DOI: 10.15514/ISPRAS-2020-32(3)-1

Архитектура системы дедуктивной верификации машинного кода

[4] *И.В. Гладышев, ORCID: 0000-0002-9922-4076 <ilya.v.gladyshev@gmail.com>*
[1,2,3,4] *А.С. Камкин, ORCID: 0000-0001-6374-8575 <kamkin@ispras.ru>*
[1] *А.М. Коцыняк, ORCID: 0000-0003-3499-4368 <kotsynyak@ispras.ru>*
[1,4] *П.А. Путро, ORCID: 0000-0001-9540-8321 <pavel.putro@ispras.ru>*
[1,2,3,4] *А.В. Хорошилов, ORCID: 0000-0002-6512-4632 <khoroshilov@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *Московский государственный университет имени М.В. Ломоносова, 119991, Россия, Москва, Ленинские горы, д. 1*
[3] *Московский физико-технический институт 141700, Россия, Московская область, г. Долгопрудный, Институтский пер., 9*
[4] *Национальный исследовательский университет «Высшая школа экономики», 101000, Россия, г. Москва, ул. Мясницкая, д. 20*

Аннотация. В последние годы ИСП РАН разрабатывает систему дедуктивной верификации машинного (бинарного) кода. Мотивация понятна: современные компиляторы, такие как GCC и Clang/LLVM, не застрахованы от ошибок; тем самым, проверка корректности сгенерированного кода (хотя бы для компонентов с повышенными требованиями к надежности и безопасности) не является лишней. Ключевая особенность предлагаемого подхода состоит в возможности переиспользования формальных спецификаций (пред- и постусловий, инвариантов циклов, лемм и т.п.) уровня исходного кода для верификации машинного кода. Инструмент основан на формальной спецификации системы команд и обеспечивает высокий уровень автоматизации: он дизассемблирует машинный код, извлекая его семантику, адаптирует высокоуровневые спецификации для машинного кода и генерирует условия верификации. Система использует ряд сторонних компонентов, включая анализатор исходного кода (Frama-C), анализатор машинного кода (MicroTESK) и SMT-решатель (CVC4). Модульная архитектура позволяет заменять один компонент другим при изменении формата входных данных или используемой техники верификации. В работе рассматривается архитектура инструмента, описывается наша реализация и демонстрируется пример верификации библиотечной функции memset.

Ключевые слова: формальные методы; дедуктивная верификация; анализ бинарного кода; проверка эквивалентности; архитектура системы команд; машинный код; тестирование компиляторов.

Для цитирования: Гладышев И.В., Камкин А.С., Коцыняк А.М., Путро П.А., Хорошилов А.В. Архитектура системы дедуктивной верификации машинного кода. Труды ИСП РАН, том 32, вып. 3, 2020 г., стр. 7-20. DOI: 10.15514/ISPRAS–2020–32(3)–1

Благодарности. Работа поддержана грантом Минобрнауки РФ RFMEFI60719X0295.

Architecture of a Machine Code Deductive Verification System

[4] *I.V. Gladyshev, ORCID: 0000-0002-9922-4076 <ilya.v.gladyshev@gmail.com>*
[1,2,3,4] *A.S. Kamkin, ORCID: 0000-0001-6374-8575 <kamkin@ispras.ru>*
[1] *A.M. Kotsynyak, ORCID: 0000-0003-3499-4368 <kotsynyak@ispras.ru>*
[1,4] *P.A. Putro, ORCID: 0000-0001-9540-8321 <pavel.putro@ispras.ru>*
[1,2,3,4] *A.V. Khoroshilov, ORCID: 0000-0002-6512-4632 <khoroshilov@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia*
[3] *Moscow Institute of Physics and Technology (State University), 9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia*
[4] *National Research University Higher School of Economics, 20, Myasnitskaya st., Moscow, 101000, Russia*

Abstract. In recent years, ISP RAS has been developing a system for machine (binary) code deductive verification. The motivation is rather clear: modern compilers, such as GCC and Clang/LLVM, are not free of bugs; thereby, it is not superfluous (at least for safety- and security-critical components) to check the correctness of the generated code. The key feature of the suggested approach is the ability to reuse source-code-level formal specifications (pre- and postconditions, loop invariants, lemma functions, etc.) at the machine code level. The tool is highly automated: provided that the target instruction set is formalized, it disassembles the machine code, extracts its semantics, adapts the high-level specifications, and generates the verification conditions. The system utilizes a number of third-party components including a source code analyzer (Frama-C), a machine code analyzer (MicroTESK), and an SMT solver (CVC4). The modular design enables replacing one component with another when switching an input format and/or a verification engine. In this paper, we discuss the tool architecture, describe our implementation, and present a case study on verifying the memset C library function.

