A Certifying Compiler for Clike Subset of C Language

Zhaopeng Li, Zhong Zhuang, Yiyun Chen, Simin Yang, Zhenting Zhang, and Dawei Fan
School of Computer Science and Technology
University of Science and Technology of China
Hefei, China
Email: {zpli, dyzz, simin, zztya, fandawei}@mail.ustc.edu.cn yiyun@ustc.edu.cn

***Abstract*—Proof-carrying code (PCC) is a technique that allows code consumers to check whether the code is safe to execute or not through a formal safety proof provided by the code producer. And a certifying compiler makes PCC practical by compiling annotated source code into low-level code and proofs. In this paper we present a certifying compiler for a subset of the C programming language, named Clike, with built-in automated theorem provers. Clike programs can be compiled by *ANSI C* compiler without any modification. Our compiler is intended to deal with data structures such as singly-linked lists, doubly-linked lists and trees. At the source level, we have designed a program logic combining a constrained first-order logic and a fragment of separation logic. We use a verification-condition-based method, and the generated verification conditions are sent to the built-in automated theorem prover. Our prover will generate proof terms when the input formula is valid. The low-level verification framework follows Hoare-style verification methods. The assembly code, its specification and proofs are generated automatically based on a variant of Stack-based Certifying Assembly Programming (SCAP). We implement our certifying compiler prototype in SML/NJ and build our prover libraries using the meta logic provided by Coq. We have used our prototype to successfully certify a considerable number of programs manipulating linked-lists and binary trees.**

***Keywords*-Certifying Compiler; Program Verification; Separation Logic; Theorem Prover; Proof-Carrying Code**

I. Introduction

Modern software is often extremely complicated and may contain many subtle bugs. Two typical examples are compilers and operating systems. Necula proposes *proof-carrying code* (PCC) [1] which allows code consumers to check whether the code is safe to execute or not through checking a formal safety proof provided by the code producer. PCC brings two grand challenges to the research field of programming languages. One is to explore more expressive program logics or type systems, so that the properties of high-level or low-level programs will be easily specified or reasoned about. The other is the research on certifying compilation [2], which explores how the compiler generates proofs for the compiled programs.

Many researchers today are focusing on methods to certify critical software and provide formal proof. One important breakthrough in operating system verification is *seL4* [3]. It provides a mathematical, machine-checked proof for the functional correctness of the *seL4* micro-kernel mainly written in the C programming language. But before running such a kernel, one must use a C compiler to compile the code. Obviously, the properties laboriously proved at the source level may be ruined in the target code if there is a single subtle bug in the C compiler.

It is ideal if we have a certified compiler at hand. But as we all know, many correct algorithms cannot be proved yet. It is nearly impossible to prove the full correctness of a realistic compiler. Leroy's *Compcert* [6], [7] is one of the certified optimizing compiler. One limitation is that *Compcert* is programmed in Coq [8], [9] and the executable compiler is obtained via automatic extraction of Caml code from Coq code, and thus we must trust the extraction process is free of bug. Moreover, many proofs must be redone if we want to involve a new optimization.

Certifying compiler is a different approach. It is easier to implement than a formal verification of the compiler. By compiling annotated source code into low-level code and proofs, certifying compilers can connect the source and target certification and make program verification more scalable and productive. Necula and Lee implement *Touchstone* certifying compiler [2]. It contains a traditional compiler for a small but type-safe subset of C and a certifier that automatically produces a proof of type safety for each assembly program produced by the compiler. The generated proofs show that the code is type safe and memory safe. In our previous work, we have designed and implemented a certifying compiler PLCC [12]. The weak point is that part of verification conditions must be proved by hand. Moreover, the source level program logic [13] is very complex.

We present in this paper a certifying compiler, namely *CComp*, for a subset of the C programming language with explicit memory allocation and deallocation. Our compiler intends to deal with data structures such as singly-linked lists, doubly-linked lists and trees. And the safety policy is much stronger than type and memory safety. We have designed a program logic combining a constrained first-order logic and a fragment of separation logic for the source language. We use a verification-condition-based method, and the generated verification conditions (VC for short) are proved by the built-in automated theorem prover. The low-level verification framework follows Hoare-style verification methods. The x86 assembly code, its specification and proofs are generated automatically based on a variant of SCAP [15]. The low-level proofs are constructed using pre-defined Coq tactics and templates, and partly by reusing the source-level VC proofs.

The main contributions of our approach are as follows:

- We integrate a built-in automated theorem prover in our certifying compiler. Our prover will generate proof-terms when the input formula is valid. These proof-terms are checkable by the proof assistant Coq. One sub-prover is for linear integer arithmetic based on decision procedure Simplex [27]. Its capability is comparable to the Coq tactic omega, but the size of proof generated by our prover is much smaller. Another sub-prover is for the separation logic fragment [22] which also produces Coq proof terms.
- Our code and proof generation is based on a realistic verification framework, a variant of SCAP. We have figured out methods to translate source-level specifications to low-level ones. Moreover, we have found ways to reuse source-level VC proofs in construction of low-level proofs. This work is the first attempt to make the SCAP technique more practical by using automated proving techniques.
- We have implemented a certifying compiler prototype. We used our prototype to certify a considerable number of programs manipulating linked-lists and binary trees.

This paper is organized as follows. In Section II we give an overview of the certifying compiler that we have implemented including the source language, assertion language, program logic and brief introductions on the source-level verification and the target verification framework. Then in Section III, the built-in automated theorem provers will be presented. Specification and proof translation/generation related topics will be discussed in Section IV. We give experimental results and evaluation of our certifying compiler prototype in Section V. Related works will be compared in Section VI. In the final section, we summarize and introduce our future work.

II. Our Certifying Compiler CComp

A. Overview

We choose a subset of C programming language, namely Clike, as the source language. We use C mainly for two reasons. One is that C is one of the most popular programming languages, especially in system software programming. The other is that C programs using pointers are not easy to be written correctly. The target language is *x86 assembly language* supported by the most popular computing platforms. The structure of *CComp* is shown in Figure 1. Programmers must annotate a Clike program with pre-/post-conditions and loop invariants (specifications for short). We use a verification-condition-based method at the source level.

