Architecture of a Machine Code Deductive Verification System

Alexander Kamkin[1,2,3,4], Alexey Khoroshilov[1,2,3,4], Artem Kotsynyak[1], Pavel Putro[1,4], Ilya Gladyshev[4]
[1] Ivannikov Institute for System Programming of the Russian Academy of Sciences (ISP RAS)
[2] Lomonosov Moscow State University (MSU)
[3] Moscow Institute of Physics and Technology (MIPT)
[4] National Research University – Higher School of Economics (HSE)
Email: {kamkin, khoroshilov, kotsynyak, pavel.putro}@ispras.ru, ilya.v.gladyshev@gmail.com

***Abstract*—In recent years, ISP RAS has been developing a system for machine 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 binary 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 and allows a user not to interact directly with the compiler output: 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 components including static analysis and verification platforms (Frama-C and Why3), 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.**

***Index Terms*—formal methods, deductive verification, binary code analysis, equivalence checking, instruction set architecture, machine code, compiler testing**

I. Introduction

The role of software in safety- and security-critical infrastructure grows continuously and at an ever-increasing speed. As a result, there is a high demand in practical methods and tools to ensure correctness of the most important components. There are a number of research projects in the area: some of them confine themselves to checking the absence of specific kinds of bugs (e.g., run-time errors), while the others try to prove *total correctness* of the software under analysis. The total correctness typically means that each possible execution of the software component *terminates* and meets the *functional contract* expressed in the form of *pre-* and *postconditions* on the component's interfaces. To prove such kind of properties, *deductive verification methods* are usually applied.

While the first ideas of the methods appeared in the works of R.W. Floyd [1] and C.A.R. Hoare [2] at the end of 1960s (inductive assertions, axiomatic semantics, etc.), deductive verification of production software became realistic just recently [3]–[7]. All the examples of deductive verification tools for the imperative programming paradigm follow the similar approach [8]:

- all statements of the programming language get *formal semantics*;
- functional requirements to the software component are formalized as *pre-* and *postconditions* of the functions (or methods) in a *specification language*;
- additional hints to a verification framework such as *loop invariants*, *ghost code*, and *lemma functions* are provided by a user;
- *verification conditions* (*VCs*) are generated by the framework and are discharged either automatically with a *solver* or with an *interactive proof assistant*;
- proof of all the VCs means that *all possible executions* of the software component satisfy the functional requirements under a set of *assumptions* on execution environment, development tools, etc.

A usual assumption is that the *machine code* (or *binary code*) generated by a *compiler* follows the formal semantics of the programming language defined by the verification framework. It would be reasonable if the compiler transformations were formally verified. Though there is ongoing research and development of such tools (a good example is CompCert [9]), the industry is still bound to high-end optimizing compilers, like GCC and Clang/LLVM. Unfortunately, they are too complex to be thoroughly verified, and bugs in the generated machine code are not uncommon [10].

As an alternative approach dismissing the unwarranted trust to a compiler, we propose to prove that the produced binary code still satisfies the functional properties expressed in the pre- and postconditions of the source code functions. The idea looks attractive because it should be much easier to check the correctness of one particular code transformation than to verify the entire compiler (in a sense, this is a test oracle that determines whether the compiler behavior is correct or not). Moreover, it makes it possible to enable aggressive optimizations that are unsafe in general but are acceptable for a given component and its functional contract. At the same time, there are a lot of difficulties to overcome:

- the target *instruction set architecture* (*ISA*) – the registers, the memory, the addressing modes, and the instructions – should be *formally specified* (there is no other way to reason about the machine code's semantics);

- the *high-level specifications* should be *adapted* to the binary code (in particular, one needs to find a correspondence between the variables in the source code and the registers and memory locations in the machine code);
- the *verification hints*, including loop invariants, ghost code, and, probably, lemma functions, should be *reused* at the binary code level or there should be an alternative way to provide them for the machine code;
- the tool should be capable of verifying functional properties of the resulting binary code in presence of arbitrary *compiler optimizations*.