Keywords: formal methods; deductive verification; binary code analysis; equivalence checking; instruction set architecture; machine code; compiler testing.

For citation: Gladyshev I.V., Kamkin A.S., Kotsynyak A.M., Putro P.A., Khoroshilov A.V. Architecture of a Machine Code Deductive Verification System. Trudy ISP RAN/Proc. ISP RAS, vol. 32, issue 3, 2020. pp. 7-20 (in Russian). DOI: 10.15514/ISPRAS–2020–32(3)–1

Acknowledgements. The research was carried out with funding from the Ministry of Science and Higher Education of the Russian Federation (the project unique identifier is RFMEFI60719X0295).

1. Введение

Роль программного обеспечения (ПО) в критической информационной инфраструктуре постоянно растет. В результате сейчас крайне востребованы прикладные методы и инструменты обеспечения корректности наиболее ответственных компонентов ПО. Научным сообществом предложен ряд подходов: некоторые из них ограничиваются проверкой отсутствия в компоненте ошибок определенных типов (например, ошибок времени исполнения), в то время как другие пытаются доказать *полную корректность*, что означает, что все возможные вычисления компонента *завершаются* и удовлетворяют *программному контракту*, выраженному в форме *пред-* и *постусловий* на интерфейсы компонента. Для доказательства такого рода свойств применяют методы *дедуктивной верификации*.

Первые идеи таких методов появились в работах Флойда [1] и Хоара [2] еще в конце 1960-ых годов (индуктивные утверждения, аксиоматическая семантика и прочее). Несмотря на это, верификация промышленного ПО стала *реалистичной* совсем недавно [3-7]. Все известные инструменты дедуктивной верификации императивных программ следуют единому подходу [8]:

- *формально* определяется *семантика* всех операторов языка программирования;
- функциональные требования к компоненту формализуются в виде *пред-* и *постусловий* функций (или методов) на некотором *языке спецификации*;
- пользователем предоставляются дополнительные подсказки для инструмента, такие как *инварианты циклов*, *вспомогательный код* (*ghost code*) и *леммы*;
- на основе спецификаций и подсказок инструментом генерируются *условия верификации*, которые проверяются с помощью *решателя* (*solver*) или *интерактивной системы доказательства теорем* (*proof assistant*);
- доказательство всех условий верификации означает, что *все возможные вычисления* компонента удовлетворяют функциональным требованиям с учетом *предположений* (*гипотез*) о среде исполнения, средствах разработки и т.п.

Одно из предположений состоит в том, что машинный (бинарный) код, сгенерированный компилятором, соответствует семантике языка программирования. Очевидно, что эту гипотезу можно принять только для верифицированных компиляторов, например, CompCert [9], используемых в основном в исследовательских проектах. В индустрии же по-прежнему используют «тяжеловесные» оптимизирующие компиляторы, такие как GCC и Clang/LLVM. К сожалению, они слишком сложны для верификации, и ошибки в сгенерированном машинном коде не являются редкостью [10].

В качестве альтернативного подхода, не принимающего на веру корректность компилятора, мы предлагаем для каждой программы доказывать, что сгенерированный бинарный код удовлетворяет функциональным спецификациям, заданным для исходного кода. Идея выглядит привлекательной – гораздо проще проверить одно конкретное преобразование кода, чем компилятор целиком. Более того, это позволяет использовать агрессивные оптимизации, которые в целом небезопасны, но приемлемы для конкретного компонента и его контракта. В то же время есть много трудностей, которые нужно преодолеть:

- целевая *архитектура* (*система команд*) – регистры, память, режимы адресации и команды – должна быть *формализована* (иначе невозможно математически строго рассуждать о семантике машинного кода);
- *высокоуровневые спецификации* должны некоторым образом *адаптироваться* к бинарному коду (в частности, должно определяться соответствие между переменными в исходном коде и элементами памяти в машинном коде);
- подсказки для инструмента верификации, включая инварианты цикла, вспомогательный код и леммы, должны *переиспользоваться* на уровне бинарного кода (или должен существовать альтернативный способ их задания);
- инструмент должен проверять функциональные свойства получающегося бинарного кода при наличии произвольных *оптимизаций компилятора*.

Оставшаяся часть статьи организована следующим образом. В разд. 2 делается обзор работ, посвященных дедуктивной верификации программ на уровне бинарного кода. В разд. 3 описывается архитектура системы дедуктивной верификации машинного кода. В разд. 4 приведен пример использования предложенного подхода – библиотечная функция `memset`, компилируемая в систему команд RISC-V. Наконец, в разд. 5 делается заключение и обрисовываются направления дальнейших исследований.