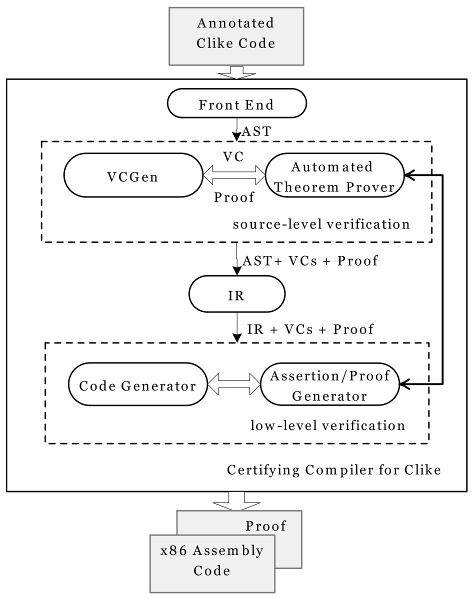

Figure 1. Structure of *CComp* Certifying Compiler

The VC generator (referred to as *VCGen*) implements a strongest post-condition calculation and produces VCs at certain specific program points. These VCs are sent to a built-in automated theorem prover. If any error is reported by our prover, the compilation is terminated showing the VCs cannot be proved.

If all VCs are valid, our prover will generate a machine-checkable proof (proof terms checkable by Coq). Following the VCGen phase, our compiler will generate assembly code and low-level specifications, and low-level proofs based on our low-level verification framework - a variant of SCAP. The generated code, specifications and proofs form a PCC package. Note that the low-level proofs mainly show that each basic block is well-formed with respect to its pre-/post-conditions. It is constructed partly by reusing the high-level VC proof terms and partly by using pre-defined proof templates and scripts designed for each instruction.

We do not implement our own proof checker. The PCC package produced by *CComp* can be checked by Coq.

B. Source-level verification

1) ***Annotated Source Language****:* Our prototype of a certifying compiler is for Clike. The representative syntax is defined in Figure 2.

In order to simplify reasoning about the safety properties of pointer programs, restrictions and assumptions are introduced on Clike. Pointer operations are restricted: pointer variables and constants can only be used in assignment, equality comparison, de-reference and as parameters of functions; the address-of operator and pointer arithmetic are not allowed. Just because of the latter restriction, a pointer

$$\begin{array}{rcl}
\textit{structdec} & ::= & \texttt{struct}\ \textit{ident}\{\textit{vardeclist}\}; \\
\textit{fundec} & ::= & \texttt{/*@}P\texttt{*/}\textit{type}\ \textit{id}(\textit{arglist})\ \textit{body}\ \texttt{/*@}Q\texttt{*/} \mid \ldots \\
\textit{arglist} & ::= & \textit{arglist}, \textit{type}\ \textit{id} \mid \textit{type}\ \textit{id} \\
\textit{type} & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{struct}\ \textit{id}* \\
\textit{body} & ::= & \{\textit{vardeclist}\ \textit{stmtlist}\} \\
\textit{stmtlist} & ::= & \textit{stmt}\ \textit{stmtlist} \mid \textit{stmt} \\
\textit{stmt} & ::= & \textit{lval} = \textit{exp}; \mid \textit{id}(\textit{explist}); \\
 & \mid & \textit{lval} = \textit{id}(\textit{explist}); \mid \texttt{free}(\textit{exp}); \\
 & \mid & \textit{lval} = \texttt{malloc(sizeof(struct}\ \textit{id}\texttt{));} \\
 & \mid & \texttt{if}(\textit{boolexp})\ \textit{block}\ \texttt{else}\ \textit{block}; \\
 & \mid & \texttt{for}(\textit{stmt}; \textit{boolexp}; \textit{stmt})\ \texttt{/*@}I\texttt{*/}\ \textit{block}; \\
 & \mid & \texttt{while}(\textit{boolexp})\ \texttt{/*@}I\texttt{*/}\ \textit{block}; \\
\textit{block} & ::= & \textit{stmt} \mid \{\textit{stmtlist}\} \\
\textit{explist} & ::= & \textit{explistx} \mid \varepsilon \\
\textit{explistx} & ::= & \textit{explistx}, \textit{exp} \mid \textit{exp} \\
\textit{exp} & ::= & \textit{number} \mid \texttt{NULL} \mid \textit{lval} \mid -\textit{exp} \\
 & \mid & \textit{exp} + \textit{exp} \mid \textit{exp} - \textit{exp} \mid \textit{exp} * \textit{exp} \mid \ldots \\
\textit{boolexp} & ::= & \texttt{true} \mid \texttt{false} \mid \textit{exp}\ \textit{rop}\ \textit{exp} \mid \ldots \\
\textit{rop} & ::= & \texttt{==} \mid \texttt{!=} \mid \texttt{>=} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{<} \\
\textit{lval} & ::= & \textit{id} \mid \textit{lval}\texttt{->}\textit{id} \\
\textit{number} & ::= & \ldots \mid \texttt{-1} \mid \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots
\end{array}$$

Figure 2. Representative Syntax of Clike Language

$$\begin{array}{llcl}
(\textit{assertion}) & A & ::= & A \vee A \mid (\Pi \mid \Sigma) \\
(\textit{Pure Formula}) & \Pi & ::= & \textit{true} \mid \textit{false} \mid \mathbb{B} \wedge \Pi \\
(\textit{Spatial Formula}) & \Sigma & ::= & \texttt{emp} \mid \mathbb{H} * \Sigma \\
(\textit{Boolean Pred.}) & \mathbb{B} & ::= & \neg\mathbb{B} \mid \mathbb{E}_1\ \textit{rop}\ \mathbb{E}_2 \\
(\textit{Heap Pred.}) & \mathbb{H} & ::= & \mathbb{E} \mapsto \mathbb{E} \mid \texttt{list}(\mathbb{E}) \\
 & & \mid & \mathbb{E} \mapsto \{id_1 : \mathbb{E}_1, \ldots, id_n : \mathbb{E}_n\} \\
 & & \mid & \texttt{lseg}(\mathbb{E}_1, \mathbb{E}_2) \mid \texttt{dlist}(\mathbb{E}) \\
 & & \mid & \texttt{tree}(\mathbb{E}) \\
(\textit{Assertion Exp.}) & \mathbb{E} & ::= & (\mathbb{E}) \mid \textit{number} \mid \textit{id} \mid \texttt{null} \\
 & & \mid & \texttt{res} \mid \texttt{old}(\textit{id}) \mid \mathbb{E}_1\ \textit{bop}\ \mathbb{E}_2 \\
(\textit{Operations}) & \textit{bop} & ::= & + \mid - \mid *
\end{array}$$

Figure 3. Representative Syntax of Assertion Language for Clike Program

can only point to the beginning address of a heap block dynamically allocated. Furthermore, `malloc` and `free` functions are considered as pre-defined functions in Clike and meet the basic requirements of safety. For example, every call to `malloc` can successfully return and the heap block allocated in this call does not overlap with any other heap block which has not been deallocated yet.