The rest of this paper is organized as follows. Section 2 overviews the works addressing deductive verification of software components at the binary code level. Section 3 describes the proposed architecture of a machine code deductive verification system. Section 4 contains experimental evaluation of the suggested approach on the example of the `memset` library function being compiled to the RISC-V ISA. Finally, Section 5 concludes the paper and outlines future work directions.

II. Related work

In the Why3-AVR project [11], the Why3 platform [12] is applied to deductive verification of branch-free assembly programs for the AVR microcontrollers. The AVR ISA is formally specified in the WhyML language (it is supposed to be done manually). The WhyML syntax allows defining assembly instructions in a way that enables "reusing" AVR programs (simple preprocessing is enough for a program to become a valid WhyML text). A programmer is able to annotate assembly code with pre- and postconditions in WhyML and check its correctness using external solvers and proof assistants. The approach seems to be useful for low-level development as Why3 has rich capabilities for code analysis and transformation. Our tool (and methodology) is a bit different: it makes it possible to reuse source-code-level specifications at the binary code level and scales well to more complex ISAs as it uses ISA specifications in dedicated languages, e.g. nML [13] (such languages are called architecture description languages or instruction set specification languages). A crucially important distinction is that our approach supports loops in programs and, respectively, loop invariants in specifications.

In [14], the HOL4 proof assistant [15] is used to verify machine-code programs for subsets of ARM, PowerPC, and x86 (IA-32). The mentioned ISAs were specified independently: the ARM and x86 models [16], [17] were written in HOL4, while the PowerPC model [18] was written in Coq [19] (as a part of the CompCert project [9]) and then manually translated to HOL4. The author distinguishes four levels of abstraction. Machine code (level 1) is automatically decompiled into the low-level functional implementation (level 2). A user manually develops a high-level implementation (level 3) as well as a high-level specification (level 4). By proving the correspondence between those levels, he/she ensures that the machine code complies with the high-level specification. The advantage of the solution is that it allows reusing verification of proofs between different ISAs. Another thing to be noted is automatic translation of loops to recursive functions. In our opinion, the level of automation can be increased by using specialized architecture description languages.

An interesting approach aimed at verifying machine code against ACSL specifications [20] is presented in [21]. The workflow is as follows: (1) the ACSL annotations are rewritten as an inline assembly code; (2) the modified sources are compiled into the assembly language; (3) the assembly code is translated into WhyML; (4) the Why platform generates the VCs and discharges them with an external solver. The approach looks similar to the proposed one; however, there are tangible distinctions. The main of them is that the workflow involves a compiler: it implies that source code modifications may be required when switching one compiler to another. Also, verification at the assembly level does not allow abandoning the compiler correctness assumption as the assembly code is an intermediate form and needs further translation. Our goal is to make the verification tool as much compiler and machine independent as possible; the configuration should include only the data type sizes.

In [22], there have been demonstrated the possibility of reusing proofs of source code correctness for verifying the machine code. The approach is illustrated on the example of a Java-like source language and a bytecode target language for a stack-based abstract machine. The paper describes how to use such a technology in the context of proof-carrying code (PCC) and shows (in a particular setting) that non-optimizing compilation preserves proof obligations, i.e. source code proofs (built either automatically or interactively) can be transformed to the machine code proofs. Although the ideas of the approach may be useful, the problem we are solving is different. Moreover, the solution is tied to a specific platform.

III. Suggested Architecture

This section describes the suggested architecture of a machine code deductive verification system. The purpose is to verify the binary code of a function against the source-code-level specifications. The tool takes the following inputs:

- the verified *source code* of the function and its *specifications* (pre- and postconditions, loop invariants, etc.);
- the *non-optimized object code* of the function;
- the *optimized object code* of the function (a subject to verification);
- the *target ISA specification* (registers, addressing modes, instructions, etc.);
- the *compiler/machine configuration* (data type sizes and an application binary interface).

