DOI: 10.15514/ISPRAS-2019-31(3)-10

Applying High-Level Function Loop Invariants for Machine Code Deductive Verification

P.A. Putro, ORCID: 0000-0001-9540-8321 <pavel.putro@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia
National Research University Higher School of Economics,
20, Myasnitskaya st., Moscow, 101000 Russia

Abstract. The existing tools of deductive verification allow us to successfully prove the correctness of functions written in high-level languages such as C or Java. However, this may not be enough for critical software because even fully verified code cannot guarantee the correct generation of machine code by the compiler. At the moment, developers of such systems have to accept the compiler correctness assumption, which, however, is extremely undesirable, but inevitable due to the lack of full-fledged systems of formal verification of machine code. It is also worth noting that the verification of machine code by a person directly is an extremely time-consuming task due to the high complexity and large amounts of machine code. One of the approaches to simplify the verification of machine code is automatic deductive verification reusing the formal specification of the high-level language function. The formal specification of the function consists of the specification of the pre- and postcondition, as well as loop invariants, which specify conditions that are satified at each iteration of the loop. When compiling a program into machine code, pre- and postconditions are preserved, which, however, cannot be said about loop invariants. This fact is one of the main problems of automatic verification of machine code with loops. Another important problem is that high-level function variables often have ‘projections' to both registers and memory at the machine code level, and the verification procedure requires that invariants be described for each variable, and therefore the missing invariants must be generated. This paper presents an approach to solving these problems, based on the analysis of the control flow graph, and intelligent search of the locations of variables.

Keywords: deductive verification; formal methods; machine code

For citation: Putro P.A. Applying High-Level Function Loop Invariants for Machine Code Deductive Verification. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 3, 2019. pp. 123-134. DOI: 10.15514/ISPRAS-2019-31(3)-10

Использование инвариантов функции высокого уровня для дедуктивной верификации машинного кода

П.А. Путро, ORCID: 0000-0001-9540-8321 <pavel.putro@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
Национальный исследовательский университет “Высшая школа экономики”
101000, Россия, г. Москва, ул. Мясницкая, д. 20

Аннотация. Существующие на сегодняшний день инструменты дедуктивной верификации позволяют успешно доказывать корректность функций, написанных на высокоуровневых языках, таких как C или Java. Однако для критического ПО этого может быть недостаточно, поскольку даже полностью верифицированный код не может гарантировать корректной генерации машинного кода компилятором. На данный момент разработчикам таких систем приходится принимать предположение о корректности

компилятора, что, однако, является крайне нежелательным, но неизбежным поступком в силу отсутствия полноценных систем формальной верификации машинного кода. Стоит также отметить, что верификация машинного кода человеком напрямую является крайне трудоёмкой задачей из-за высокой сложности и больших объёмов машинного кода. Одним из подходов, позволяющих упростить верификацию машинного кода, является автоматическая дедуктивная верификация с переиспользованием формальной спецификации функции языка высокого уровня. Формальная спецификация функции состоит из спецификации пред- и постусловия, а также инвариантов циклов, позволяющих определить какие условия сохраняются на каждой итерации цикла. При компиляции программы в машинный код пред- и постусловия сохраняются, что, однако, нельзя сказать об инвариантах циклов. Этот факт является одной из основных проблем автоматической верификации машинного кода с циклами. Другой немаловажной проблемой является то, что локальные переменные функций высокого уровня могут иметь ‘позиции’ как на регистры, так и на память на уровне машинного кода. Если абстрагироваться от конкретного компилятора, то не существует строгих правил сопоставления локальных переменных их позициям, а процедура верификации инвариантов циклов, тем не менее требует того, чтобы локальным переменным были сопоставлены конкретные позиции. В данной работе приводится подход к решению этих проблем, а также рассматриваются альтернативные пути решения, предложенные в аналогичных исследованиях.

Ключевые слова: дедуктивная верификация; формальные методы; машинный код.