Note that, annotation begins with `/*@` and ends with `@*/`. It is compatible with the syntax of comment in *ANSI C*. So annotated Clike programs can be compiled by *ANSI C* compiler without any modification. In the annotations, P, Q, I are assertions which will be defined below.

*2) **Assertion and Specification Language**:* The main syntax of assertion language is shown in Figure 3 inspired by the Smallfoot project [24], but our assertion language is more expressive by supporting linear integer arithmetic and other predicates.

The assertions are $\vee$-combinations of basic formulae to express different cases at a program point. The basic formula consists of two parts : a pure formula (the pure part) and a spatial formula (the spatial part). Symbol "$\mid$" is used to separate the two parts in syntax; in semantics, it is equal to the logic conjunction. Pure formulae are used to specify the relations between stack variables, such as `x==1` $\wedge$ `y + 1 >= z` (suppose `x, y, z` are integer typed identifers). We use identifers not appearing in the program to express unexplicit existential logic variables. Identifier `res` is used to describe the return value of a function (`res` is treated as a keyword in Clike). And `old(id)` is used to denote the initial value of `id` at the function entry point. Currently the expressions are restricted since our prover can only support linear integer arithmetic.

Spatial formula describes the data heap using separating conjunction (*) [11] and *heap predicates*. `emp` is the predicate for empty heap. To describe heap cells, points-to relation $\mapsto$ can be used. E.g., if `p` points to an uninitialized structure whose type is defined as following:

```
struct list{ int data; struct list* next;};
```

Then it can be expressed using formula:

$$p \mapsto \{data : 0, next : NULL\}.$$

We support data structures like the singly-linked list (and its segment), the doubly-linked list (and its segment) and the tree by built-in inductive heap predicates: `list, lseg, dlist, dlseg, tree`. And we use classic definition of `lseg, tree, dlseg` from the separation logic. Note that, `list, dlist`, circular linked-list can be defined using list segment predicates. It is quite straightforward, so omitted here.

Specifications of functions are given in the form of pre-/post-conditions and loop invariants which are assertions.

*3) **Program Logic**:* Judgements of the program logic for Clike are in the form of Hoare triples. For example, the core inference rules for well-formed statements in triples are of the form

$$\vdash\ \{P\}\ \textit{stmtlist}\ \{Q\}.$$

The rules are extended and adapted from separation logic. Due to limited space, rules are omitted, interested reader can refer to our technique reports [14].

*4) **Verification Condition Generator**:* VCGen uses the inference rules to do a *strongest-postcondition* (forward) calculation. When doing such a calculation, VCGen will check at each program point whether the assertion is valid or not using the interfaces provided by our automated theorem prover. For example, if the assertion is

$$x < 0 \wedge y == x + 1 \wedge y > 1 \mid \Sigma,$$

the assertion will be reported to be *False* using the *checkPure* interface of the linear integer arithmetic prover. Also if the pure part and spatial part are inconsistent, the whole assertion results in invalidity. Such a mechanism will make our VCGen more effective.

Our approach generates VCs at several specific program points. Suppose there is a loop in a function with a return statement, three VCs will be produced : one at the loop entry, the second at the loop exit and the last one at the point of return statement. Each VC will be proved using the prover immediately. And if it is invalid, error will be reported and the compilation and verification process will terminate.

*5) **Automated Theorem Prover:*** We integrate an automated theorem prover in our certifying compiler. This prover will be used in both source-level and low-level verification. Because our goal is to generate PCC package, i.e., code with proofs, we design and implement the automated prover which can produce proof terms (terms of λ-calculus). Details will be introduced in Section III.

C. Low-level verification

The ultimate goal of our certifying compiler project is to implement a certifying compiler which is able to be used in certified system software development. Since another research group in our laboratory is using SCAP-based method to verify operating system kernels, we choose one similar framework in order to make our prototype usable by that group. For this prototype, the low-level verification framework we chose is a variant of SCAP (*vSCAP* for short). Once a program is compiled with proofs, it is guaranteed by *vSCAP* that the produced assembly code will execute without getting stuck. In this subsection, we will introduce the abstract machine, specification language, inference rules and its soundness. Together they form the low-level verification framework.

*1) **Abstract Machine:*** Currently, we focus on user code compiled by *CComp*, so self-modified code is not considered. That is, the memory resident by code will not change during program execution. So we separate this portion of memory from the data portion ($\mathbb{M}$) and denote it as code heap ($\mathbb{C}$). In Figure 4, the x86-style abstract machine called m86 is defined.

A world $\mathbb{W}$ of m86 includes code heap $\mathbb{C}$, machine state $\mathbb{S}$ and a basic block $\mathbb{I}$ in execution. Note that, there is no program counter (pc) in the abstract machine, we use this basic block to simulate it.

Machine state includes data heap $\mathbb{H}$, register file $\mathbb{R}$, flag register file $\mathbb{R}_f$ and a stack $\mathbb{K}$. The machine uses an explicit control stack (two registers bp, sp and a data list $\mathbb{D}$) to simplify stack operations and proofs.

Note that, some details of the design for simplification are listed as follows :

- Arithmetic overflow is not considered in this machine, so only compare instructions will change the flag registers.
- In the prologue, instruction `enter i` is used to create stack frame and allocate space for local variables. An instruction `leave` is used to restore old stack frame before function return.

The operational semantics of m86 is modeled as small-step transition between worlds [14].