The tool output (report) contains the overall verdict (indicating whether the [optimized] binary code of the function is correct) and some auxiliary information including the verdicts for all the generated VCs. Fig. 1 depicts components required to build the system and how they interact each other. The subsections below describe each of the components in brief.

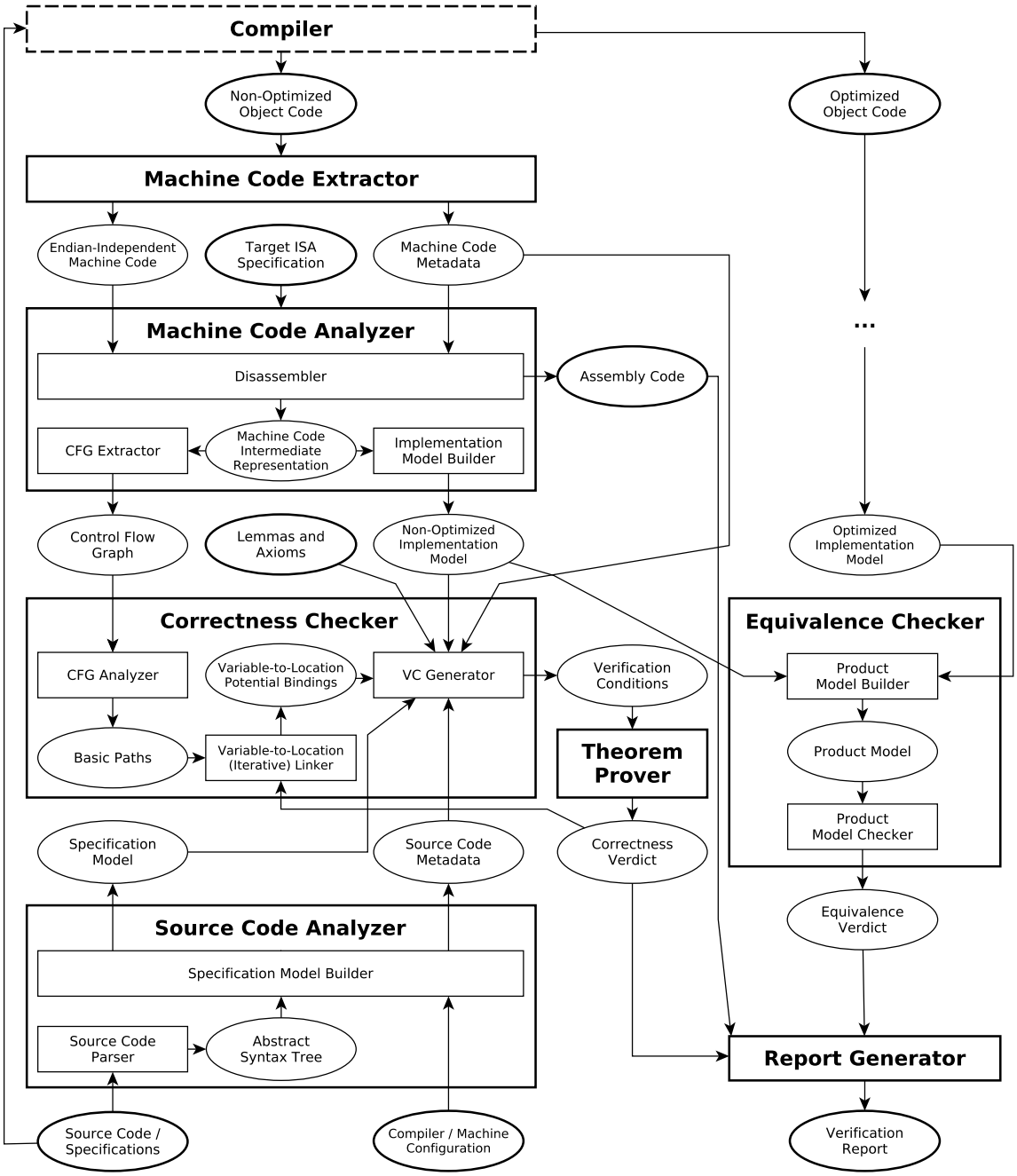

Fig. 1. The suggested architecture of a machine code deductive verification system