2. Обзор литературы

В проекте Why3-AVR [11] с помощью платформы Why3 [12] верифицируются программы без ветвлений и циклов, разработанные на языке ассемблера микроконтроллера AVR. Система команд AVR представляется формально на языке WhyML – синтаксис позволяет описывать команды таким образом, чтобы ассемблерный код (после простого препроцессирования) был допустимым WhyML-текстом. Программист может аннотировать

код пред- и постусловиями и проверять его корректность с помощью внешних решателей или интерактивных систем доказательства теорем. Инструмент может быть полезен для низкоуровневой разработки, поскольку Why3 обладает богатыми возможностями для анализа и преобразования кода. Наш подход отличается: (1) он позволяет повторно использовать спецификации уровня исходного кода для верификации бинарного кода; (2) он масштабируется на более сложные архитектуры за счет использования специализированных языков, таких как nML [13], для спецификации систем команд; (3) он поддерживает циклы в программах (и, соответственно, инварианты циклов в спецификациях).

В работе [14] для верификации машинного кода на подмножествах ARM, PowerPC и x86 (IA-32) используется интерактивная система доказательства теорем HOL4 [15]. Упомянутые архитектуры были формализованы независимо: модели ARM и x86 [16,17] разработаны на HOL4, а модель PowerPC [18] – на Coq [19] (в рамках проекта CompCert [9]), а затем вручную переведена на HOL4. Автор выделяет четыре уровня абстракции: машинный код (уровень 1) автоматически декомпилируется в низкоуровневую реализацию (уровень 2); пользователь вручную разрабатывает высокоуровневую реализацию (уровень 3), а также высокоуровневую спецификацию (уровень 4); доказательство соответствия между этими уровнями гарантирует, что машинный код удовлетворяет высокоуровневой спецификации. Преимущество решения состоит в возможности переиспользования некоторых доказательств (уровни 3 и 4 не зависят от архитектуры). Еще один момент, который следует отметить, – автоматическая трансляция циклов в рекурсивные функции. На наш взгляд, уровень автоматизации можно повысить за счет использования специализированных языков описания систем команд.

Интересный подход к проверке соответствия машинного кода ACSL-спецификациям [20], рассмотрен в работе [21]. Процесс выглядит следующим образом: (1) ACSL-аннотации записываются в виде ассемблерных вставок; (2) модифицированный исходный код транслируется в язык ассемблера; (3) полученный ассемблерный код преобразуется в WhyML; (4) платформа Why3 генерирует условия верификации и проверяет их с помощью внешнего решателя. Метод похож на предлагаемый нами, однако есть существенные отличия. Основное из них заключается в том, что процесс верификации «завязан» на компилятор – при переходе с одного компилятора на другой может потребоваться доработка инструмента. Кроме того, верификация на уровне языка ассемблера не позволяет полностью отказаться от гипотезы о корректности компилятора – ассемблерный код является промежуточным представлением, подлежащим дальнейшей трансляции. Наша цель – сделать инструмент верификации как можно более независимым от компилятора и целевой машины.

В работе [22] демонстрируется возможность переиспользования доказательств корректности исходного кода для верификации машинного кода. Подход иллюстрируется на примере Java-подобного языка, компилируемого в байт-код абстрактной стековой машины. Статья посвящена кодированию с переносом доказательств (PCC – Proof-Carrying Code) и показывает, что (при определенных предположениях) компиляция сохраняет доказательства; другими словами, доказательство корректности исходного кода (построенное автоматически или интерактивно) может быть преобразовано в доказательство корректности машинного кода. Хотя идеи этого подхода могут быть полезны, решаемая нами задача отличается от описанной. Кроме того, предложенное решение привязано к конкретной аппаратной платформе.

3. Предлагаемая архитектура

В этом разделе описывается предлагаемая архитектура системы дедуктивной верификации машинного кода. Система предназначена для проверки машинного кода функции на соответствие спецификациям уровня исходного кода. Инструмент принимает на вход следующие данные:

- верифицированный *исходный код* функции и ее *спецификации* (пред- и постусловия, инварианты циклов и т.п.);
- *неоптимизированный объектный код* функции;
- *оптимизированный объектный код* функции (код, подлежащий проверке);
- *спецификация целевой архитектуры* (формальное описание регистров, режимов адресации и команд микропроцессора);
- *конфигурация компилятора и целевой машины* (размеры типов данных и двоичный интерфейс приложений).

Выходом инструмента является отчет о верификации, содержащий общий вердикт – является ли машинный код функции корректным, – а также частные вердикты для полученных в процессе анализа условий верификации. На рис. 1 изображены компоненты системы и связи между ними.

