的驱动程序。在FFI开发过程中，软件系统开发者需要使用FFI技术的特殊语法来重写外部接口，重写后的接口与结构化数据的代码称为转换代码。由于转换代码采用FFI的语法规则而非软件系统的语法规则，开发者还需编写适配代码以帮助转换代码可被软件系统语言调用。此外，为了保障系统的稳定性，还需编写测试代码对软件系统语言接口进行测试。

5）外部函数接口生成器：指辅助生成FFI相关代码的自动化程序。FFIGEN技术通过将FFI相关规则构建成自动机来辅助开发者生成FFI代码，但是现有研究的自动化程度较低，无法生成适配代码和测试代码，从而导致FFIGEN在FFI开发中并未得到广泛应用，具体分析详见2.2节。

2 相关工作

2.1 基于FFI的项目开发研究现状

FFI技术作为不同编程语言间通信的桥梁，已在各种项

目和研究中得到了广泛应用[16]。现有针对FFI的项目开发研究主要集中在FFI技术的可靠性上[7,17-18],通过系统化论述使它更易被开发者使用。

此外,为解决编译器不支持FFI代码的编译过程检查从而导致编码错误的问题,Lee等[19-20]与Kondoh等[21]对常见错误进行分类,并实现了针对FFI技术的静态分析工具,以便开发者在开发过程中及时发现错误。此外,Li等[22]也以检查错误为关键进行研究,以减少FFI开发时的编码错误。而上述这些方法未能让开发者脱离FFI开发中编码量大且难以使用的困扰。

2.2　基于FFIGEN的项目开发研究现状

使用自动生成技术可以减少手工编写转换代码的工作量,从而提高开发效率和代码质量,然而,现有的FFIGEN尽管能够生成转换代码,但无法完全减少开发者在FFI开发中的工作量,甚至使问题复杂化。本文对现有研究进行了分析与总结,发现存在以下问题。

2.2.1　现有研究无法自动分析外部接口

SWIG[9]是一个持续维护与更新的FFIGEN[10],开发者需要手动标注或按特定规则编写一套输入表,以辅助SWIG理解外部接口中的信息,从而完成相关代码生成[23]。此外,如果外部接口包含复杂的结构型数据,这项工作将会更繁琐,且编写输入表的过程中极易引入错误。为了改善这个问题,JACAW[24]提供了对单个C语言头文件自动读取并自动生成Java转换代码的能力[25],然而,JACAW无法识别、推理出参数性质,从而导致生成准确度不足,并无法联合分析多个函数头文件,使它难以用于复杂情况下。ctypesgen[11]是Python专用的FFIGEN,但脚本语言中的参数不存在参数性质,导致该方法仅能运用在脚本语言上,无法应用于其他高级语言。

2.2.2　现有研究的输出无法被直接使用

图2详细展示了FFI开发过程中的编码细节,从外部接口到转换代码的过程被称为FFI开发过程,从转换代码到软件系统语言接口的过程被称为封装过程。现有的FFIGEN仅能生成转换代码,这些代码仍处于一种中间状态,经过复杂的封装流程,才能开发出软件系统语言接口。

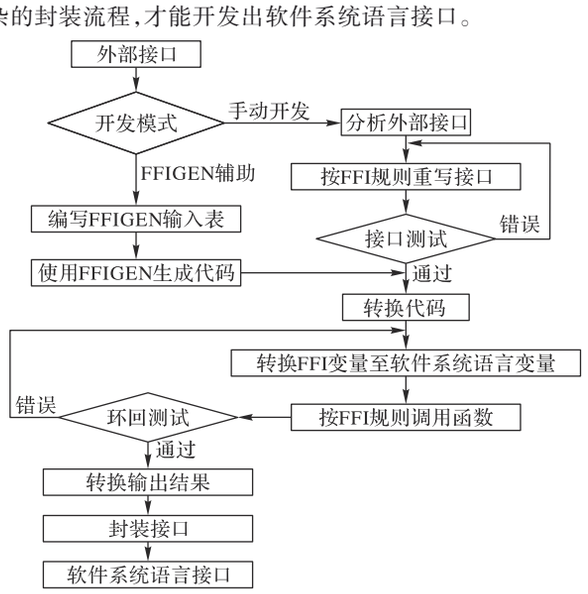


图2　外部接口适配至软件系统语言接口的开发流程

Fig. 2　Development process of adapting foreign interface to language interfaces of software system

在封装过程中,开发者需要仔细考虑所使用的数据类型是否合适,并确保类型之间的正确转换,才能完成图2中转换FFI变量至软件系统语言变量的步骤,并需要外部接口与软件系统间FFI通信的环回测试才可保障适配代码正常工作。编写适配代码不仅会增加开发负担,还可能会因人为编码引入错误,会导致严重的软件故障并增加测试成本。

2.2.3　现有研究未对代码进行测试

软件测试是保证软件质量和可靠性的关键步骤,它能有效地发现软件中存在的缺陷和错误[26]。根据现有研究,一些软件开发机构将超过40%的研发资源投入到软件测试中[27]。

FFIGEN最初是作为迁移其他平台遗留系统的工具而设计的[28],因此未将测试纳入FFIGEN的工作中。然而,为了确保在不同的操作系统、处理器架构和编译器下都能正常工作,需要对FFIGEN生成的代码进行兼容性测试和调整,将会引入大量的测试工作,特别是针对软件系统与外部接口间的环回测试,以确保参数能正常传递且不出现内存异常,而绝大多数FFIGEN无法自动或半自动生成测试代码[29]。因此,开发者通常使用其他测试工具或手工编写测试用例进行软件测试。