A. Machine Code Extractor

A *machine code extractor* is a simple tool that extracts the *endian-independent machine code* of the function from the given *object code*. An implementation relies on the object file format and can use existing utilities (e.g. GNU Binutils [23]). In addition to the machine code, the tool collects *metadata* including the function address table (with the starting addresses of the functions being called from the target one) and other useful information.

B. Machine Code Analyzer

Using binary code as-is limits the applicability of a verification system to a single ISA. A more flexible solution is to translate the *machine code* to an architecture-agnostic *intermediate representation* (*IR*), a sequence of instructions whose semantics is formally defined. A *disassembler* – a component that performs such kind of translation – can be a standalone tool implemented for a particular platform or be automatically constructed from the *target ISA specification* represented in

an architecture description language. The specification defines the microprocessor's registers, memory, addressing modes, and instructions. Besides the machine code IR, the disassembler produces the *assembly code*. While a specialized IR is considered to be a better choice for component integration, a human-readable assembly code is used to generate *verification reports* and to make sure that the disassembler works properly.

A *control flow graph* (*CFG*) *extractor* searches for branch instructions, resolves their targets, and splits the sequence of instructions into the basic blocks (BBs). Branches with unresolved targets and branches whose targets are out of the sequence range are considered to be external calls/returns. The extracted CFG is annotated with additional data gathered from the ISA specification, e.g. the branch conditions.

An *implementation model builder* translates the machine code IR to a logical form. Constructing the *implementation model* depends on the IR notation (and, indirectly, on the ISA specification formalism): if the language is formal enough [16]–[18], the IR itself may serve as a model; otherwise [13], an extra effort is required. In any case, the tool should formally represent all register and memory modifications done in the code. The output format is better to be well-established, such as SMT-LIB [24], HOL [15] or Coq [19], or to support translation into that kind of languages, e.g. WhyML [12].

C. Source Code Analyzer

A *source code parser* gets the *source code* of the function along with its *specifications* and produces the *abstract syntax tree* (*AST*). Usually, static analysis platforms allow developing custom plugins for source-to-source translation. A *specification model builder* – a component that maps the specifications to a logical form – can be implemented as such a plugin. It may happen that a plugin for an appropriate target language already exists; however, it is highly unlikely that that plugin is suitable for machine code verification. The specification model should take into account the *compiler/machine configuration* including sizes of platform-dependent data types. There are also *metadata* to be collected: the function arguments, the local variables, and the loop invariants. That information is used for generating *verification conditions* (*VCs*).

D. Correctness Checker

The main difference between the source and machine code correctness checking lies in the stage of VC generation. When verifying source code with the classic deductive verification approach, we generate the VCs independently from each other and then discharge them with a solver or an interactive proof assistant. However, information about the function's variables is lost during the compilation and cannot be restored directly. This fact makes it impracticable to reuse the high-level loop invariants within the classical VC generation scheme. Roughly speaking, we need to check all possible *bindings* between the source code's variables and the machine code's registers and memory locations. Assuming that k is the number of variables used in the loops and n is the number of locations, there are $k!\,C_n^k$ options to check.

To overcome the issue, the system uses a special component, called a *variable-to-location linker*, responsible for searching the correspondence between the variables and the locations. Before starting the linker, a *CFG analyzer* examines the CFG and extracts the *basic paths*, i.e. chains of BBs (or, more generally, acyclic subgraphs) that cover the CFG and connect the function/loop entry/exit points; thus, each basic path targets a loop invariant initialization, a loop invariant preservation, or the postcondition. The linker starts from the empty set of bindings and tries to iteratively solve the variable-to-location assignment problem. It applies heuristics to prioritize assignments and takes into account information about proved/disproved invariants to prune the search.