Для цитирования: Путро П.А. Использование инвариантов функции высокого уровня для дедуктивной верификации машинного кода. Труды ИСП РАН, том 31, вып. 3, 2019 г., стр. 123-134 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(3)-10

1. Introduction

In the 1960s, Floyd [1] and Hoare [2] put forward their theories that the full correctness of the program code can be proved mathematically. The proposed methods are called deductive verification, but could not immediately gain popularity due to the lack of automation, as well as low performance and the high cost of hardware computing resources. However, in recent decades, these methods are experiencing a rebirth due to the rapid development of methods for solving the SMT [3] problem, and growing performance of hardware devices. Technological leap in this area allowed to verify the application and system software by means of personal computers. In this paper, the author adheres to the use of methods of deductive verification, because unlike other methods of formal verification, such as for example model checking, deductive verification allows proving full correctness, but not only the absence of a certain class of errors.

Increasing the availability of formal verification methods has led to the fact that now formal verification is becoming a standard in the creation of systems designed to work with safety- and security-critical infrastructure. These systems are verified and tested carefully, but industrial verification tools work only at source code level when testing can't guarantee that are no errors in the program. Here we can notice a security hole when compilation of the correct code introduces errors that can't be detected by the testing system. Without anyways for solving this problem – developers have to make «The assumption of the correctness of the compiler». According to a study [4] conducted in 2016, the total number of bugs found in GCC+LLVM are more than 50000. This is one of the main reasons why this assumption may not be sufficient for critical systems. There are only two ways that can allow developers to abandon this assumption: the first is to create a fully formally verified compiler, and the second is to formally verify machine code. There are some tries in recent 15 years that aimed to verify machine code or to create fully formally verified compiler but there is still no generally accepted industrial solution in machine code verification.

In this paper we consider an approach of deductive verification of machine code obtained by compiling the source code, the correctness of which has been proved by methods of deductive verification. The approach proposes to use a number of techniques designed to reuse the function specification in a high-level language to prove the correctness of the compiled machine code.

The process of deductive verification of machine code has several serious differences from deductive verification of code in high-level languages. The first difference is that in machine-independent high-level programming languages, the set of basic operations amounts to several tens, and the size of instruction sets of modern processors amounts to hundreds and may even exceed a thousand different instructions. In addition, many of these instructions can have side effects, such as setting processor flags or storing the result in a predefined register. This variety of instructions does not allow to effectively generate state-change formulas during parsing of machine code and requires a definition of the processor model and its instruction set. The second difference is that machine code is always a sequence of instructions with operands and does not have the complex syntax that is present in modern programming languages. This feature allows you to automate the parsing of the machine code of different processors using only one tool. The third difference is that in machine code there is no explicit design to indicate the loops, such as operators «for» or «while» in languages C/C++. Instead, loops are organized by using a set of conditional and unconditional branches. However, the presence of such instructions does not mean that there are loops in the program because they are also used to organize any branching. This difference requires the construction and analysis of the control flow graph of the program. In this case, the control flow graph extractor should be able to distinguish branches from other processor instructions, be able to determine the target of the transition and the condition under which it will be done. If there is a control flow graph, the problem of finding loops in the program can be reduced to the problem of finding the components of strong connectivity in the oriented graph. The last significant difference is the absence of a direct connection between the names and positions of local variables in a high-level language program and their location in memory or in the registers of the program in machine code. It is necessary when we try to reuse specifications of the high-level function. A similar dependence exists when mapping function parameters to registers and stack and is determined by the target processor ABI. Using information about ABI allows you to automatically map the parameters and the result of the function to the appropriate positions in the machine code. However, the process of proving loop invariants involves the use of local variables. As a result, when trying to prove the invariants of a high-level function at the machine code level, there is a problem associated with the absence of the ability to directly associate high-level local variables with machine code. There are also other differences related to program function calls, system calls, and exception handling, but these are beyond the scope of this paper. As you can see the first two considered differences are common and observed in the processing of any machine code, while the second two appear only in the case of processing functions with loops. In this paper, the most attention is paid to the solution of the problems caused by the second two differences while the previous author's paper is devoted to the first two [5].