2.3　相关工作评估与对比

综上所述,本文对现有相关工作中的方法进行评估与对比,结果如表1所示,现有的FFIGEN在实现自动化和简化开发流程等方面仍存在一定的局限性,无法满足当前业界的需求,大部分功能仍需要人工辅助或人工开发来完成。

表1　AFIG与现有方法的对比

Tab. 1　Comparison between AFIG and existing methods

方法	接口分析	分析对象	自动化完成		
			编码	封装	测试
SWIG[9-10]	×	单个	○	×	△
JACAW[24]	△	单个	△	×	×
ctypesgen[15]	△	单个	△	×	×
AFIG	○	多个	○	○	△

注:“○”表示可用于实际开发;“△”表示需要人工辅助或准确度不足;“×”表示不支持。

如图3所示,本文把在第1章中提及的转换代码、适配代码的编写称为FFI开发工作,测试代码的编写与运行称为测试工作,相较于纯人工方法,FFIGEN在一定程度上减轻了软件系统开发者的工作负担,而考虑到FFI的代码框架复杂程度、开发流程的复杂性、测试编码及系统安全问题等诸多因素,设计出一套相较于现有研究更自动化的FFI相关代码生成方法尤为重要。

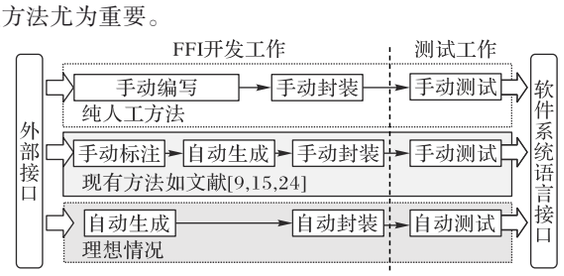


图3　FFI开发方式对比

Fig. 3　Comparison of FFI development methods

经分析,现有方法在自动化程度上的不足是因为无法完成自动、精确的接口分析,即无法识别出函数中参数性质,从而导致它无法进行转换代码、适配代码、测试代码的自动生

成;而本文提出的AFIG可解决这个问题,接下来的部分将介绍AFIG的设计和实现细节。

3 基于源码分析的AFIG方法设计

3.1 AFIG框架

AFIG的原型系统架构如图4所示,为实现自动化的外部函数接口生成技术,AFIG由代码抽取器、外部接口识别器、代码生成器和多平台测试器组成。通过输入需要FFI代码生成的外部接口与函数库,便可输出经测试的软件系统语言的软件开发工具包(Software Development Kit, SDK)。

代码抽取器负责分析并收集外部接口的所有函数和数据类型信息。通过反向编译器获取函数库源码中与外部接口相关的代码,并替换用户自定义类型以确保代码中的信息能被外部接口识别器所识别。外部接口识别器可通过分析代码抽取器整理的代码片段,得到多语言融合的统一表示,用以辅助代码生成器的生成工作。代码生成器依赖规则矩阵来生成指定软件系统语言的代码,这些代码按照规定的目录结构进行组织,然后交付给多平台测试器进行最后的测试。最终,AFIG会将指定软件系统语言的SDK与测试报告交付给软件系统开发者参考。

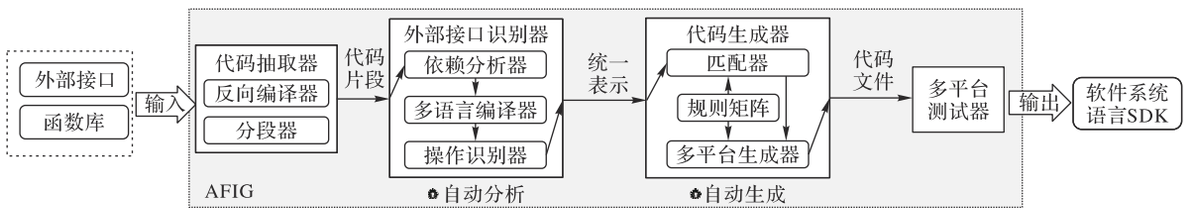


图4 AFIG原型系统架构

Fig. 4 System architecture of AFIG prototype

3.2 AFIG核心组件设计

3.2.1 外部接口识别器

通过对表1的分析发现,现有研究无法完全实现自动识别接口的原因在于没有手动标注参数性质的情况下,难以理清函数内部对参数的处理方式,从而无法选择相应的FFI变量类型与语法来建立映射。为解决该问题,外部接口识别器通过抽象语法树的关键字与上下文分析对源代码进行标注,并分析代码中的操作逻辑关系,由此得到外部接口中的全部信息。

因此,外部接口识别器的重点是分析出外部接口及源码中的接口信息、依赖关系和参数性质,在接收到代码抽取器中的源码片段后,外部接口识别器将进行外部接口信息的识别工作。在识别过程中,外部接口识别器通过自动标注源代码并分析关系语法树,将源码片段转换成多语言融合的统一表示,以便于代码生成器用统一的方式在不同语言间构建映射关系,从而提高软件可扩展性。本节将以图5中的代码作为输入示例详细阐述外部接口的识别流程。