(*World*)	$\mathbb{W}$	$::=$	$(\mathbb{C}, \mathbb{S}, \mathbb{I})$
(*Code Heap*)	$\mathbb{C}$	$::=$	$\{\mathtt{l} \leadsto \mathbb{I}\}*$
(*State*)	$\mathbb{S}$	$::=$	$(\mathbb{H}, \mathbb{R}, \mathbb{R}_f, \mathbb{K})$
(*Data Heap*)	$\mathbb{H}$	$::=$	$\{\mathtt{l} \leadsto \mathtt{w}\}*$
(*Stack*)	$\mathbb{K}$	$::=$	$(\mathtt{w}_{\mathtt{bp}}, \mathtt{w}_{\mathtt{sp}}, \mathbb{D})$
(*Stack Data*)	$\mathbb{D}$	$::=$	$\mathtt{w} :: \mathbb{D} \mid \mathtt{dnil}$
(*Register File*)	$\mathbb{R}$	$::=$	$\{\mathtt{r} \leadsto \mathtt{w}\}*$
(*Flag Reg. File*)	$\mathbb{R}_f$	$::=$	$\{\mathtt{flag} \leadsto \mathtt{b}\}*$
(*Register*)	r	$::=$	eax | ebx | ecx | edx
		|	edi | esi
(*Flag Register*)	flag	$::=$	zf | sf
(*Address*)	a	$::=$	(i) | i(r)
(*Instruction*)	c	$::=$	addir i, $\mathtt{r}_\mathtt{d}$ | movir i, $\mathtt{r}_\mathtt{d}$
		|	movrm $\mathtt{r}_\mathtt{s}$, a | movkr i, $\mathtt{r}_\mathtt{d}$
		|	pushr $\mathtt{r}_\mathtt{s}$ | popr $\mathtt{r}_\mathtt{d}$
		|	cmpi i, $\mathtt{r}_\mathtt{d}$ | cmpr $\mathtt{r}_\mathtt{s}$ $\mathtt{r}_\mathtt{d}$
		|	je l | jne l | jg l | jge l
		|	enter i | leave
(*Basic Block*)	$\mathbb{I}$	$::=$	c;$\mathbb{I}$ | jmp l | ret
		|	call f, l
(*Labels, Word*)	l, f, w	$::=$	i(*integer*)
(*Bit*)	b	$::=$	0 | 1

Figure 4. Abstract Machine m86 in vSCAP (Part)

(*Assertion*)	p	$\in$	$State \rightarrow Prop$
(*Guarantee*)	g	$\in$	$State \rightarrow State \rightarrow Prop$
(*Code Spec.*)	a	$::=$	(p, g)
(*Code Heap Spec.*)	Ψ	$::=$	$\{\mathtt{l} \leadsto \mathtt{a}\}*$

Figure 5. Specifications of vSCAP

*2) **Specification:*** The specification is defined in Figure 5. We use the meta-logic (*Prop*) of Coq as our assertion language.

Each basic block is specified by a pair of assertion and guarantee. Assertion p is a predicate taking the current state as a parameter which means the current state must satisfy the predicate p. Guarantee g takes two states as parameter: the current state and the state at the return point of the current function (the point right before the instruction `ret`, if it ever returns). So the guarantee is used to describe the relation between these two states.

*3) **Program Logic and Soundness:*** We use the following judgements to define inference rules:

$$\Psi \vdash \{\mathtt{a}\}\mathbb{W} \quad \text{(well-formed world)}$$
$$\Psi \vdash \mathbb{C} : \Psi' \quad \text{(well-formed code heap)}$$
$$\Psi \vdash \{\mathtt{a}\}\mathbb{I} \quad \text{(well-formed basic block)}$$

Due to space limit, please refer to our technical reports [14] to see the inference rules. We have proved the soundness of the rules with respect to operational semantics of m86. The Coq implementation of *vSCAP* is also available on our web site.

III. Built-in Automated Theorem Prover

A. Overview

We integrate into our compiler a built-in automated theorem prover with proof-term output. The structure is shown in

Figure 6. The input formula is in the form of an implication $A_1 \Rightarrow A_2$ which is generated by VCGen. The prover is designed to produce proof-terms when the formula is valid. There are two main sub-provers, one for linear integer arithmetic, and the other for separation logic fragment. And we leave other domain-specific provers for future work to support more logics.

The basic formula consists of two parts : pure formula and spatial formula. The pure formula and spatial formula accepted by the prover are defined by the following syntax (Pred is the name of built-in predicate, such as lseg, tree, and *etc*.):

$$\begin{array}{rcl}
\text{binop} & ::= & + \mid - \mid * \\
\text{relop} & ::= & = \mid \neq \mid > \mid < \mid \geq \mid \leq \\
\text{expression(E)} & ::= & \text{id} \mid \text{N} \mid \text{E } \textit{binop} \text{ E} \\
\text{number(N)} & ::= & 1 \mid 2 \mid \cdots \\
\text{pure-term(P)} & ::= & \text{E } \textit{relop} \text{ E} \\
\text{spatial-term(S)} & ::= & \text{E} \mapsto \text{E} \mid \text{Pred(E,}\cdots\text{,E)} \\
\Pi & ::= & \text{true} \mid \text{P} \mid \Pi \wedge \Pi \\
\Sigma & ::= & \text{emp} \mid \text{S} \mid \Sigma * \Sigma
\end{array}$$

The assertion language used in our prover is quite low-level. For example, the heap pointed by a Clike pointer will be partitioned to several separated sub-heaps. Each sub-heap is asserted by the pointer plus the offset (with respect to the Clike type system). E.g.,

$$p \mapsto \{data : 0, next : NULL\},$$

will be presented as follows:

$$p + 0 \mapsto 0 \;*\; p + 4 \mapsto 0.$$

In order to simplify the design of the sub-provers, the prover will generate a sequence of entailments ($\Pi \wedge \Sigma \vdash \Pi' \wedge \Sigma'$) according to the structure of VC, then call the sub-provers, and generate the proof terms for input VC if it is valid. For example, if the input VC is $\Pi_1 \wedge \Sigma_1 \vee \Pi_2 \wedge \Sigma_2 \Rightarrow \Pi'_1 \wedge \Sigma'_1 \vee \Pi'_2 \wedge \Sigma'_2$, our prover turns this VC into several entailments as follows:

- $\Pi_1 \wedge \Sigma_1 \vdash \Pi'_1 \wedge \Sigma'_1$,
- $\Pi_1 \wedge \Sigma_1 \vdash \Pi'_2 \wedge \Sigma'_2$,
- $\Pi_2 \wedge \Sigma_2 \vdash \Pi'_1 \wedge \Sigma'_1$,
- $\Pi_2 \wedge \Sigma_2 \vdash \Pi'_2 \wedge \Sigma'_2$.

They are sent to the separation logic prover one by one. If there exists one which can be proved valid, the input VC is valid; otherwise it is invalid.

B. Linear integer arithmetic prover

The Linear integer arithmetic prover serves as a key component for both the separation logic prover and the VC-Gen. It provides two features: checking implication between two given pure formulae and finding implicit equalities of variables in them. The prover will return both results and proof objects.