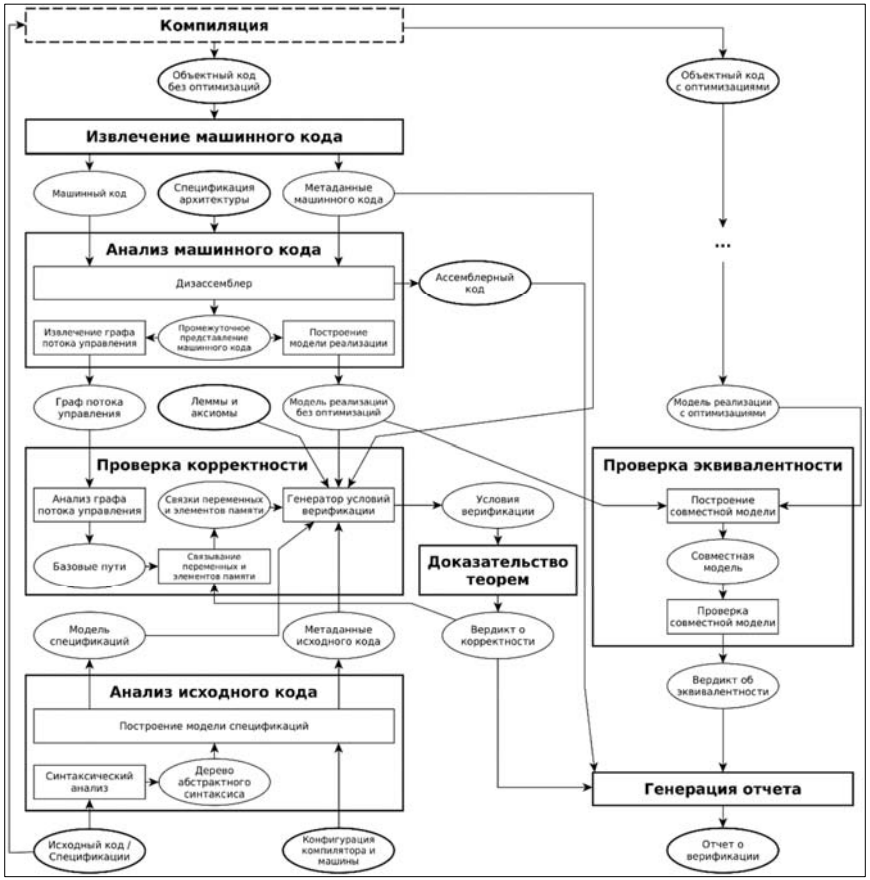

Рис. 1. Предлагаемая архитектура системы дедуктивной верификации машинного кода
Fig. 1. The proposed architecture of the system for deductive verification of machine code

3.1 Модуль извлечения машинного кода

Модуль извлечения машинного кода принимает на вход *объектный файл*, полученный при компиляции *исходного кода*, и выделяет *машинный код* заданной функции с учетом порядка байтов целевой машины. Реализация модуля зависит от формата объектного файла и может использовать существующие утилиты, например, GNU Binutils [23]. Помимо прочего, модуль извлекает из объектного файла вспомогательную информацию (*метаданные*), в том числе таблицу адресов вызываемых функций.

3.2 Модуль анализа машинного кода

Анализ машинного кода в том виде, в каком он есть, ограничивает область применения инструмента одной архитектурой. Более гибкое решение – использование архитектурно-независимого *промежуточного представления*. Для трансляции машинного кода в промежуточное представление используется *дизассемблер*. Это может быть как отдельный инструмент, разработанный для конкретной системы команд, так и модуль, автоматически построенный на основе *спецификации целевой архитектуры*. В спецификации описываются регистры микропроцессора, режимы адресации и команды. Помимо специализированного промежуточного представления дизассемблер выдает *ассемблерный код*, используемый для отладки и формирования отчета о верификации.

На основе полученного промежуточного представления строится *граф потока управления*. Для этого в последовательности машинных команд идентифицируются команды ветвления и вычисляются адреса переходов. За решение этой задачи отвечает *модуль извлечения графа потока управления*. Полученный граф снабжается дополнительной информацией, взятой из спецификации архитектуры, в том числе условиями переходов.

Модуль построения модели реализации переводит промежуточное представление машинного кода в логическую форму, понятную инструментам доказательства теорем. Сложность процедуры зависит от языка представления: если он формализован [16]-[18], в качестве модели может выступать само представление; в противном случае [13] нужны дополнительные преобразования. Для записи модели можно использовать языки, применяемые в системах доказательства теорем, такие как SMT-LIB [24], HOL [15] и Coq [19], или языки, допускающие трансляцию в указанные выше, например, WhyML [12].

3.3 Модуль анализа исходного кода

Синтаксический анализатор получает на вход исходный код функции вместе с ее спецификациями и строит *дерево абстрактного синтаксиса* (*AST – Abstract Syntax Tree*). Дерево отображается в *модель спецификаций* – множество утверждений, задающих функциональные требования (для представления утверждений также можно использовать языки, применяемые в системах доказательства теорем).