语句标注	源代码	参数性质
s1	struct INFO {	
sa2	unsigned char Name[16]; };	
f3	voidwrite_to_file(char *filename, INFO *info);	
f4	INFOread_from_file(char *filename);	
fi5	voidwrite_to_file(char *filename,	
fi6	INFO *info) {	
7	INFO *fp;	in
o8	fp= fopen(filename, "w");	r
o9	fwrite(p, sizeof(INFO), 1, fp);	r
10	fclose(fp); }	in
fi11	intread_from_file(char *filename,	
fi12	INFO *info){	
13	FILE *fp;	in
14	int result;	in
o15	fp= fopen(filename, "rb");	r
o16	if (info == NULL){	r
r17	return 0; }	ret
o18	result =fread(info, sizeof(INFO), 1, fp);	w
19	fclose(fp);	in
r20	return result; }	ret

图5 源代码及标注结果

Fig. 5 Source codes and annotation results

AFIG关系语法树通过分析函数、参数和操作之间的依赖关系,构建出用于辅助函数接口解析的特殊树型结构,从而可转换至AFIG关系表格,以生成统一表示。具体方法包含以下步骤:

1)代码标注。图5为源码标注结构示例。首先,外部接口识别器会使用多语言编译器的规则对函数接口的参数进行语句标注,对源代码语句中的关键字进行识别,其中:s表示构体类型声明;sa表示结构体成员变量声明;f表示函数声明;fi表示函数参数;o表示针对函数参数的操作代码。语句标注的编号按照代码对应的行数依次递增。然后,外部接口识别器将根据函数接口操作类型分析源代码中的参数性质,将源代码中的特殊操作划分为内部变量操作(in)、读取(r)、写入(w)与返回值(ret)。

2)关系分析。外部接口识别器将图5中的标注信息组成图6所示的关系语法树,其中保存了函数与参数、结构体与成员变量、参数与操作之间的依赖和调用关系,通过这种方式,可推断出代码中的参数依赖、操作依赖和成员关系。

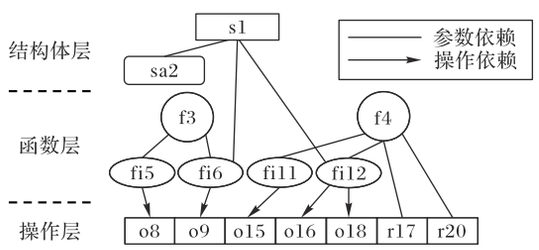


图6 关系语法树结构

Fig. 6 Structure of relational syntax tree

3)结果聚合。通过自顶向下的关系语法树分析,外部接口识别器提取了每个标准语句中的操作、变量名称和关键字等信息,并据此构建了如图7所示的关系表格。此分析确保了AFIG可以明确每个参数是输入、输出还是同时包含两者,进一步使外部接口识别器能够准确地标注接口。

最终,经过整理关系表格中的信息便可得到多语言融合的统一表示。这种统一表示的设计支撑了AFIG的可扩展性,通过实现统一表示转化为不同软件系统语言的转换规

则,即可动态扩展AFIG所支持的编程语言。

函数	参数	类型	相关操作	返回值
f3	fi5	char*	o8-r(读取);	无
	fi6	s1 *	o9-r(读取);	
f4	fi11	char *	o15-r(读取);	int
	fi12	s1 *	o16-r(读取),o18-w(写入);	
结构体	参数		类型	
s1	sa2		unsigned char[16]	

图7　关系表格示意图

Fig. 7　Schematic diagram of relationship table

3.2.2　规则矩阵

规则矩阵由相互关联的统一表示规则集、FFI规则集、单元测试规则集和编程语言规则集组成。

规则1与规则2为形式化描述函数接口转换规则的巴斯特范式(Backus-Naur Form, BNF)。规则1描述了由统一表示转换至JNA转换代码的规则,其中:type表示统一表示中的参数类型;identifier表示变量名称;io type表示参数性质;ε表示空串。规则1中第1)~10)行规则负责解析统一表示并匹配出重要数据字段。第11)~14)行规则定义出由统一表示生成转换代码的具体流程:将统一表示中参数列表的每一个参数的类型与性质传入jna函数(转换关系如表2所示,展示了不同的统一表示参数类型向JNA、Ctypes、cgo语言的对应规则)进行转换,最后加以拼接得到转换代码FFI cc。

表2　FFI规则集中基本变量类型的映射关系

Tab. 2　Mapping relationships of basic variable types in FFI rule set

统一表示	JNA	Ctypes	cgo
int	int	c_int	C. int
char	byte	c_wchar	C. char
short	short	c_short	C. short
long	long	c_long	C. long
long long	long	c_longlong	C. longlong
unsigned char	char	c_ubyte	C. uchar
unsigned short	short	c_ushort	C. ushort
unsigned int	int	c_uint	C. uint
unsigned long	long	c_ulong	C. ulong
int*	Pointer	c_int_p	*C. int
char*	String/byte[]	c_char_p	*C. char
void*	Pointer	c_void_p	unsafe. Poniter

规则1　由统一表示转换至FFI接口(JNA转换代码)。

核心语句　统一表示AFIG_LIIR;FFI接口声明FFI cc;

1) <AFIG_LIIR> ::= {<function declaration>}
2) <function declaration> ::= <type> <identifier> '(' <param list> ')' ';'
3) <type> ::= 'int' | 'char' | 'float' | 'int *' | 'void' | …
4) <param _list> ::= <param> {',' <param>} | ε
5) <param> ::= <type> <identifier> <io type>
6) <io type> ::= 'in' | 'out' | 'in&out'
7) <identifier> ::= <letter> {<letter or digit>}
8) <letter> ::= 'a' | 'b' | … | 'z' | 'A' | … | 'Z'
9) <letter or digit> ::= <letter> | <digit>
10) <digit> ::= '0' | '1' | … | '9'
11) <FFI cc> ::= 'public interface' <identifier>'{' {<FFI declaration>} '}'
12) <FFI declaration> ::= jna(<type>,<io type>)<identifier> '(' <FFI param list> ')' ';'
13) <FFI param list> ::= <FFI param>{',' <FFI param>} | ε
14) <FFI param> ::= <jna(<type>,<io type>)> <identifier>