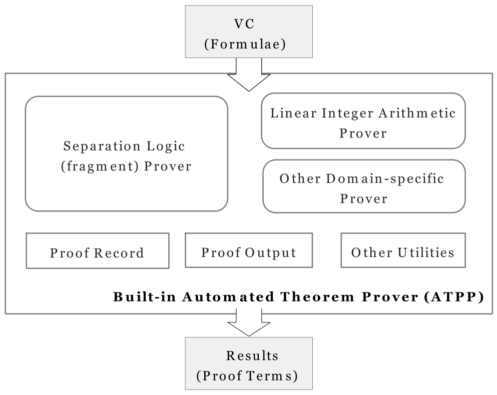

Figure 6. Structure of Built-in Automated Theorem Prover

*1) **Check Implication**:* The input is in the form of $\Pi_1 \vdash \Pi_2$, where Π_i is the conjunction of some integral expressions. We prove it by contradiction, that is $\Pi_1 \wedge \neg\Pi_2 \vdash False$. Here the *Simplex* algorithm is used to check the satisfiability of $\Pi_1 \wedge \neg\Pi_2$. If $\Pi_1 \wedge \neg\Pi_2$ is not satisfiable, $\Pi_1 \wedge \neg\Pi_2 \vdash False$ is true.

The first step is to normalize all the expressions to the form $const \leq exp$ according to the comparator. As we only concern about natural numbers in our memory model, we can avoid introducing δ in original *Simplex* and simply transform $x > n$ to $n+1 \leq x$. And we can detect the failed case $n < x < n+1$ in an early stage of our prover. The tricky part is how to deal with equality and in-equality:

$$\begin{array}{c}
\Pi_1 \wedge a = b \vdash \Pi_2 \Rightarrow \Pi_1 \wedge (a \leq b) \wedge (b \leq a) \vdash \Pi_2 \\
\Pi_1 \wedge a \neq b \vdash \Pi_2 \Rightarrow (\Pi_1 \wedge a < b \vdash \Pi_2) \vee (\Pi_1 \wedge b < a \vdash \Pi_2)
\end{array}$$

After the normalization step, we can build an initial context and use *Simplex* algorithm to check the satisfiability of current context and adjust it. This step is recursively called until a satisfied model is found or no more adjustment can be done.

*2) **Find Implicit Equalities**:* We collect all the expressions of comparator "=" and find implicit equalities:

$$\begin{cases}
n \leq x \leq n & x = n \\
x = n + z_1 \wedge y = n + z_2 \wedge z_1 = z_2 & x = y \\
x = p_1 + q_1 \wedge y = p_2 + q_2 & x = y \\
\quad \wedge\, p_1 = q_1 \wedge p_2 = q_2 &
\end{cases} \tag{1}$$

Currently, we do not allow predicates asserting on stack variables and their relations, thus we can bypass uninterpreted or interpreted functions here. So calculating congruence closure is quite straightforward.

*3) **Proof Objects**:* Proof objects generation in the linear integer arithmetic prover (*pure prover* for short) does not use the style of record-and-replay in our separation logic prover. The reason is that we find many redundant proof terms in the output objects and make them significantly large. In order to simplify the proof objects, we generate the objects in our pure prover as follow.

Within each step of our pure prover, we store proved sub-goals in a "proof library" and mark each sub-goal as a proof hole. When prover solved the problem, we get a bunch of proof objects and a proof tree of the original formula. The generation is to traverse this tree and fill all of these proof holes by finding the right proof objects in the "proof library".

For example, giving the hypotheses $\Pi : x \le z \wedge z \le y \wedge y \le x \wedge x{+}y \le z$, to prove $x = y$, we find the proof of $x = y$ in the "proof library" : $x \le y \wedge y \le x \rightarrow x = y$, so what we need is the proof of $x \le y$ and the proof of $y \le x$.Then we look up them in the library. They are $x \le z \wedge z \le y \rightarrow x \le y$ and one of the hypotheses, and $x \le z$ and $z \le y$ are also hypotheses. So we find a minimal proof while $x + y \le z$ is redundant.

$$\frac{\dfrac{\Pi \ \vdash x{\le}z \qquad \Pi \ \vdash z{\le}y}{\Pi \ \vdash x{\le}y} \qquad \Pi \ \vdash y{\le}x}{\Pi \ \vdash x{=}y}$$

C. Separation logic prover

The separation logic prover in our compiler aims to prove the formula in the form of $\exists x_1, x_2, \ldots, x_n, \Pi \wedge \Sigma \vdash \Pi' \wedge \Sigma'$ and generate machine-checkable proof objects. Π is the pure formula not involving objects in heap and Σ is the spatial formula indicating the allocated heap. $x_1, x_2, \ldots, x_n$ are existential variables in Π and Σ (we will make them implicit in the following text).

Currently, we support some built-in predicates in spatial formula including *lseg, list, cyclic-list, dlseg, dlist, tree*. Readers can refer to [11] for more detail.

The prover is inspired by Smallfoot [24]. A formula is processed by the prover in the following steps:

1) Simplify: Send each pair of expressions to our pure prover and get all the equality relations. Then rewriting is performed based on these relations.

For example:

$$p + 1 + 1 \mapsto v \vdash p + 2 \mapsto v$$

$p{+}1{+}1$ and $p{+}2$ are equal, and we will rewrite this formula to

$$p + 2 \mapsto v \vdash p + 2 \mapsto v$$

2) Unfold: Try to apply the following rules (some rules are omitted due to space limit) until no rules can be applied. This step tries to unfold predefined predicates.

- **Remove**. Remove predicates in the spatial formula that can be inferred by `emp`.

$$\frac{\Pi \wedge \Sigma \vdash \Pi' \wedge \Sigma'' \qquad \Pi \wedge \Pi' \vdash F \qquad F \wedge emp \vdash \Sigma'}{\Pi \wedge \Sigma \vdash \Pi' \wedge \Sigma' * \Sigma''}$$

- **Match**. Get equality entailment, and use the linear integer arithmetic prover to solve.

$$\frac{\Delta \vdash \Delta' \qquad \Delta \wedge \Delta' \vdash E_0 = E_1}{\Delta * E \mapsto E_0 \vdash \Delta' * E \mapsto E_1}$$

- **LSeg-Left**. Unfold Lseg from left.

$$\frac{\Delta * lseg(E_1, E_0) \vdash \Delta' \qquad \Delta \wedge \Delta' \vdash E \neq E_0}{\Delta * lseg(E, E_0) \vdash \Delta' * E \mapsto v * E + 4 \mapsto E_1}$$

- **List**. Unfold List.

$$\frac{\Delta * lseg(E, nil) \vdash \Delta' \qquad \Delta \wedge \Delta' \vdash E \neq nil}{\Delta * list(E) \vdash \Delta'}$$

Note that, some entailments in the rules, e.g. $\Delta \wedge \Delta' \vdash E \neq E_0$ in rule **LSeg-Left**, are determined by our linear integer arithmetic prover.