The core of a verification system is its *VC generator*. It constructs VCs for given *bindings* and passes them to a *theorem prover*. It requires a lot of data to generate correct VCs: the *implementation/specification models*, the *machine/source code metadata*, and, probably, some *lemmas and axioms*.

After all the VCs have been discharged, a *report generator* collects all the information about the verification process and produces a human-readable *verification report*.

E. Equivalence Checker

Verification of the *optimized machine code* is performed by checking its equivalence to the non-optimized one. Therefore, if the non-optimized version meets all the functional requirements, as verified by the *correctness checker*, then the optimized version also meets the criterion. This implies that verification of a compiler-optimized binary code requires the non-optimized counterpart. After proving the VCs for the non-optimized code, the tool tries to prove the equivalence of the two binaries. The approach, like many others [25], [26], is based on semantic alignment of the implementation models (programs) and construction of the product model (joint transfer graph). Though the *equivalence checker* handles compiler optimizations, it is compiler-independent and does not rely on any information provided by a compiler.

F. Theorem Prover

The *theorem prover* is an external component responsible for proving/disproving VCs. The main requirement is that it should support reasoning about bit vectors and bit-vector arrays. It is quite natural to model a microprocessor as follows: (1) the registers and memory locations are bit vectors; (2) the register files and memory units are bit-vector arrays; (3) the instructions are operations over bit vectors. The tool can be of one of the two types (or a combination of both): an automatic SMT solver or an interactive proof assistant. On the one hand, SMT solvers enable a fully automated verification process; on the other hand, there are situations when they are unable to give a definitive verdict. In such situations, interactive proof assistants may come in handy.

Module / IO Format	Implementation	Notes
Object Code	ELF [29]	A popular format for executables, object code, shared libraries, and core dumps
Machine Code Extractor	Based on Binutils [23]	Uses `readelf` to extract endian-independent machine code
Machine Code Metadata	Function address table	Contains relative addresses for all functions being called from the target one
Machine Code Analyzer	MicroTESK [30], [31]	Provides a number of tools for binary code analysis: the disassembler, the CFG extractor, and the implementation model builder
Target ISA Specification	nML [13]	Provides facilities for specifying assembly syntax, binary encoding, and semantics of microprocessor instructions and addressing modes
Machine Code IR	MIR	An inner MicroTESK representation
Assembly Code	Assembly language	Format is specified in the target ISA specification
Control Flow Graph	JSON	Describes basic block boundaries, links to the successors, and branch conditions
Implementation Model	SMT-LIB 2.6 [24]	Static single assignment form of the basic blocks
Source Code / Specifications	C / ACSL [20]	C code annotated with pre- and postconditions and loop invariants
Compiler/Machine Configuration	JSON	Sizes of the basic C types and the application binary interface
Source Code Analyzer	Frama-C [32] / Why3 [12]	Uses Frama-C as a frontend, a custom ACSL-to-WhyML translation plugin, and Why3 for WhyML-to-SMT-LIB translation
Specification Model	SMT-LIB 2.6 [24]	Pre- and postconditions, loop invariants, etc.
Source Code Metadata	JSON	Describes signatures of the generated SMT-LIB functions
Lemmas and Axioms	SMT-LIB 2.6 [24]	Auxiliary definitions to help SMT solvers
Correctness Checker	MicroVer [33]	The main part of the tool (see Section 3)
Verification Conditions	SMT-LIB 2.6 [24]	Check the loop invariant initializations/preservations and the postcondition
Theorem Prover	CVC4 [34]	A powerful SMT solver that supports bit vectors and bit-vector arrays
Correctness Verdict	Plain text	*Yes*, *no* or *unknown*: if *no*, a counter-examples is provided
Equivalence Checker	MicroTESK [31]	Constructs the product of two implementation models and verifies it deductively
Equivalence Verdict	Plain text	see *Correctness Verdict*
Verification Report	Plain text	Summarizes information: verdicts for all VCs and counter-examples (if necessary)