规则2的目的是对规则1生成的转换代码进行分析并生成对应的适配代码,其中:FFI param list表示FFI接口参数列表;FFI declaration表示FFI接口声明;identifier表示标识符规则。转换代码中的参数由JNA类型构成的,通过以下3步完成适配代码:1)通过函数java_type生成出将Java变量转换至JNA变量的语句param conversion;2)通过函数invoke生成将JNA变量传入转换代码的语句FFI call;3)通过函数construct生成将转换代码的输出结果变为Java变量的语句output conversion。

规则2　FFI接口(JNA转换代码)封装至Java适配代码。

核心语句　适配代码(Java语言函数) function;将输入参数转换至FFI接口参数的代码语句param conversion;调用FFI接口的代码语句FFI call;将FFI接口输出转换为Java类型的代码语句output conversion;

1) <function> ::= < header> '{'<param conversion> ';' <FFI call> ';'<output conversion> ';' 'return' <identifier> ';' '}'
2) <header> ::= <access modifier> <return type><identifier> '(' <params> ')'
3) <access modifier> ::= 'public' | 'private' | 'protected'
4) <return type> ::= <type> | 'void'
5) <type> ::= 'int' | 'float' | 'double' | 'char' | 'String' | …
6) <jna type> ::= 'int' | 'byte' | 'byte[]' | 'Pointer' | …
7) <params> ::= <java_type(<FFI param list>)> | ε
8) <FFI call> ::= <identifier> '=' invoke(<FFI declaration>,<identifier>) ';'
9) <param conversion> ::= {<jna type> <identifier> '=' construct(<jna type>,<params>) ';'}
10) <output conversion>::= {<type> <identifier> '=' construct(<type>,<params>) ';'}

3.2.3　代码生成器

图8展示了代码生成器内部工作机制。代码生成器以外部接口识别器生成的统一表示作为输入,通过与规则矩阵中4种规则集进行匹配与分析,最终输出软件系统语言的转换代码、适配代码与测试代码,并打包为SDK。其中:统一表示规则集定义了统一表示与FFI代码的转换关系;FFI规则集定义转换代码如何转换为软件系统语言代码;单元测试框架与编程语言规则集定义了软件系统语言的编码格式。

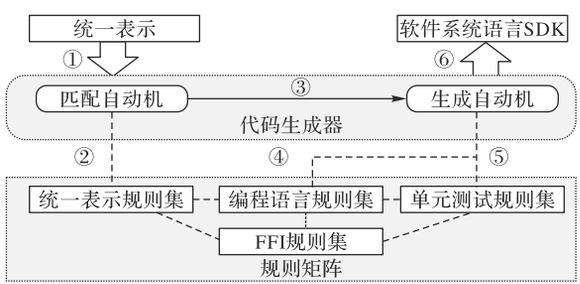


图8　代码生成器工作原理

Fig. 8　Working principle of AFIG code generator

代码生成器的运行过程可分为以下6步:

第1步:代码生成器接收由外部接口识别器传入的统一表示。

第2步:代码生成器根据预设的软件系统语言调用自动机匹配统一表示中的变量类型,并使用规则1和表2示例的

映射关系对统一表示进行标注，例如将C语言中char *参数类型标注为Java调用C语言的JNA技术中的String类型。

第3步：将标注结果传入至辅助生成代码的自动机中。至此，基于自动机的匹配工作结束，生成器将统一表示转化为源语言到目标软件系统语言的FFI标注片段。匹配工作结束后，代码生成器将依据规则2所示的生成规则进行软件系统语言代码的生成工作。

第4步：代码生成器将标注后的统一表示进行解析，并将其中的函数、参数替换为指定的FFI框架的编码格式，从而生成转换代码。接着，代码生成器将联合FFI与编程语言规则集来生成适配代码，用于封装转换代码中的FFI语法，以便开发者可在不了解FFI的情况下直接调用外部接口。

第5步：在生成全部业务代码后，生成器还需要使用单元测试框架的语法规则构建测试代码，以对适配代码与转换代码进行部分测试，确保开发者在使用时不会出现FFI的调用异常，如内存或是参数转换错误。

第6步：将转换代码、适配代码与测试代码打包为软件系统语言的软件开发包，并传入多平台测试器进行测试工作。

多平台测试器将根据软件系统语言平台的类型，在相应的环境下对接口进行测试工作，并将测试报告返回给软件系统开发者，减少了FFI开发与测试过程中绝大部分的工作量。

3.3 基于依赖关系分析的任务聚合与负载均衡策略

AFIG以全自动化的方式扩展了FFI相关代码生成的能力，但随着需要分析的外部函数库源代码量的增加，AFIG的分析、匹配与生成工作量也会随之增多。因此，需要充分利用多核处理器的优势进行并行任务处理。然而，由于源代码之间存在依赖，如何有效在多个核心上分配代码生成任务是自动化FFI代码生成过程中的一个全新的挑战，这对于提升AFIG的性能具有至关重要的影响，可进一步支持开发者的多次迭代和调试过程，以减少开发者的等待时间。