Платформы статического анализа позволяют разрабатывать плагины для трансляции исходного кода в требуемые представления. *Модуль построения модели спецификаций* может быть реализован как такой плагин. Для распространенных языков моделирования существуют готовые трансляторы, однако они не всегда подходят для анализа машинного кода (в частности, в них не учитывается двоичный интерфейс приложений).

3.4 Модуль проверки корректности

Основное различие в проверке корректности исходного и машинного кода проявляется на стадии генерации условий верификации. Для исходного кода условия верификации генерируются и доказываются независимо друг от друга. Во время компиляции информация о переменных функции теряется, что делает невозможным использование высокоуровневых

инвариантов циклов в рамках классической схемы генерации условий верификации. В общем случае требуется проверить все возможные соответствия между переменными исходного кода и элементами памяти машинного кода (включая регистры). Если k – число переменных, используемых в циклах, а n – число элементов памяти, задействованных в машинном коде, требуется рассмотреть $k!\,C_n^k$ вариантов.

Для поиска соответствия между переменными и элементами памяти используется *модуль связывания*. Сначала *анализатор графа потока управления* выделяет *базовые пути* – цепочки базовых блоков (в общем случае, ациклические подграфы), покрывающие граф и соединяющие вершины следующих типов: начало функции, конец функции, вход в цикл, выход из цикла. Связывание переменных и элементов памяти осуществляется итеративно, путем поэтапного пополнения множества *связок*: при добавлении связки проверяется истинность условий верификации всех базовых путей, зависимых от связываемой переменной; порядок перебора связок управляется эвристиками.

Ядром системы является *генератор условий верификации*, который для заданного базового пути и заданного соответствия между переменными и элементами памяти строит *условие верификации*, подлежащее доказательству. Часто для успешного доказательства условий верификации требуются вспомогательные *леммы и аксиомы*.

3.5 Модуль проверки эквивалентности

Оптимизированный код, подлежащий верификации, сопровождается *неоптимизированной версией*, полученной из того же исходного кода. При отсутствии оптимизаций сохраняется структура потока управления и, как следствие, инварианты циклов (без этого описанный выше подход не применим). Таким образом, функциональные требования проверяются для кода без оптимизаций, после чего доказывается *эквивалентность* оптимизированной и неоптимизированной версий. Используемый подход к проверке эквивалентности, как и многие другие [25,26], основан на семантическом сопоставлении моделей программ и построении совместной модели (графа совместного исполнения).

3.6 Модуль доказательства теорем

Для доказательства условий верификации используется *модуль доказательства теорем*. В качестве этого модуля можно использовать существующие решатели, как автоматические, так и интерактивные. Основное требование – поддержка битовых векторов и массивов. Данное требование возникает из-за способа моделирования микропроцессоров: (1) регистры и ячейки памяти – битовые векторы; (2) регистровые файлы и блоки памяти – массивы битовых векторов; (3) команды – операции над битовыми векторами.

Табл. 1. Реализация системы дедуктивной верификации машинного кода
Table 1. Implementation of a deductive verification system for machine code

Модуль/формат данных	**Реализация**	**Комментарий**
Объектный код	ELF [29]	Популярный формат для представления объектного кода
Модуль извлечения машинного кода	Основан на Binutils [23]	Использует readelf для извлечения машинного кода

Метаданные машинного кода	Таблица адресов функций	Содержит относительные адреса всех функций в объектном файле
Модуль анализа машинного кода	MicroTESK [30], [31]	Дизассемблирование, построение графа потока управления, построение модели реализации
Язык спецификации архитектуры	nML [13]	Описание синтаксиса языка ассемблера, двоичной кодировки и семантики команд микропроцессора
Промежуточное представление машинного кода	MIR	Внутреннее представление MicroTESK
Ассемблерный код	Язык ассемблера	Формат описывается в спецификации архитектуры
Граф поток управления	JSON	Описывает границы базовых блоков, переходы между базовыми блоками и условия переходов
Модель реализации	SMT-LIB 2.6 [24]	Код базовых блоков в SSA-форме
Исходный код и спецификации	C / ACSL [20]	Код на языке Си, аннотированный пред- и постусловиями и инвариантами циклов
Конфигурация целевой машины и компилятора	JSON	Размеры типов данных языка Си и описание двоичного интерфейса приложений
Модуль анализа исходного кода	Frama-C [32] / Why3 [12]	Frama-C – разбор Си и ACSL, плагин – трансляция ACSL в WhyML, Why3 – трансляция ACSL в SMT-LIB
Модель спецификаций	SMT-LIB 2.6 [24]	Пред- и постусловия, инварианты циклов и т.п.
Метаданные исходного кода	JSON	Описывают сигнатуры сгенерированных SMT-LIB функций