3) Clean Up: Remove predicates that are identical from the premise and the conclusion.

After these steps, we will finally get a formula in one of following forms:

- $emp \vdash emp$ This is what we want, and we can conclude the formula can be proved.
- $F \vdash emp$ This is a special state when we are doing frame inference, F is the *frame* we need.
- otherwise, the prover will give an error message to show the formula can not be proved.

To get machine checkable proof, we develop a separation logic library in Coq which includes a memory model (formalized heap), separation logic definitions, pre-defined predicates for supported data structures and all the related lemmas. The prover will record each step it takes during proof searching, and we directly build a Coq-checkable proof term.

For example, we have a Coq lemma :

$$example\_rule : forall\ F, P, Q,\ P \Rightarrow Q \ \rightarrow \ (P * F) \Rightarrow (Q * F).$$

and we recorded *example_rule F prem conc;...* during the proof search, then the proof term will look like:

$$example\_rule \ \ F \ \ P \ \ Q \ \ \_$$

P and Q can be inferred from the current premise and conclusion, _ is a proof hole and needs to be filled with proof objects generated by the remaining records of the prover.

IV. Generating Low-level Specifications and Proofs

According to our low-level verification framework, each program must be well-formed with respect to its specifications. That is, each basic block of the program should be well-formed under its pre-condition (we can use the pre-condition of the continuation as the post-condition, so post-condition is omitted). Our code generator produces x86 assembly codes as Coq data structures. The code can be easily extracted and turned into other well-known syntax. We do not discuss the extractor here.

Proof generation (including translation) is one of most important modules in our compiler. The first step in our low-level proof generator is to produce the specification environment Ψ, i.e., a map from labels to specifications.

Some specifications are translated from high-level ones, and others are calculated using the low-level rules. Then according to this environment, we generate one lemma and its proof (well-formedness proof) in Coq for each basic block. Since we can access the source-level VCs and their proof, we can reuse them to construct the low-level proofs.

In order to explain the whole process clearly, we classify program points into two categories; the first category includes the entry points, and exit points of functions and the entry points of loops, and we name them as annotated points. At these points in source level, user provides the annotations which can be translated to the low-level counterparts. All the other points belong to the second category (ordinary points).

A. Specification generation

Proof terms are complex and can not be read and understood by human easily. So it is not feasible to manipulate proof terms directly. We succeed in finding a new method which can shift the complexity of proof translation to specification translation. The less we change the specifications of the source level in the translation, the more likely we can reuse the source-level proofs.

In the source level, assertions at each program point are divided into pure formulae and spatial formulae. In the low-level machine model we also divide memory into two parts: the stack and the heap (The heap doesn't intersect with the stack). We view the stack as a linear table, which can be read and written using the offset to the current frame pointer `ebp`.

Since the low-level machine uses registers and the stack, the properties about registers and the location of every variable in the stack should be specified. We describe the contents in the stack by the name of variables and use an expression to describe the value of variables. The benefit is that the expression is the same as that in source level, and the spatial formula doesn't change much when it is translated from the source level. Since parameters are passed on the stack, the compiler always knows what the stack looks like at function entry and exit. This part of assertion for stack is easy to generate. For example,

$$x > 0 \wedge y = x + 1 \wedge q = NULL \mid p \mapsto \{data : 1, next : q\}$$

is a pre-condition of function using four parameters (namely `x`, `y`, `p` and `q`. The translated low-level pre-condition is

$$x > 0 \wedge y = x + 1 \wedge q = 0 \wedge (p + 0 \mapsto 1 * p + 4 \mapsto q)\ \mathbb{S}.\mathbb{H} \wedge\ \mathbb{S}.\mathbb{K} = (ebp, esp, ra :: x :: y :: p :: q :: \mathbb{D}).$$

Note that $\mathbb{S}, \mathbb{D}$, x, y, p, q, ebp, esp, ra are all auxiliary variables. $\mathbb{S}$ is the state, $\mathbb{S}.\mathbb{H}$ denotes the heap in the sate and $\mathbb{S}.\mathbb{K}$ denotes the stack part. $\mathbb{D}$ is the stack data which the function cannot read and write. ra is the return address on the stack top. The pure formula $x > 0 \wedge y = x + 1 \wedge q = NULL$ is directly used in low level. The spatial formula $p \mapsto \{data : 1, next : q\}$ is translated to a predicate $(p + 0 \mapsto 1 * p + 4 \mapsto q)$. It is the inner syntax of the prover. And the last part is generated to describe the stack using compilation information.

At the entry point of a loop, we get the properties of the stack by analyzing every possible entry of the loop to get the stack loop invariant. As for the registers, the register `eax` is used to hold the return value when function returns something to the caller. We do not care about other registers at the exit of functions. So at the exit of functions it equals to the variable `res` in the source-level assertion. That is, `R(eax) = res` is added to the low-level post-condition. Assertions for registers and the stack in ordinary points are calculated using the rules in *vSCAP*. The process in essence is a strongest post-condition calculation.

B. Proof generation

In the back-end, as an important component, our proof generator generates the proofs of the well-formedness of each program. According to the well-formed code heap rule (CDHP rule) of vSCAP, each basic block must be well-formed under its specification. For each basic block, we can using SEQ rule for sequence instructions and JMP rule for direct jump instruction and *etc*.

It is not hard for the generator to judge which rule should be used. The most difficult part is to generate proofs for the sub-goals after applying these inference rules. Taking SEQ rule as an example, after applying this rule, four sub-goals should be proved. The first sub-goal is to prove current instruction (c) is a sequence instruction (trivial). The second sub-goal is to prove the basic block removing current instruction is also well-formed (call generator again). The third is to prove $p\ \mathbb{S} \Rightarrow p'\ Next_c(\mathbb{S})$. Since p' is generated according semantic of current instruction c, it is just something like $A \Rightarrow A$ (trivial). For the last goal, it can be proved similarly.

We summarize regular the patterns and lemmas from manual proofs and define some tactics in Coq. The tactics we provided to construct the proof scripts successfully reduce the size of the generated proof by 10+ times compare with manual proofs. When we come across one instruction, we generate relevant proofs for the well-formedness of this instruction according to its assertion. If the instruction follows an annotated point or it is a function call, we divide the proof into two parts. The first part (VC in source level) can be proved by applying the relevant proof term. The second part (stack and register related) which is in the form of $A \Rightarrow A$, can always be proved by using the tactic *auto* in Coq.

All of the tactics are written in Coq, using the L-tac languages. Each of them consists of a series of Coq tactics and is compiled into a package. When generating the proof, our proof generator will use the tactics to prove some sub-goals and then prove the goal.