2. Related work

In [6], the HOL4 proof assistant [7] is used to verify machine programs in subsets of ARM, PowerPC, and x86 (IA-32). These ISAs were specified independently: the ARM and x86 models [8], [9] were written in HOL4 while the PowerPC model [10] was written in Coq [11] and then manually translated to HOL4. There are four levels of abstraction. Machine code (level 1) is automatically decompiled into the low-level function model in HOL4 (level 2). A user describes the high-level function model (level 3) as well as the functional specification (level 4). By proving the equivalence between the levels, the user ensures that the machine code complies with the functional specification. In our opinion, automation can be increased by using specialized ISA description languages and SMT solvers. For proving the correctness of the programs with loops, it uses loops to recursive functions translation technique. This technique is available only for interactive provers due to efficiency problems of automatic solvers while processing programs with loops.

An interesting approach aimed at verifying machine code against ACSL [12] specifications is presented in [13]. The general scheme is as follows: first, the ACSL annotations are rewritten as an inline assembly; second, the modified sources are compiled into assembly language; third, the

assembly code is translated into a Why program; finally, the Why environment generates verification conditions and proves them with external solvers. The approach looks similar to the proposed one; however, there some distinctions. E.g., there are separate primitives for storing/loading variables of a different type (32- and 64-bit integers, single and double floating-point numbers, etc.), which leads to certain limitations in dealing with pointers. It is also worth noting that verification at the assembly level does not allow us to abandon the compiler correctness assumption, as the assembly code is an intermediate form and needs additional translation. This approach relies on the compiler while processing programs with loops, as compiler places rewritten as inline assembly loop invariants into the beginning of the loop and automatically bind local variables to the registers or memory.

In [14], there have been demonstrated the possibility of reusing correctness proofs of high-level programs for the related machine code verification. The approach is illustrated on the example of a Java-like source language and a bytecode target language. The paper describes a scenario of using such a technology in the context of proof-carrying code (PCC) and shows (in a particular setting) that compilation preserves proof obligations, i.e. source code proofs (built either automatically or interactively) can be transformed to the machine code proofs. The problem we are solving is different (though some ideas may be useful); moreover, we would like to make our solution architecture and compiler independent. In the case of the processing of the programs with loops, if loop invariants are preserved by compilation, their proving will be a trivial process.

3. Using the control flow graph for VC generation

In deductive verification of programs in high-level languages, various syntactic constructions allow determining the presence of loops in the program, their contents, as well as the conditions of exit from the loop. However, when processing the machine code such structures do not exist, and to search for loops and other branch operations need to build a control flow graph (CFG). As part of the study for the processing of machine code used MicroTESK toolkit [15] (full justification for the use of MicroTESK for deductive verification of machine code is given in paper [5]). The use of this tool, in particular, allows to describe the processor model in the language of nML [16], and on the basis of this model to automatically analyze the binary code and build its behavior model in the logical language SMT-LIB [17]. In addition, CFG extractor has been added to this tool over the past year.

3.1 The format of the CFG

MicroTESK toolkit is able to determine whether the instruction described in nML is a branch instruction, determine the branch condition and the target address. In addition, MicroTESK has an advanced algorithm for calculating the target address of the transition, which allows it to calculate indirect targets, such as in a situation where the target address is preloaded into the register and the branch is carried out already on the register. Such capabilities in combination with the use of nML processor models allow you to automatically generate CFG for any processor modeled using nML. The generated CFG is saved in JSON format [18], and has the following format.