传统的多核并行计算通常采用静态的任务划分方式，将任务均匀地分配给计算核心。然而，在存在数据依赖关系的任务中，直接将任务顺序交付给不同计算核心可能导致依赖死锁或等待问题，进而降低运行效率。如图9所示，为解决此问题，本文提出基于依赖关系分析的任务聚合与负载均衡策略，简称为任务聚合策略。该策略消除了AFIG中代码生成任务的依赖关系，从而允许调度器对代码生成任务进行细粒度划分及调度。

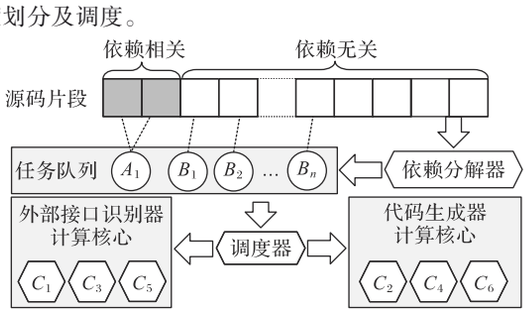


图9 基于依赖关系分析的任务聚合加速方法

Fig. 9 Dependency analysis-based task aggregation acceleration method

如图9与算法1所示，基于依赖分析的任务聚合策略算法的具体逻辑如下：核心节点通过依赖分解器对源代码中的参数与结构体间的依赖进行预分析，在依赖分析过程中，根据依赖关系及顺序将代码中依赖相关的任务存储至依赖相关任务集合A中，将不含依赖关系的代码片段保存至依赖无关集合B中，其中，集合A中的单个任务包含需要顺序分析且含有依赖关系的代码片段，而不同任务之间不存在依赖关系。

调度器将任务集合队列中的任务独立分配到不同的外部接口识别器计算核心，以生成统一表示。生成后的结果将传递给代码生成器计算核心进行处理。相比之下，传统的外部接口识别器与代码生成器的协作方式是单核心串行的，即代码生成器需要在外部接口识别器识别结束后才可进行生成工作；但是，通过采取基于依赖关系分析的任务聚合策略，可以将外部接口识别器生成的相互独立的统一表示分配给不同的代码生成器计算核心，以实现并行、流水线处理，从而提高代码生成器的计算效率。

算法1 基于依赖关系分析的任务聚合策略。

输入 等待分析的源代码集合S；

输出 依赖相关任务集合A；依赖无关任务集合B。

1) while S不为空 do
2) 初始化依赖相关任务集合A和依赖无关任务集合B
3) 对S中代码以结束语句分片，得到代码片段集P；K为代码片段集P中的片段数量
4) for 代码片段k=1, 2, …, K
5) if p_k中使用非基本类型 then
6) for 所有与p_k依赖相关片段p_x
7) if $p_x \in a_i$ then
8) 将$a_{length(A)+1} = \{p_x\}$插入$A$中
9) else if $p_x \in b_i$ then
10) 将$a_{length(A)+1} = \{p_x, p_k\}$插入$A$中
11) 从B中移除p_x
12) else if $p_x \in P$ then
13) 将$a_{length(A)+1} = \{p_x, p_k\}$顺序插入$A$中
14) 按依赖顺序排序，并合并A中p_x相关任务
15) else
16) 将$b_{length(B)+1} = \{p_k\}$插入$B$中
17) return A, B

此外，为确保多核的负载均衡，调度器对每个计算核心的输入代码数量和已处理完成的代码数量进行统计，通过实时反馈获得计算核心上的任务队列长度，保证不同计算核心的任务量接近。采用这种负载均衡下的多核加速策略与任务聚合模式，可利用并行优势减少处理时间显著提升了AFIG的代码生成速度。具体性能结果详见4.4节。

4 实验与结果分析

4.1 实验设置

为了评估本文方法的有效性，对AFIG原型系统进行测试，并对从C语言适配至Java 1.8、Python 3.7和Golang 1.19的3种编程语言的项目生成进行了全面的研究和分析。

在评估和测试AFIG原型时，本文使用了C语言的GM-T 0018-2012密码设备应用接口规范(Security Device Function, SDF)[30]接口与光学字符识别(Optical Character Recognition, OCR)引擎Tesseract 5.3.1接口作为需要适配至其他语言的外部接口。接口的信息如表3所示，经参考文献[31]与源代码总结出数据集的函数接口操作可覆盖函数接口基本操作类型。

表3　数据集信息统计

Tab. 3　Dataset information statistics

外部接口	结构体数	宏与用户定义类型行数	函数量
SDF	8	106	50
Tesseract	10	169	114

4.2　自动化程度分析

为证明AFIG具有高自动化的特征,本文通过2种方法比较了SWIG、AFIG和人工编写的项目完成能力:

1)比较人工编写和由AFIG生成的FFI相关代码完成度。如表4所示,本文使用AFIG对Tesseract和SDF项目进行FFI相关代码的生成,并将生成结果与人工编写的代码进行比较,使用完成的业务功能点作为开发完成度的评判标准,测试完成度将依据覆盖率的加权计算为评判标准,测试覆盖率数据详见表5,由AFIG自动生成的项目可完成绝大数开发工作并满足部分测试工作,而SWIG[10]不能生成适配代码与测试代码。在SDF接口的实验中,AFIG可减少FFI开发中98.14%的开发编码量以及41.95%的测试编码量,相较于SWIG,在同等任务下可额外降低61.27%的开发成本,极大程度上解决现有研究生成不全面的问题。