For example, given a basic block as follows:

```
enter 2        ;;alloc local variables
push ecx       ;;ecx on stack top
movkr 1 eax    ;;copy stack top to eax
leave          ;;free local variables
ret            ;;return to caller
```

To prove this basic block is well-formed, we must prove in turn that each instruction is well-formed. For each instruction, to prove the well-formedness, we write some scripts which will automatically match the current goal and use the right tactics to prove it. In this case, to prove instruction `enter 2` is well-formed, we use a tactic named "instruction_proof". For instruction `leave`, we use the "leave_proof" tactic and so on. Only a few tactics are needed to complete the whole proof. The output proof is very simple and can be automatically checked by Coq.

When we have proved the well-formedness of all basic blocks, we can prove the well-formedness of the code heap by using our pre-defined tactics.

V. Experiments and Evaluation

We implement our certifying compiler prototype in SML/NJ and build our prover libraries using the meta logic (CiC) provided by Coq. Table I shows the modules we have implemented and the lines of code for each module. Currently we have spent more than 3 man-years on this prototype and some features are still under active development. We continue to extend the range of programs being compiled and certified. In this section we present some early experimental results to answer the questions such as how large the safety proofs are, and how expensive proof checking is compared to certifying compilation.

We used our prototype to certify several programs manipulating linked-lists and binary trees (see Table II). Test case *Min* is a function returning the minimal argument. Test case *Cal* contains a function call to Min. And test case *List-alloc* and *List-insert* are programs manipulating singly-linked list. The proof size is huge comparing to the code size. And the program using spatial predicates will take long compilation time and checking time.

Module	Description	LoC
Front end, AST and ect.	SML/NJ code	1300
VCGen	SML/NJ code	1400
Integer arithmetic prover	SML/NJ code	3400
Separation prover	SML/NJ code	3200
Prover libraries	Coq definitions, lemmas and other scripts	4200
Spec. and proof generator	SML/NJ code	4000
SCAP (variant)	Coq definitions, lemmas, soundness proof and other	8000

Table I
TABLE OF CCOMP MODULES AND CODE SIZE

Test Case	Min	Cal	List-alloc	List-insert
Code Size(Byte)	183	285	643	1212
Proof Size(KB)	34	53	71	172
Compilation Time(ms)	<100	<100	2000	6000
Checking Time(s)	30	43	50	82

Table II
THE EXPERIMENTAL RESULTS OF SOME CCOMP TEST CASES

VI. Related Work

A. Certifying Compiler

Necula and Lee implement Touchstone certifying compiler [2] for a small but type-safe subset of C. It contains a traditional compiler producing optimized DEC Alpha machine code and a certifier that automatically produces a proof of type safety for each assembly program produced by the compiler. The generated proof shows that the code is type safe and memory safe. Comparing to this work, our certifying compiler accepts much larger subset of C language, and we use a logic system to specify the source code which is more expressive than type system. Our target code is x86 assembly code and along with the code there is a formal proof that the code confirms to its specification. The code and proof are formalized in Coq according to low-level verification framework, so it could be checked by Coq easily.

Colby *et al.* implement Special J [4], a certifying compiler for a large subset of Java. It compiles Java byte code into target code for Intel's x86 architecture. It can handle non-trivial run-time mechanisms such as object representation, dynamic method dispatch and exception handling. But the generated proof does not go beyond Java type safety. Moreover, at low-level there is a VCGen like Touchsone. It must be trusted to be implemented correctly. So the *trusted computing base* (TCB) is larger than our system. We have VCGen at the source level, but we have fundamental proof at the low level which exclude any modules such as VCGen or proof generator out of the TCB. What we trust are the low-level machine model (including its semantics), the source-level to low-level specification translation (this module is implemented in very small size of code, and it is easy to review them manually) and the proof checker (Coq).

Chen *et al.* design and implement a pointer logic [13] and a certifying compiler for PointerC (PLCC)[12]. As an extension of Hoare logic, the pointer logic expresses the change of pointer information for each statement in its inference rules to support program verification. The pointer logic differs from separation logic for concerning pointer aliasing. With this logic equipped, PLCC supports more expressive annotations including functional correctness. A simple theorem prover, which produces corresponding proofs for pointer-related VCs, is embedded into the compiler as well. The weakness is that integer-related VCs must be proved interactively in Coq. The compiler design, as well as the proof checking at assembly level, suffers from the inconsistency of the two kinds of VCs.

B. Verifiers

The Spec# static program verifier (code named Boogie) generates logical verification conditions from a Spec# program [19], [20]. Internally, it uses an automatic theorem prover that analyzes the verification conditions to prove the correctness of the program or find errors in it.

VCC is a verifier for Concurrent C [21]. It is a tool that proves correctness of annotated concurrent C programs or finds problems in them. VCC extends C with design by contract features, like pre- and post-condition as well as type invariants. Annotated programs are translated to logical formulae using the Boogie tool, which passes them to Z3 [18] to check their validity.

In their project, they check and verify source-level programs. The low-level code is no longer checked. Of course, these tools will help the programmers finding more bugs. But after verification, compiler will compile the programs into assembly code. So if the compilers do have bugs, what they verify laboriously at the source level will be quite useless and insufficient.

C. Theorem Proving Related

Simplify is a theorem prover for program checking [17]. It uses the Nelson-Oppen method to combine decision procedures for several important theories, and also employs a matcher to reason about quantifiers. It is used in the project ESC/Java which is a compile-time program checker that finds common programming errors.

Z3 [18] is a high-performance theorem prover being developed at Microsoft Research. Z3 supports linear real and integer arithmetic, fixed-size bit-vectors, and etc. Z3 is integrated with a number of program analysis, testing, and verification tools including: Spec#/Boogie[19], [20], and *etc*. As a contrast, our prover support less theories than Z3 but adding more theory support is our on-going work. Our prover differs from Z3 for generating Coq-checkable proof terms for valid formulae.

Berdine *et al.* proposes a decidable fragment of separation logic [22], [23] oriented to linked lists, and study decision procedures for validity of entailments. Then they implement *Smallfoot*, a tool for checking certain lightweight separation logic specifications [24]. The assertions describe the shapes of data structures rather than their detailed contents, and this allows reasoning to be fully automatic. Inspired by this project, we extend the assertion language and logic to support linear integer arithmetic, circular-linked list and other predicates. Moreover, our prover supports frame inference. When reasoning function call, VCGen will ask the prover to calculate a frame first. Using the frame rule and the frame, VCGen could proceed.