1) All basic blocks are placed in the list with the name «blocks».
2) Each basic block has an index in the «blocks» list and has the following format:
   a. The «range» list that includes the sequence number of the first and last instruction of the base block in the context of the entire function being analyzed. Used for extraction of the SMT-LIB representation of the block from the SMT-LIB representation of machine code.
   b. The «asm» list that contains instructions of a basic block in the assembler language of the target processor.
   c. The «vars_start» list, which contains the SMT-LIB versions of the main variables of the nML model of the processor such as registers and memory, but not temporary and auxiliary variables. Versions are specified for the entry point of the basic block.

d. The «vars_end» list contains values similar to the list of «vars_start», however, the version specified for the exit point of the basic block.
e. The field «condition» contains the branch condition. The MicroTESK nML internal representation syntax is used to write the condition «true» for unconditional branches and in the case when there is no branch in the block.
f. The field «condition_smt» same as «condition», however, is recorded using SMT-LIB.
g. The field «target_taken» containing the index of the basic block in the «blocks» list, which will be passed to the control in the case when «condition» is met, and «null» for blocks, in which is the function exit point.
h. Optional field «target_ntaken» containing the index of the basic block in the "blocks" list, which will be passed to the control in the case when «condition» is not met. Defined only for blocks with a conditional branch.

This structure of the graph contains all the necessary information for generating verification conditions. Below is an example of the extracted CFG for the function of calculating the sum of numbers from 0 to N (Table 1).

```
{
  "blocks": [
  {
    "range": [0, 8],
    "vars_start": ["MEM!1","XREG!1"],
    "vars_end": ["MEM!37","XREG!15"],
    "asm": [
      "addi sp, sp, -48",
      "sd s0, 40(sp)",
      "addi s0, sp, 48",
      "addi a5, a0, 0",
      "sw a5, -36(s0)",
      "sw zero, -20(s0)",
      "addi a5, zero, 1",
      "sw a5, -24(s0)",
      "jal zero, 0x10"
    ],
    "condition": "true",
    "target_taken": 1
  },
  {
    "range": [16, 20],
    "vars_start": ["MEM!53","XREG!27"],
    "vars_end": ["MEM!53","XREG!36"],
    "asm": [
      "lw a4, -24(s0)",
      "lw a5, -36(s0)",
      "addiw a4, a4, 0",
      "addiw a5, a5, 0",
      "bge a5, a4, -22"
    ],
    "condition": "i1 sge i64 a5, a4",
    "condition_smt":
"op_20_instruction.operation.action.block_0!1",
    "target_taken": 2,
    "target_ntaken": 3
  },
  {
    "range": [9, 15],
    "vars_start": ["MEM!37","XREG!15"],
    "vars_end": ["MEM!53","XREG!27"],
    "asm": [
```



```
      "lw a4, -20(s0)",
      "lw a5, -24(s0)",
      "addw a5, a4, a5",
      "sw a5, -20(s0)",
      "lw a5, -24(s0)",
      "addiw a5, a5, 1",
      "sw a5, -24(s0)"
    ],
    "condition": "true",
    "target_taken": 1
  },
  {
    "range": [21, 25],
    "vars_start": ["MEM!53","XREG!36"],
    "vars_end": ["MEM!53","XREG!45"],
    "asm": [
      "lw a5, -20(s0)",
      "addi a0, a5, 0",
      "ld s0, 40(sp)",
      "addi sp, sp, 48",
      "jalr zero, ra, 0"
    ],
    "condition": "true",
    "target_taken": null
  }
  ]
}
```

Table 1. Example: ACSL-annotated C code, RISC-V assembler code and machine code of sum function

ACSL-annotated C code	Assembly code	Machine code

`/*@ axiomatic Sum {`
`*@ logic integer sum(integer n);`
 	
`*@ axiom sum_init:`
`*@ \forall integer n;`
`*@ n <= 0 ==> sum(n) == 0;`
 	
`*@ axiom sum_step_dec:`
`*@ \forall integer n;`
`*@ n > 0 ==> sum(n) == sum(n-1) + n;`
`*@ }`
`*/`
 	