2)为了验证测试代码生成结果的有效性,使用JaCoCo(Java Code Coverage)工具[32]对Tess4J的人工编写项目与AFIG生成项目进行测试覆盖率比较。结果如表5所示,AFIG生成的测试代码接近人工测试水平,且在函数测试覆盖率上高于人工。

表4　不同开发方法的代码完成度比较　　单位:%

Tab. 4　Code completion comparison among different development methods　　unit: %

外部接口	开发完成度近似值		测试完成度近似值		
	SWIG	AFIG	人工	SWIG	AFIG
Tesseract	16.97	96.39	57.98	0	40.78
SDF	27.31	98.14	68.52	0	41.95

表5　测试覆盖率统计　　单位:%

Tab. 5　Test coverage statistics　　unit: %

项目名称	覆盖率			
	路径	语句	分支	函数
Tess4J(人工编写)	42.4	72.58	43.75	73.17
Tess4J(AFIG生成)	35.6	65.23	42.42	100.00

4.3　生产效益分析

AFIG的便利性如下:

1)开发者可以在完全不了解FFI技术、不分析外部接口的情况下使用AFIG完成FFI代码生成。根据AFIG的设计和实验结果,只要开发者提供了外部接口和软件系统环境信息,即可使用AFIG自动生成经过检验且功能完备的软件系统语言接口,以调用外部函数接口。

2)AFIG提高实际生产效率。实验中AFIG将C语言SDF接口向多种编程语言进行适配,结果如图10所示,向AFIG输入的外部接口的代码行(Line Of Code, LOC)为318的SDF接口,AFIG生成的Java、Python、Golang代码总计分别为4658、2633、2280行(LOC)。以标准程序员1天的有效代码行数为200行(LOC)进行统计,共可有效节约47.859天人的开发工作量。

此外,本文对AFIG与其他现有研究项目产生的生产效益进行比较,使用不同技术辅助SDF接口的FFI开发任务,实验结果如图11所示,以有效代码行数为每天200行(LOC)代码作为程序员的标准编码速度,AFIG可生成多种代码从而节约项目中的编码工作量,相较于SWIG,在同等任务下可额外降低61.27%的开发成本,AFIG可有效提高FFI项目的开发速度。

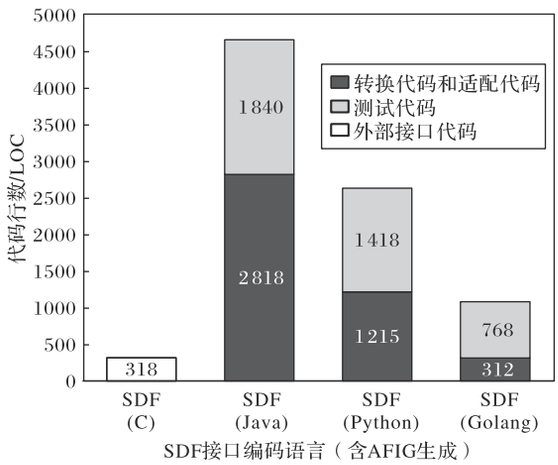


图10　AFIG的代码生成结果统计

Fig. 10　Statistics of AFIG code generation results

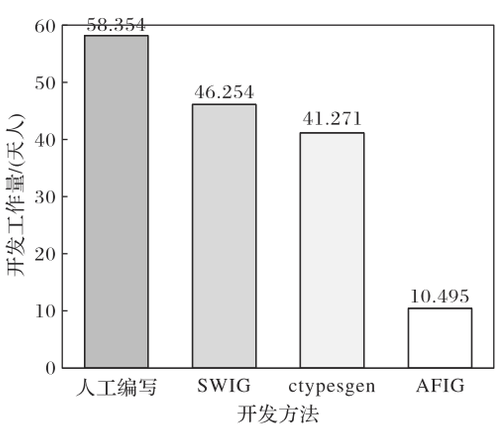


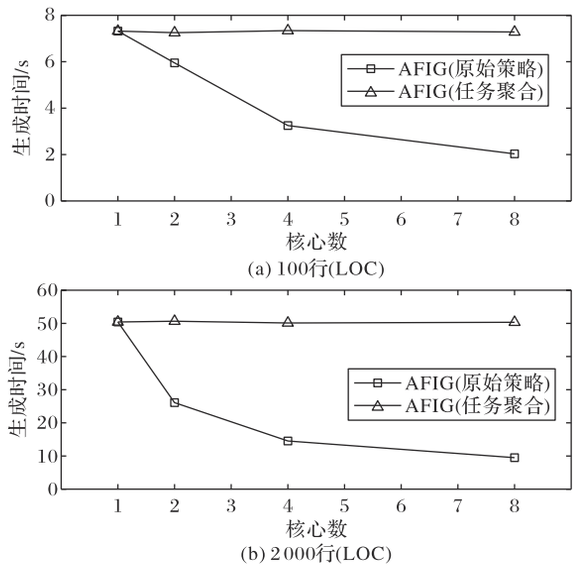
图11　同一任务下使用不同方法的开发工作量比较

Fig. 11　Development effort comparison among different methods under same task

4.4　任务聚合与负载均衡策略性能分析

采用Intel Core i7-9700K@3.60 GHz8核心处理器,内存为16 GB的Windows 10操作系统,对3.3节中提出的基于依赖关系分析的任务聚合与负载均衡策略进行性能评估。本实验将随机选取SDF与Tesseract接口中的代码片段作为数据集,在此基础上比较原始策略与使用任务聚合策略的代码生成速度。实验结果如图12所示,原始策略无法充分使用提供的多核心资源,导致生成时间持续不变,而使用任务聚合的情况下,代码生成效率可随核心数的增长而线性提升。