TABLE I
OUR IMPLEMENTATION OF THE MACHINE CODE DEDUCTIVE VERIFICATION SYSTEM

IV. Evaluation

In this section, we overview our implementation of the machine code deductive verification system and have a brief look at its application to the `memset` C library function [6] (naïve implementation) being compiled to the RISC-V ISA [27]. Table I shows information on the components and the input/output formats used in the system. Table II represents `memset`'s ACSL-annotated C code, assembly code, and binary code. The function has been successfully verified; the results (including the generated VCs and the tools to reproduce some steps) are available online [28].

V. Conclusion

The industry needs practical methods and tools for formal verification of software components. The majority of the existing solutions perform source-code-level analysis. However, it is not guaranteed that compilers are error-free; therefore, the most critical software requires verification at the binary code level. Developing a machine code verification system is a challenging and time-consuming task; for this reason, it is almost imperative to reuse existing verification and static analysis frameworks. In this work, we have identified a set of components required to build such a system and have shown a way how they may be composed together. We have selected appropriate engines from among existing software, supplemented them with the missing ones, and built a tool that is able to automatically verify machine code against the source-code-level ACSL specifications. It is worth noting that the approach is relatively independent of the target platform as it uses ISA specifications.

The work is in progress, and, certainly, many things are subjects to improvement. Future research directions are as follows. First, to complete the verification system, we should fully support the ACSL language. Second, the list of available ISAs has to be extended (to date, we have specified several popular microprocessor architectures, including RISC-V, ARM, MIPS, and, partially, Power). Third, we are working on industry-applicable techniques for equivalence checking of optimized and non-optimized machine programs. Finally, the tool requires more thorough assessment on a more representative benchmark (we have verified about 20 functions so far).

Acknowledgment

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).

References

[1] R.W. Floyd. *Assigning Meanings to Programs*. Mathematical Aspects of Computer Science. Proceedings of Symposia in Applied Mathematics, 19, 1967. P. 19-32. DOI: 10.1090/psapm/019/0235771.

[2] C.A.R. Hoare. *An Axiomatic Basis for Computer Programming*. Communications of the ACM, 12(10), 1969. P. 576-585. DOI: 10.1145/363235.363259.

[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), 32(1), 2014. P. 2:1-2:70. DOI: 10.1145/2560537.

[4] E. Cohen, W. Paul, S. Schmaltz. *Theory of Multi Core Hypervisor Verification*. SOFSEM 2013: Theory and Practice of Computer Science. Lecture Notes in Computer Science (LNCS), 7741, 2013. P. 1-27. DOI: 10.1007/978-3-642-35843-2_1.

[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, 82, 2014. P. 77-97. DOI: 10.1016/j.scico.2013.01.006.

ACSL-ANNOTATED C CODE	ASSEMBLY CODE	BINARY CODE

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

TABLE II
THE ACSL-ANNOTATED C CODE, THE ASSEMBLY CODE, AND THE BINARY CODE OF THE `memset` FUNCTION

[6] D. Efremov, M. Mandrykin, A. Khoroshilov. *Deductive Verification of Unmodified Linux Kernel Library Functions*. International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA). Lecture Notes in Computer Science (LNCS), 11245, 2018. P. 216-234. DOI: 10.1007/978-3-030-03421-4_15.

[7] D.R. Cok. *OpenJML: JML for Java 7 by Extending OpenJDK*. NASA Formal Methods (NFM). Lecture Notes in Computer Science (LNCS), 6617, 2011. P. 472-479. DOI: 10.1007/978-3-642-20398-5_35.