Леммы и аксиомы	SMT-LIB 2.6 [24]	Вспомогательные определения для SMT-решателей
Модуль проверки корректности	MicroVer [33]	Основная часть инструмента (см. разд. 3)
Условия верификации	SMT-LIB 2.6 [24]	Инициализация и сохранение инвариантов циклов, выполнимость постусловия
Модуль доказательства теорем	CVC4	Открытый SMT-решатель с поддержкой битовых векторов и массивов
Вердикт о корректности	Обычный текст	“Да”, “нет” или “неизвестно”: если “нет”, то предоставляется контрпример
Модуль проверки эквивалентности	MicroTESK [31]	Проверяет эквивалентность оптимизированного и неоптимизированного кода
Вердикт об эквивалентности	Обычный текст	См. вердикт о корректности
Отчет о верификации	Обычный текст	Результат верификации: результаты проверки всех условий верификации и контрпримеры

4. Апробация подхода

В этом разделе описывается реализация системы дедуктивной верификации машинного кода и ее применение для библиотечной функции memset [6], компилируемой в систему команд RISC-V [27]. В табл. 1 приведены данные о компонентах и форматах ввода-вывода, используемых в реализованной системе.

В табл. 2 представлена информация о функции `memset`: исходный код на языке Си (вместе с ACSL-спецификациями), ассемблерный код и бинарный код. Результаты верификации функции, включая сгенерированные условия верификации и инструменты, необходимые для воспроизведения шагов метода, находятся в открытом доступе [28].

Табл. 2. Исходный, ассемблерный и бинарный код функции `memset`
Tab. 2. Source, assembly and binary code of the `memset` function

Исходный код	**Ассемблерный код**	**Бинарный код**
/*@	addi sp, sp, -64	1301 01FC
requires \typeof(s) <: \type(char *);	sd s0, 56(sp)	233C 8102
requires \valid((char *)s+(0..count-1));	addi s0, sp, 64	1304 0104
assigns ((char *)s)[0..count-1];	sd a0, -40(s0)	233C A4FC
ensures \forall char *p;	addi a5, a1, 0	9387 0500
(char *)s <= p < (char *)s + count ==> *p == (char AENO) c;	sd a2, -56(s0)	2334 C4FC
ensures \result == s;	sw a5, -44(s0)	232A F4FC
*/	ld a5, -40(s0)	8337 84FD
void *memset(void *s, int c, size_t count) {	sd a5, -24(s0)	2334 F4FE
char *xs = s;	ld a5, -56(s0)	8337 84FC
/*@	sd a5, -32(s0)	2330 F4FE
loop invariant \valid((char *)xs+(0..count-1));	jal zero, 0xe	6F00 C001
loop invariant \valid((char *)s+(0..\at(count, Pre)-1));	ld a5, -24(s0)	8337 84FE
loop invariant 0 <= count <= \at(count, Pre);	addi a4, a5, 1	1387 1700
loop invariant (char *)s <= xs <= (char *)s + \at(count, Pre);	sd a4, -24(s0)	2334 E4FE
loop invariant xs - s == \at(count, Pre) - count;	lw a4, -44(s0)	0327 44FD
loop invariant \forall char *p;	andi a4, a4, 255	1377 F70F
(char *)s <= p < xs ==> *p == (char AENO) c;	sb a4, 0(a5)	2380 E700
loop assigns count, ((char *)s)[0..\at(count, Pre)-1];	ld a5, -56(s0)	8337 84FC
loop variant count;	addi a4, a5, -1	1387 F7FF
*/	sd a4, -56(s0)	2334 E4FC
while (count-- AENOC)	bne a5, zero, -18	E39E 07FC
*xs++ = (char) AENOC c;	ld a5, -40(s0)	8337 84FD
	addi a0, a5, 0	1385 0700
return s;	ld s0, 56(sp)	0334 8103
}	addi sp, sp, 64	1301 0104
	jalr zero, ra, 0	6780 0000

5. Заключение

Индустрия ПО нуждается в прикладных средствах формальной верификации. В большинстве инструментов анализируется исходный код программ; между тем, поскольку компиляторы могут содержать ошибки, наиболее критичное ПО следует дополнительно проверять на уровне бинарного кода. В этой работе предложена архитектура системы дедуктивной верификации машинного кода и описана конкретная система (реализующая эту архитектуру), позволяющая автоматически проверять машинный код на соответствие ACSL-

спецификациям, заданным для исходного кода на языке Си. Предложенный подход практически независим от целевой архитектуры, поскольку основан на спецификациях системы команд.