Distefano and Parkinson introduce a novel methodology for verifying a large set of Java programs in program verification [25]. It combines the idea of abstract predicate families and the idea of symbolic execution and abstraction using separation logic. They also implement an automatic verification system, called *jStar*. But it doesn't support linear integer arithmetic in their assertion language.

To summarize, these provers such as Simplify, Z3, Smallfoot and *jStar* are tools applying to automatic reasoning. As a contrast, our prover concerns both automation and machine-checkable proof.

D. Verification Framework

Feng *et al.* presents a simple but flexible Hoare-style framework for modular verification of assembly code with all kinds of stack-based control abstractions, including function call/return [15]. We change the machine model and some rules to support proof generation. The changes are straightforward but it helps greatly in easing the generation process. For example, the explicit stack makes specification translation and reusing high-level verification condition possible because we do not need to translate the stack part as memory predicate. If we use the original memory model where the stack and the heap are not separated, the proof concerning spatial formulae must be redone.

VII. CONCLUSION

This paper presents the design and implementation of a certifying compiler for a subset of C programming language supporting programs manipulating data structures such as linked lists and trees. It automatically produces a safety proof of the generated x86 assembly code according to its specifications. That is a PCC package based on a low-level verification framework which can be checked by the proof assistant Coq. And we are still working on this prototype to get more test cases passing through.

We explore methods to integrate an automated theorem prover into the compiler. The advantage of our prover is that proof-terms can be generated when the input formula is valid. Currently, our prover concludes sub-provers for separation logic fragment and for linear integer arithmetic. For future work, we want to add user-defined predicates support and integrate more domain-specific provers as needed.

And for the language and compiler, we plan to add more language features for Clike, such as type casting, union types, the address-of operator and pointer arithmetic. The influence of code optimization on certifying compiler designs is the future work to be considered. We also want to support loop invariant inference and hybrid data structures in our certifying compiler.

ACKNOWLEDGMENT

This research is based on work supported in part by grants from National Natural Science Foundation of China under

Grant No.90718026. Any opinions, findings and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] George C. Necula. Proof-Carrying Code. In POPL 1997: *The* 24^{th} *Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106-119, Paris, France, January 15-17, 1997.

[2] George C. Necula, Peter Lee. The Design and Implementation of a Certifying Compiler. In Proceedings of PLDI 1998, *the '98 Conference on Programming Language Design and Implementation*, Montreal, 1998.

[3] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, and *etc*.. seL4: Formal verification of an OS kernel. In Proceedings of the 22^{nd} *ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, October, 2009.

[4] Colby C, Lee P, Necula G C, et al. A Certifying Compiler for Java. ACM SIGPLAN Notices, 35(5): 95C107, 2000.

[5] Intel. Intel Architecture Software Devloper's manual. Intel Corporation, 2010.

[6] Xavier Leroy. Formal Certification of a Compiler Back-end, or: Programming a Compiler with a Proof Assistant. In Proceedings of the POPL 2006 : The 33^{rd} Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, January 11-13, 2006.

[7] Xavier Leroy. Formal Verification of a Realistic Compiler. Communications of the ACM, 52(7):107-115, 2009.

[8] The Coq Development Team. The Coq Proof Assistant Reference Manual. The Coq release v8.2, Feb. 2009.

[9] Y. Bertot and P. Castéran. Interactive Theorem Proving and Program Development C CoqArt: The Calculus of Inductive Constructions. EATCS Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[10] J. C. Reynolds. Separation logic: a Logic for Shared Mutable Data Structures. In Proceedings of *the* 17^{th} *Annual IEEE Symposium on Logic in Computer Science*, pages 55-74, July 2002.

[11] J. C. Reynolds. An Introduction to Separation Logic. Lecture notes available at http://www.cs.cmu.edu/ jcr/, Spring 2005.

[12] Yiyun Chen, Lin Ge, Baojian Hua, Zhaopeng Li, Cheng Liu and Zhifang Wang. A Pointer Logic and Certifying Compiler. Frontiers of Computer Science in China, 1(3):297-312, July, 2007.

[13] Yiyun Chen, Baojian Hua, Lin Ge and Zhifang Wang. A Pointer Logic for Safety Verification of Pointer Programs. Chinese Journal of Computers, 31(3):372-380, March, 2008.

[14] Zhaopeng Li, Zhong Zhuang, Yiyun Chen, Simin Yang, Zhenting Zhang, and Dawei Fan. Technique reports for Project CComp, available at Url http://kyhcs.ustcsz.edu.cn/ssl/ccomp/, May. 2010.

[15] Xinyu Feng, Zhong Shao, Alexander Vaynberg, Sen Xiang, and Zhaozhong Ni. Modular Verification of Assembly Code with Stack-Based Control Abstractions. In Proceedings of *the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 401-414, Ottawa, Canada, 2006.

[16] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In Proceedings of FM 2006: *International Symposium on Formal Methods*, volume 4085 of Lecture Notes in Computer Science, pages 460-475. Springer, 2006.

[17] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson,James B. Saxe, Raymie Stata. Extended Static Checking for Java. In Proceedings of *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, Jun 17-19, 2002, Berlin, Germany.

[18] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In Proceedings of *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Budapest, Hungary, 2008.

[19] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte.The Spec# Programming System: An overview. In CASSIS 2004, LNCS vol. 3362, Springer, 2004.

[20] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In FMCO 2005, LNCS vol. 4111, Springer, 2006.

[21] Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies, A Practical Verification Methodology for Concurrent Programs, no. MSR-TR-2009-15, February 2009.

[22] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A Decidable Fragment of Separation Logic. In Proceedings of FSTTCS 2004: *Foundations of Software Technology and Theoretical Computer Science*, 24^{th} International Conference, Chennai, India, December 16-18, 2004.

[23] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic Execution with Separation Logic. In Proceedings of APLAS 2005: *Programming Languages and Systems,* 3^{rd} *Asian Symposium*, Tsukuba, Japan, November 2-5, 2005.

[24] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In Proceedings of *Formal Methods for Components and Objects,* 4^{th} *International Symposium*, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005.

[25] Dino Distefano, Matthew J. Parkinson J. jStar: Towards Practical Verification for Java. In Proceedings of the 23^{rd} *ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, Pages 213-226, Nashville, TN, USA, 2008.

[26] Moore J S. Piton: a Mechanically Verified Assembly-Language. Norwell: Kluwer Academic Publishers, 1996.

[27] B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In CAV06, LNCS 4144, pages 81-94. Springer-Verlag, 2006.