`/*@ requires 0 <= n <= 65535;`
`*@ ensures \result == sum(n);`
`*/`
`int sum(int n) {`
`  int s = 0;`
`  /*@ loop invariant 1 <= i <= n+1;`
`  *@ loop invariant s == sum(i-1);`
`  *@ loop variant n-i;`
`  */`
`  for(int i = 1; i <= n; i++) {`
`    s += i;`
`  }`
`  return s;`
`}`
	`addi sp, sp, -48`
	`sd s0, 40(sp)`
	`addi s0, sp, 48`
	`addi a5, a0, 0`
	`sw a5, -36(s0)`
	`sw zero, -20(s0)`
	`addi a5, zero, 1`
	`sw a5, -24(s0)`
	`jal zero, 0x10`
	`lw a4, -20(s0)`
	`lw a5, -24(s0)`
	`addw a5, a4, a5`
	`sw a5, -20(s0)`
	`lw a5, -24(s0)`
	`addiw a5, a5, 1`
	`sw a5, -24(s0)`
	`lw a4, -24(s0)`
	`lw a5, -36(s0)`
	`addiw a4, a4, 0`
	`addiw a5, a5, 0`
	`bge a5, a4, -22`
	`lw a5, -20(s0)`
	`addi a0, a5, 0`
	`ld s0, 40(sp)`
	`addi sp, sp, 48`
	`jalr zero, ra, 0`
		`fd01 0113`
		`0281 3423`
		`0301 0413`
		`0005 0793`
		`fcf4 2e23`
		`fe04 2623`
		`0010 0793`
		`fef4 2423`
		`0200 006f`
		`fec4 2703`
		`fe84 2783`
		`00f7 07bb`
		`fef4 2623`
		`fe84 2783`
		`0017 879b`
		`fef4 2423`
		`fe84 2703`
		`fdc4 2783`
		`0007 071b`
		`0007 879b`
		`fce7 dae3`
		`fec4 2783`
		`0007 8513`
		`0281 3403`
		`0301 0113`
		`0000 8067`

3.2 Joining basic blocks for verification conditions generation

The basic blocks themselves are not suitable targets for generating verification conditions (VC), as they may not contain specific targets, but only state change formulas. There are several types of verification conditions in deductive verification. The first and foremost is the postcondition. Also as VC can be used various custom asserts or conditions for checking the security of the program execution, such as for example the absence of indexing out of range of the array. Also, as VC uses invariants of loops. In this case, each invariant can be further divided into checking the initialization of this invariant – that is, checking the condition of the invariant before the execution of the loop code, as well as checking the preservation of the invariant - the preservation of the compliance of the invariant for the next iteration of the loop, provided that all the invariants are compliances on the current one. Therefore, the basic blocks must be joined and marked so that one or more of these conditions can be matched to each of them. Accordingly, the algorithm for combining the base blocks can be defined as follows. In the first step, using the fields "target_taken" and "target_ntaken", the array of edges of the CFG is selected from the set of basic blocks. In the second step, to search for loops in the program, the graph uses an algorithm to search for strongly related components in a directed graph. The author's implementation uses Tarjan's algorithm [19] implemented by the ocaml-containers library [20]. To find nested loops, this step must be repeated recursively for all found base block sets, and the relationship between the first and last base block in the loop must be broken. In the third step, you need to depth-first search the graph for marking and joining blocks. The traversal must start from the zero base block – the program entry point. At the input, there is a set of basic blocks, as well as a set of chains of strongly related component - loops. The output is a set of joined and marked basic blocks suitable for VC generation.

1) If the block has two targets, they must be processed separately, and the results combined.
2) If the current block and its target is not in the loop– it is necessary to "join" these blocks and proceed to the processing of the joined block.
3) If the current block is not included in the loop, and its target is included in the loop, it must be marked as the loop entry point. Next, proceed to the processing of its target.
4) If the block and its target are in the same loop and do not make a loop, they must be joined and proceed to the processing of the joined block.
5) If the block and its targets are in the same loop and thus make a loop, they must be combined and the result is returned.
6) If the unit is part of the loop, and its target is included in a nested loop it is necessary to mark as a loop entry point and proceed to the processing its target.
7) If the block is included in the nested loop, and its target in the outer loop, then the block must be marked as the loop exit point and joined with the target and proceed to the processing of the joined block.
8) If the block in the loop but its target is no, then the block must be marked as the exit point of the loop and joined with the target and proceed to process the joined block.
9) If the block target is null, the result must be returned.