Работа по созданию системы не завершена, и многие ее элементы подлежат улучшению. В будущем мы планируем расширить поддержку языка ACSL и пополнить список доступных целевых архитектур (на данный момент специфицированы RISC-V, ARM, MIPS и, частично, Power). Кроме того, мы работаем над практически применимыми методами проверки эквивалентности машинных программ, полученных путем компиляции одного исходного кода с разными параметрами оптимизации. Отдельный вопрос – тестирование системы и разработка репрезентативного тестового набора (к настоящему времени система была испытана на 20 небольших функциях, что явно недостаточно для оценки ее возможностей и ограничений).

Список литературы / References

[1] R.W. Floyd. Assigning Meanings to Programs. Mathematical Aspects of Computer Science. Proceedings of Symposia in Applied Mathematics, vol. 19, 1967, pp. 19-32.

[2] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. Communications of the ACM, vol. 12, issue 10, 1969, pp. 576-585.

[3] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, G. Heiser. Comprehensive Formal Verification of an OS Microkernel. ACM Transactions on Computer Systems (TOCS), vol. 32, issue 1, 2014, pp. 2:1-2:70.

[4] E. Cohen, W. Paul, S. Schmaltz. Theory of Multi Core Hypervisor Verification. Lecture Notes in Computer Science, vol. 7741, 2013, pp. 1-27.

[5] P. Philippaerts, J.T. Mühlberg, W. Penninckx, J. Smans, B. Jacobs, F. Piessens. Software Verification with VeriFast: Industrial Case Studies. Science of Computer Programming, vol. 82, 2014, pp. 77-97.

[6] D. Efremov, M. Mandrykin, A. Khoroshilov. Deductive Verification of Unmodified Linux Kernel Library Functions. Lecture Notes in Computer Science, vol. 11245, 2018, pp. 216-234.

[7] D.R. Cok. OpenJML: JML for Java 7 by Extending OpenJDK. Lecture Notes in Computer Science, vol. 6617, 2011, pp. 472-479.

[8] A. Kamkin, A. Khoroshilov, A. Kotsynyak, P. Putro. Deductive Binary Code Verification Against Source-Code-Level Specifications. Lecture Notes in Computer Science, vol. 12165, 2020, pp. 43-58.

[9] CompCert Project. Available at: http://compcert.inria.fr, accessed 12.07.2020.

[10] C. Sun, V. Le, Q. Zhang, Z. Su. Toward Understanding Compiler Bugs in GCC and LLVM. In Proc. of the International Symposium on Software Testing and Analysis (ISSTA), 2016, pp. 294-305.

[11] M. Schoolderman. Verifying Branch-Free Assembly Code in Why3. Lecture Notes in Computer Science, vol. 10712, 2017, pp. 66-83.

[12] J.-C. Filliˆatre, A. Paskevich. Why3 – Where Programs Meet Provers. Lecture Notes in Computer Science, vol. 7792, 2013, pp. 125-128.

[13] M. Freericks. The nML Machine Description Formalism. Technical Report TR SM-IMP/DIST/08, TU Berlin CS Department, 1993, 47 p.

[14] M.O. Myreen. Formal Verification of Machine-Code Programs. Ph.D. Thesis. University of Cambridge, 2009, 131 p.

[15] K. Slind, M. Norrish. A Brief Overview of HOL4. Lecture Notes in Computer Science, vol. 5170, 2008, pp. 28-32. DOI: 10.1007/978-3-540-71067-7 6.

[16] A. Fox. Formal Specification and Verification of ARM6. Lecture Notes in Computer Science, vol. 2758, 2003, pp. 25-40.

[17] K. Crary, S. Sarkar. Foundational Certified Code in a Metalogical Framework. Technical Report CMU-CS-03-108. Carnegie Mellon University, 2003, 19 p.

[18] X. Leroy. Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant. In Proc. of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL), 2006, pp. 42-54, DOI: 10.1145/1111037.1111042.

[19] Y. Bertot. A Short Presentation of Coq. Lecture Notes in Computer Science, vol. 5170, 2008, pp. 12-16.

[20] P. Baudin, P. Cuoq, J.-C. Filliˆatre, C. March´e, B. Monate, Y. Moy, V. Prevosto. ACSL: ANSI/ISO C Specification Language. Version 1.13, 2018, 114 p.

[21] T.M.T. Nguyen, C. March´e. Hardware-Dependent Proofs of Numerical Programs. Lecture Notes in Computer Science, vol. 7086, pp. 314-329.

[22] G. Barthe, T. Rezk, A. Saabas. Proof Obligations Preserving Compilation. Lecture Notes in Computer Science, vol. 3866, pp. 112-126.