此外,为测量本实验的任务分配与负载均衡效益,在使用基于依赖关系分析的任务聚合与负载均衡策略的情况下,对不同代码量的数据集进行实验。实验结果如图13所示,在外部接口的代码总量持续增加的情况下,经过并行化加速后AFIG的生成过程均保持着高速线性运行,从而减少开发人员的等待时间。



(a) 100行(LOC)

(b) 2 000行(LOC)

图12 不同数据规模下原始策略和任务聚合与负载均衡策略的生成时间对比

Fig. 12 Generation time comparison of original strategy and task aggregation with load balancing strategy across various scales

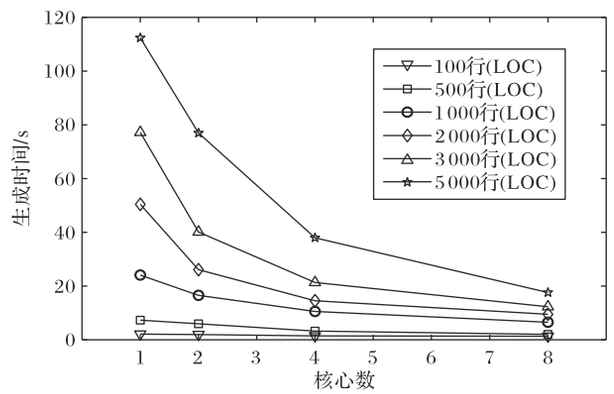


图13 任务聚合与负载均衡策略在不同核心数与不同代码行数(LOC)下的生成时间对比

Fig. 13 Generation time comparison of task aggregation with load balancing strategy with different cores and Line of Code (LOC)

5 结语

本文提出了基于源码分析的自动化外部函数接口生成(AFIG)方法,该方法利用基于抽象语法树的逆向分析器、多语言融合的统一表示以及多语言转换规则矩阵,实现全自动化的FFI相关代码的生成,改善了FFI技术中存在的人工编码工作量大、门槛高等问题。为验证AFIG的有效性,本文实现了AFIG原型系统并利用开源项目进行了验证,在SDF数据集下,相较于SWIG,AFIG能够减少高达98.14%的人工编码以及41.95%的测试工作,在同等任务下可额外降低61.27%的开发成本。同时,通过利用基于依赖关系分析的任务聚合与负载均衡策略,为FFI代码生成任务并行化处理提供了解决方案,使生成效率可随计算资源的增加而线性提升。

下一步的工作计划是在当前基础上,探索使用机器学习技术快速解析源代码,而不再依赖于自动机规则的匹配和标注。通过引入机器学习算法,训练模型自动学习源代码的结构、语义和特征,以更快的速度进行代码分析和生成。这样的方法将减少对手工规则的依赖,并能够更好地适应不同编程语言和项目的特点。在后续将致力于探索和实现这种基于机器学习的加速方法,并评估它们在AFIG中的性能和效果。

参考文献 (References)

[1] WYNN D C, ECKERT C M. Perspectives on iteration in design and development [J]. Research in Engineering Design, 2017, 28(2): 153-184.

[2] FOWLER M. Refactoring: Improving the Design of Existing Code [M]. [S. l.]: Addison-Wesley Professional, 2018: 328-332.

[3] YALLOP J, SHEETS D, MADHAVAPEDDY A. A modular foreign function interface [J]. Science of Computer Programming, 2018, 164: 82-97.

[4] YAN Y, GROSSMAN M, SARKAR V. JCUDA: a programmer-friendly interface for accelerating Java programs with CUDA [C]// Proceedings of the 15th International Euro-Par Conference on Parallel Processing. Berlin: Springer, 2009: 887-899.

[5] AMATO G, SCOZZARI F. JGMP: Java bindings and wrappers for the GMP library [J]. SoftwareX, 2023, 23: 101428.

[6] PARK J, LEE S, HONG J, et al. Static analysis of JNI programs via binary decompilation [J]. IEEE Transactions on Software Engineering, 2023, 49(5): 3089-3105.

[7] HWANG S, LEE S, KIM J, et al. JustGen: effective test generation for unspecified JNI behaviors on JVMs [C]// Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering. Piscataway: IEEE, 2021: 1708-1718.

[8] HANSEN L T. FFIGEN Manifesto and overview [EB/OL]. (1996-02-06) [2023-08-26]. https://www. ccs. neu. edu/home/lth/ffigen/manifesto. html.

[9] BEAZLEY D M. Simplified wrapper and interface generator [EB/OL]. (1996-02-26) [2023-08-26]. https://www. dabeaz. com/swig/SWIG_Doc1. pdf.

[10] WILLIAM S F, BETTS O, NADLINGER D, et al. SWIG (Simplified Wrapper and Interface Generator) [CP/OL]. (2023-08-18) [2023-08-25]. https://github. com/swig/swig.

[11] ROBERTSON A. Python wrapper generator for ctypes [CP/OL]. (2022-10-19) [2023-04-19]. https://pypi. org/project/ctypesgen/.

[12] 聂鹏,耿技,秦志光. 软件测试用例自动生成算法综述[J]. 计算机应用研究, 2012, 29(2): 401-405. (NIE P, GENG J, QIN Z G. Survey on automatic test case generation algorithms for software testing [J]. Application Research of Computers, 2012, 29(2): 401-405.)