The procedure of joining blocks is the base for the graph traversal and is carried out according to the following rules.

1) The procedure allows you to create a new block based on two existing ones.
2) Joining is possible if the target («target_taken» or «target_ntaken») of the first block is the second block. In all other cases, the result of the join is not determined.
3) The targets of the joined block will be the targets of the second block.
4) Condition («condition_smt») of the joined block will be the condition of the second block.

5) If the second block is a loop exit point, the initial state «vars_start» of the combined block will be the initial state of the second block, otherwise the initial state of the first block.
6) If at least one of the joining blocks is marked as the loop exit point, the joined block must also be marked as the loop exit point.
7) If the second block is marked as the loop entry point, the result should be marked as the loop entry point.
8) If the second block «closes» the loop, i.e. its target is the first block, its SMT-LIB representation should be changed so that all elements of the final state «vars_end» of the second block should get new unique names. Any other conflicts between any variables in SMT-LIB representations of the joining blocks must be resolved in the same way.
9) SMT-LIB the representation of the joined block must be obtained by concatenating the SMT-LIB representations of the merged blocks. In this case, if the condition «condition» of the first block is not empty («true»), it must be added as SMT-LIB assert to the representation code of the joined block. Also, if the join follows the «target_ntaken» branch, the condition must be inverted. Also, if the blocks do not follow each other in the program, the final state of the first block also needs to be associated with the initial state of the second block at the level of the SMT-LIB representation.

As a result of following this algorithm, in most cases, you can create a set of code blocks on which you can directly prove various verification conditions. The algorithm allows processing machine code with loops, nested loops, sequential loops, as well as code generated by the presence of the brake and continue statements in the program, but is not able to cope with tasks when, for example, several entries to one loop and other non-trivial situations caused by the use of transition instructions are detected in the control flow graph. However, such situations cause difficulties already at the stage of verification of the source code, and the construction of an algorithm that allows you to automatically deductively verify any machine code is an unsolvable task.

If we apply the algorithm to the CFG function of the sum of the numbers presented above, we will be able to allocate three blocks to prove VC. The first block will have index 0, have loop entry status and be used to prove the correctness of the initialization of the loop invariants. The second block will be a join of blocks 1 and 2 and will be used to prove the preservation of loop invariants. The third block will be a join of blocks 1 and 3 and will be used to prove the postcondition provided the invariants are correct.

4. Automatic binding of high- and low-level local variables

In general, to describe loop invariants, high-level functions use local variable names that are not available when working with machine code. In general, information about binding local variables to specific positions on the stack or registers is not available. Of course, you can require the user to manually provide this data and even give examples where such requirements will have a positive impact on system performance. However, in most cases, manual mapping of local variables to their positions will be a bottleneck in the performance of the verification system, as well as reduce the degree of automation. From the above, we can conclude that the system should automatically determine the location of local variables, and the possibility of their manual input should be optional. To determine the positions of local variables, the author has developed an algorithm for efficient search of positions in the VC generation, which includes the following steps.

1) All potential positions of local variables are calculated. This can be done both by means of machine code analysis (similar to those used by modern disassemblers and debuggers) and with the help of an existing logical model of machine code and SMT-solver. The author proposes to use the second option because it is a more universal approach. In this approach, for each memory write instruction it is required to prove by solver that there are no positions on the stack that could change as a result of the operation. If solver managed to generate a counterexample, the

position found is a potential position for the local variable. This algorithm allows you to find positions for local variables, but it can be difficult if there is an array on the stack. In this case, if solver fails to determine which position of the array was recorded, the result may not be determined and the user will have to specify the position himself. Similarly, you can calculate the registers to which the local variable can be mapped.