[8] A. Kamkin, A. Khoroshilov, A. Kotsynyak, P. Putro. *Deductive Binary Code Verification Against Source-Code-Level Specifications*. Tests and Proofs (TAP). Lecture Notes in Computer Science (LNCS), 12165, 2020. P. 43-58. DOI: 10.1007/978-3-030-50995-8_3.

[9] *CompCert Project* – http://compcert.inria.fr

[10] C. Sun, V. Le, Q. Zhang, Z. Su. *Toward Understanding Compiler Bugs in GCC and LLVM*. International Symposium on Software Testing and Analysis (ISSTA), 2016. P. 294-305. DOI: 10.1145/2931037.2931074.

[11] M. Schoolderman. *Verifying Branch-Free Assembly Code in Why3*. Verified Software. Theories, Tools, and Experiments (VSTTE). Lecture Notes in Computer Science (LNCS), 10712, 2017. P. 66-83. DOI: 10.1007/978-3-319-72308-2_5.

[12] J.-C. Filliâtre, A. Paskevich. *Why3 — Where Programs Meet Provers*. Programming Languages and Systems (ESOP). Lecture Notes in Computer Science (LNCS), 7792, 2013. P. 125-128. DOI: 10.1007/978-3-642-37036-6_8.

[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*. Theorem Proving in Higher Order Logics (TPHOLs). Lecture Notes in Computer Science (LNCS), 5170, 2008. P. 28-32. DOI: 10.1007/978-3-540-71067-7_6.

[16] A. Fox. *Formal Specification and Verification of ARM6*. Theorem Proving in Higher Order Logics (TPHOLs). Lecture Notes in Computer Science (LNCS), 2758, 2003. P. 25-40. DOI: 10.1007/10930755_2.

[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*. Principles of Programming Languages (POPL), 2006. P. 42-54, DOI: 10.1145/1111037.1111042.

[19] Y. Bertot. *A Short Presentation of Coq*. Theorem Proving in Higher Order Logics (TPHOLs). Lecture Notes in Computer Science (LNCS), 5170, 2008. P. 12-16. DOI: 10.1007/978-3-540-71067-7_3.

[20] 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.

[21] T.M.T. Nguyen, C. Marché. *Hardware-Dependent Proofs of Numerical Programs*. Certified Programs and Proofs (CPP), 2011. Lecture Notes in Computer Science 7086. P. 314-329. DOI: 10.1007/978-3-642-25379-9_23.

[22] G. Barthe, T. Rezk, A. Saabas. *Proof Obligations Preserving Compilation*. Formal Aspects in Security and Trust (FAST), 2005. Lecture Notes in Computer Science 3866. P. 112-126. DOI: 10.1007/11679219_9.

[23] *GNU Binutils* – https://www.gnu.org/software/binutils

[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*. Programming Languages and Systems (APLAS), 2017. Lecture Notes in Computer Science (LNCS), 10695. P. 127-147. DOI: 10.1007/978-3-319-71237-6_7.

[26] B. Churchill, O. Padon, R. Sharma, A. Aiken. *Semantic Program Alignment for Equivalence Checking*. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2019. P. 1027-1040. DOI: 10.1145/3314221.3314596.

[27] *RISC-V Foundation* – https://riscv.org

[28] *Results of memset binary code verification* – https://forge.ispras.ru/attachments/7472

[29] *Executable and Linkable Format (ELF))* – http://www.skyfree.org/linux/references/ELF_Format.pdf

[30] M. Chupilko, A. Kamkin, A. Kotsynyak, A. Protsenko, S. Smolov, A. Tatarnikov. *Test Program Generator MicroTESK for RISC-V*. International Workshop on Microprocessor and SOC Test and Verification (MTV), 2018. 6 p. DOI: 10.1109/MTV.2018.00011.

[31] *MicroTESK Framework* – http://www.microtesk.org

[32] *Frama-C Platform* – http://frama-c.com

[33] *MicroVer Project* – https://forge.ispras.ru/projects/microver

[34] *CVC4 Solver* – https://github.com/CVC4/CVC4