[13] 顾荣,罗义力,仇伶玮,等. 跨语言用户态文件系统框架读写性能优化[J]. 电子学报, 2023, 51(6): 1590-1606. (GU R, LUO Y L, QIU L W, et al. Reading and writing performance optimization of cross-language FUSE framework [J]. Acta Electronica Sinica, 2023, 51(6): 1590-1606.)

[14] DOUBROVKINE D, WALL T, GREEN A, et al. Java native library: jna [CP/OL]. (2023-03-02) [2023-04-19]. https://github. com/java-native-access/jna.

[15] HELLER T. ctypeslib — useful additions to the ctypes FFI library [CP/OL]. (2021-12-17) [2023-04-19]. https://pypi. org/project/ctypeslib3/.

[16] LU H, JIN C J, HELU X H, et al. AutoD: intelligent blockchain application unpacking based on JNI layer deception call [J]. IEEE Network, 2021, 35(2): 215-221.

[17] GRICHI M, ABIDI M, JAAFAR F, et al. On the impact of inter-language dependencies in multilanguage systems empirical case study on Java Native Interface applications (JNI) [J]. IEEE

Transactions on Reliability, 2021, 70(1): 428-440.

[18] ABIDI M, RAHMAN M S, OPENJA M, et al. Are multi-language design smells fault-prone? an empirical study [J]. ACM Transactions on Software Engineering and Methodology, 2021, 30(3): No. 29.

[19] LEE S, LEE H, RYU S. Broadening horizons of multilingual static analysis: semantic summary extraction from C code for JNI program analysis [C]// Proceedings of the 2020 35th IEEE/ACM International Conference on Automated Software Engineering. Piscataway: IEEE, 2020: 127-137.

[20] LEE S. JNI program analysis with automatically extracted C semantic summary [C]// Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. New York: ACM, 2019: 448-451.

[21] KONDOH G, ONODERA T. Finding bugs in Java native interface programs [C]// Proceedings of the 2008 International Symposium on Software Testing and Analysis. New York: ACM, 2008: 109-118.

[22] LI S, TAN G. JET: exception checking in the Java native interface [J]. ACM SIGPLAN Notices, 2011, 46(10): 345-358.

[23] BEAZLEY D M. Automated scientific software scripting with SWIG [J]. Future Generation Computer Systems, 2003, 19(5): 599-609.

[24] HUANG Y, WALKER D W. JACAW: a Java-C automatic wrapper [R]. Wales, UK: Cardiff University, Department of Computer Science, 2002: 1-9.

[25] VAIRALE V S, HONWADKAR K N. Wrapper generator using Java Native interface [J]. International Journal of Computer Science and Information Technology, 2010, 2(2): 125-139.

[26] 单锦辉,姜瑛,孙萍. 软件测试研究进展[J]. 北京大学学报(自然科学版), 2005, 41(1): 134-145. (SHAN J H, JIANG Y, SUN P. Research progress in software testing [J]. Acta Scientiarum Naturalium Universitatis Pekinensis, 2005, 41(1): 134-145.)

[27] 张海藩,吕云翔. 软件工程[M]. 北京:人民邮电出版社, 2013: 149-175. (ZHANG H F, LYU Y X. Software Engineering [M]. Beijing: Posts & Telecom Press, 2013: 149-175.)

[28] LI J, ZHANG Z, YANG H. A grid oriented approach to reusing legacy code in ICENI framework [C]// Proceedings of the 2005 IEEE International Conference on Information Reuse and Integration. Piscataway: IEEE, 2005: 464-469.

[29] BUBAK M, KURZYNIEC D, LUSZCZEK P. Creating Java to native code interfaces with Janet [J]. Scientific Programming, 2001, 9(1): 39-50.

[30] 国家市场监督管理总局. GB/T 36322—2018,信息安全技术密码设备应用接口规范[S]. 北京:中国国家标准化管理委员会, 2018: 1-44. (State Administration for Market Regulation. GB/T 36322-2018, Information security technology: Cryptographic device application interface specifications [S]. Beijing: Standardization Administration of the People's Republic of China, 2018: 1-44.)

[31] 宣兆新,马旭. 商用密码模块接口统一调用技术研究[J]. 信息安全与通信保密, 2021(10): 75-81. (XUAN Z X, MA X. Study on interface unification of commercial cryptographic modules [J]. Information Security and Communications Privacy, 2021(10): 75-81.)

[32] HOFFMANN M R, JANICZAK B, MANDRIKOV E. JaCoCo Java code coverage library [CP/OL]. (2023-04-18) [2023-04-19]. https://github.com/jacoco/jacoco.

This work is partially supported by National Key R&D Program of China (2022YFC3320900); R&D Program of Beijing Municipal Education Commission (KM202311232005); Beijing Advanced Innovation Center for Future Blockchain and Privacy Computing, and Beijing Laboratory of National Economic Security Early-warning Engineering.

SUN Shuo, born in 1999, M. S. candidate. His research interests include operating system kernels, computer networks.

ZHANG Wei, born in 1980, Ph. D., professor. His research interests include network and data security, hardware-software co-design.

FENG Wendi, born in 1994, Ph. D., associate professor. His research interests include mobile edge computing, software-defined networks, network function virtualization.

ZHANG Yuwei, born in 1994, Ph. D., assistant research fellow. His research interests include artificial intelligence, intelligent software testing.