2) Each local variable is assigned a potential positions set, which may depend on its size in bytes.
3) For each invariant from the list of invariants for the proof, the «cost» of proving its correct initialization should be calculated. Here, the cost is the number of all possible combinations of potential positions of local variables involved in the description of this invariant. Here it is necessary to take into account that each variable has its own unique memory location, therefore, combinations of potential positions should consist only of unique values. It is also worth noting that this step should be carried out only when proving the initialization of the invariants, since the initialization of the invariant is proved independently of the other loop invariants, and the proof of preserving the loop invariant must be proved given that all other invariants must be satisfied. Thus, when proving the correct initialization of the invariant, it is possible to reduce the number of local variables necessary for binding, and, as a consequence, the «cost» of the proof may differ for different invariants. If it is necessary to prove the loop invariant preservation the cost of all invariants should be considered equal to.
4) For the least cost invariant, it is necessary to try to prove correctness for each possible combinations of potential positions of local variables on which the invariant depends. The results («unsat»/«unknown»/«sat» verdicts) should be saved in a separate list.
5) If there was no one verdict is «unsat», it is necessary to mark the invariant unproven. If at least one verdict «unsat» has been obtained, then the invariant should be considered proved and the potential positions of local variables on which the proved invariant depends should be filtered, leaving only those positions that consist in combinations of potential positions for which the invariant has been proved (the «unsat» verdict).
6) Remove the proved invariant from the list of invariants for the proof and (if there are still invariants in the list) proceed to step 3.

Using this algorithm, it is possible in some cases to significantly reduce the number of generated targets relative to a complete search. As an example, let's take a function with 3 different local variables with values $s = -1, i = 0, n = 90$ at the loop entry point and three potential positions on the stack: $sp - 0x10, sp - 0x18, sp - 0x24$, respectively. For simplicity, the size of variables and positions will be considered equal to 4 bytes. Initially, the correspondence between the positions of local variables is not known. Based on these data, we try to prove the initialization of the following invariants: $0 <= i < 100, s == 2 * i - 1$ and $s < n + i$. According to the algorithm, we calculate the cost of proving the invariants, which will be equal to 3, 6 and 6, respectively. Next, try to prove the correctness of the first invariant with the cost of 3 and will find two potential positions ($sp - 0x18$ and $sp - 0x24$) for the variable i (we will also be able to prove the invariant for variable n). Second, we perform filtering to remove the position of the $sp - 0x10$ for the variable i. Recalculate the costs of the remaining invariants: the result of 4 and 4 (the cause of reducing the cost is reducing the number of potential positions for the variable i, on which the invariants depend). When proving the correctness of the initialization of the second invariant, only one set of potential positions for the variables s and i is $sp - 0x10$ and $sp - 0x18$, respectively, will be selected. We filter and proceed to the proof of the last invariant. It is possible to select only one target for it – the position for n will be selected by the elimination method. We prove the invariant, perform filtering and get the same correspondence between variables and their positions on the stack, which was set by the compiler.

5. Evaluation

At the time of writing, the approach proposed by the author was partially implemented in the system of deductive verification of machine code [5]. Using this approach, the machine code of the function of the sum of numbers from 0 to N (Table 1), as well as some other functions with loops, was successfully verified. For each generated VC, a verdict was obtained confirming its correctness. In addition, the positions of local variables on the function stack were strictly determined by the computer during the verification process. More complex testing is planned after the full implementation of the approach.

6. Conclusion

Verification of functions with loops is one of the main stumbling blocks in the verification of machine code. Various research groups have proposed various solutions that however impose serious limitations, such as the need to use interactive proof assistants or the introduction of dependency on the target compiler. However, due to the high complexity of the machine code structure, the use of interactive proof assistants can significantly slow down the verification process and require very experienced staff. Dependency on the target compiler also reduces the universality of the approach and requires its integration into the source code compilation process, which can cause some difficulties. In contrast to these works, the author proposes to use a compiler-independent approach based on the use of automatic SMT-solvers. To implement the approach, we propose to use two main algorithms, as well as a tool for CFG extraction. The first algorithm allows the basic blocks of the function CFG to be joined in such a way that they become suitable for VC proving. The second algorithm allows restoring the lost links between the local variables of the high-level function and their positions in memory or on the processor registers at the machine code level. Using this approach allows in most cases to generate such VC, which will be sufficient for deductive verification of the machine code of the function with loops.