[23] GNU Binutils. Available at: https://www.gnu.org/software/binutils, accessed 12.07.2020

[24] C. Barrett, P. Fontaine, C. Tinelli. The SMT-LIB Standard Version 2.6. Release 2017-07-18. 104 p.

[25] M. Dahiya, S. Bansal. Black-Box Equivalence Checking Across Compiler Optimizations. Lecture Notes in Computer Science, vol. 10695, pp. 127-147.

[26] B. Churchill, O. Padon, R. Sharma, A. Aiken. Semantic Program Alignment for Equivalence Checking. In Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2019, pp. 1027-1040.

[27] RISC-V Foundation. Available at: https://riscv.org, accessed 12.07.2020.

[28] Results of memset binary code verification. Available at: https://forge.ispras.ru/attachments/7472, accessed 12.07.2020.

[29] Executable and Linkable Format (ELF. Available at: http://www.skyfree.org/linux/references/ELF Format.pdf, accessed 12.07.2020.

[30] M. Chupilko, A. Kamkin, A. Kotsynyak, A. Protsenko, S. Smolov, A. Tatarnikov. Test Program Generator MicroTESK for RISC-V. In Proc. of the International Workshop on Microprocessor and SOC Test and Verification (MTV), 2018, 6 p.

[31] MicroTESK Framework. Available at: http://www.microtesk.org, accessed 12.07.2020.

[32] Frama-C Platform. Available at: http://frama-c.com, accessed 12.07.2020.

[33] MicroVer Project. Available at: https://forge.ispras.ru/projects/microver, accessed 12.07.2020.

[34] CVC4 Solver. Available at: https://github.com/CVC4/CVC4, accessed 12.07.2020.

Информация об авторах / Information about authors

Илья Владимирович ГЛАДЫШЕВ – студент магистратуры по направлению «Системное программирование» в ВШЭ. Сфера научных интересов: формальная верификация и операционные системы.

Ilya Vladimirovich GLADYSHEV is a Master's student studying system programming at the National Research University "Higher School of Economics". His research interests include formal verification and operating systems.

Александр Сергеевич КАМКИН – кандидат физико-математических наук, ведущий научный сотрудник отдела технологий программирования ИСП РАН; преподает в МГУ, МФТИ и ВШЭ. Область научных интересов: проектирование цифровой аппаратуры, верификация и тестирование, статический и динамический анализ HDL-описаний, высокоуровневый синтез.

Alexander Sergeevich KAMKIN is a leading researcher at the Software Engineering Department of Ivannikov Institute for System Programming of the Russian Academy of Sciences (ISP RAS). He is also a lecturer at Moscow State University (MSU), Moscow Institute of Physics and Technology (MIPT), and Higher School of Economics (HSE). His research interests include digital hardware design, verification and testing, static and dynamic analysis of HDL descriptions, and high-level synthesis. Alexander has a PhD in Physics and Mathematics.

Артем Михайлович КОЦЫНЯК младший научный сотрудник отдела технологий программирования. Область научных интересов: верификация и тестирование, компиляторные технологии, высокоуровневый синтез и формальные методы.

Artem Mikhailovich KOTSYNYAK is a junior researcher at the Software Engineering Department of Ivannikov Institute for System Programming of the Russian Academy of Sciences (ISP RAS). His research interests include verification and testing, compiler technologies, high-level synthesis and formal methods.

Павел Андреевич ПУТРО получил степень бакалавра в области программной инженерии и степень магистра в области системного программирования в ВШЭ, Москва, Россия. Работает ИСП РАН. Исследовательские интересы включают дедуктивную верификацию, логическое программирование и статический анализ машинного кода.

Pavel Andreevich PUTRO received a bachelor's degree in software engineering and a master's degree in system programming from the National Research University Higher School of Economics, Moscow, Russia. He works at the Ivannikov Institute for System Programming of the RAS. His research interests include deductive verification, logic programming, and machine code static analysis.

Алексей Владимирович ХОРОШИЛОВ, ведущий научный сотрудник, кандидат физико-математических наук, директор Центра верификации ОС Linux в ИСПРАН, доцент кафедр системного программирования МГУ, ВШЭ и МФТИ. Основные научные интересы: методы проектирования и разработки ответственных систем, формальные методы программной инженерии, методы верификации и валидации, тестирование на основе моделей, методы анализа требований, операционная система Linux.

Alexey Vladimirovich KHOROSHILOV, Leading Researcher, Ph.D. in Physics and Mathematics, Director of the Linux OS Verification Center at ISP RAS, Associate Professor of System Programming Departments at Moscow State University, the Higher School of Economics, and Moscow Institute of Physics and Technology. Main research interests: design and development methods for critical systems, formal methods of software engineering, verification and validation methods, model-based testing, requirements analysis methods, Linux operating system.