The work is in progress. At the moment, the approach has been partially implemented in the system of deductive verification of machine code. Its full implementation and testing is the nearest direction for further work. Also among the possible areas for further research can be identified the study of problems arising in the proof of the correctness of functions containing calls to other functions or system calls. There is a separate issue with the security check of the machine code execution, i.e. the absence of exceptions at the processor level, incorrect memory readings or stack overflows. Also of great importance is the study of the applicability of the machine code deductive verification system for solving real industrial problems.

References

[1]. R.W. Floyd. Assigning Meanings to Programs. Mathematical Aspects of Computer Science, vol. 19, 1967. P. 19-32.
[2]. C.A.R. Hoare. An Axiomatic Basis for Computer Programming. Communications of the ACM, vol. 12, no. 10, 1969, pp. 576-585.
[3]. C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, 2009, pp. 825-885.
[4]. C. Sun, V. Le, Q. Zhang, Z. Su. Toward understanding compiler bugs in gcc and llvm. In Proc. of the 25th International Symposium on Software Testing and Analysis, 2016, pp. 294-305.
[5]. Putro P.A. Combining ACSL Specifications and Machine Code. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 4, 2018. pp. 95-106. DOI: 10.15514/ISPRAS-2018-30(4)-6
[6]. M.O. Myreen. Formal Verification of Machine-Code Programs. Ph.D. Thesis. University of Cambridge, 2009, 131 p.
[7]. K. Slind, M. Norrish. A Brief Overview of HOL4. Lecture Notes in Computer Science (LNCS), vol. 5170, 2008, pp. 28-32.
[8]. A. Fox. Formal Specification and Verification of ARM6. Lecture Notes in Computer Science (LNCS), vol. 2758, 2003, pp. 25-40

[9]. K. Crary, S. Sarkar. Foundational Certified Code in a Metalogical Framework. Technical Report CMU-CS-03-108. Carnegie Mellon University, 2003. 19 p.
[10]. 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, pp. 42-54.
[11]. Y. Bertot. A Short Presentation of Coq. Lecture Notes in Computer Science, vol. 5170, 2008, pp. 12-16.
[12]. P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, V. Prevosto. ACSL: ANSI/ISO C Specification Language. Version 1.13, 2018, 114 p.
[13]. T.M.T. Nguyen, C. Marché. Hardware-Dependent Proofs of Numerical Programs. Lecture Notes in Computer Science, vol. 7086, 2011, pp. 314-329.
[14]. G. Barthe, T. Rezk, A. Saabas. Proof Obligations Preserving Compilation. Lecture Notes in Computer Science, vol. 3866, 2005, pp. 112-126.
[15]. MicroTESK Framework – http://www.microtesk.org
[16]. M. Freericks. The nML Machine Description Formalism. Technical Report TR SM-IMP/DIST/08, TU Berlin CS Department, 1993, 47 p.
[17]. C. Barrett, P. Fontaine, C. Tinelli. The SMT-LIB Standard Version 2.6. Release 2017-07-18, 104 p.
[18]. JavaScript Object Notation – https://www.json.org/
[19]. R. E. Tarjan, Dep-first search and linear graph algorithms. SIAM Journal on Computing, vol. 1, no. 2, 1972, pp. 146-160.
[20]. ocaml-containers library – https://github.com/c-cube/ocaml-containers.

Информация об авторах / Information about authors

Павел ПУТРО получил степень бакалавра в области программной инженерии в Национальном исследовательском университете «Высшая школа экономики», Москва, Россия. В настоящее время он продолжает обучение в этом университете по магистерской программе «Системное программирование». Работает в институте системного программирования им. В.П. Иванникова РАН. Исследовательские интересы включают дедуктивную верификацию, логическое программирование и статический анализ машинного кода.

Pavel PUTRO received a bachelor's degree in software engineering from the National Research University Higher School of Economics, Moscow, Russia. Currently, he is continuing his studies at this University on the master's program «System programming”. 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.