Sound Symbolic Linking in the Presence of Preprocessing

Gijs Vanspauwen and Bart Jacobs

iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium
{gijs.vanspauwen,bart.jacobs}@cs.kuleuven.be

Abstract. Formal verification enables developers to provide safety and security guarantees about their code. A modular verification approach supports the verification of different pieces of an application in separation. We propose symbolic linking as such a modular approach, since it allows to decide whether or not earlier verified source files can be safely linked together (i.e. earlier proven properties remain valid).
If an annotation-based verifier for C source code supports both symbolic linking and preprocessing, care must be taken that symbolic linking does not become unsound. The problem is that the result of a header expansion depends upon the defined macros right before expansion.
In this paper, we describe how symbolic linking affects the type checking process and why the interaction with preprocessing results in an unsoundness. Moreover, we define a preprocessing technique which ensures soundness by construction and show that the resulting semantics after type checking are equivalent to the standard C semantics. We implemented this preprocessing technique in VeriFast, an annotation-based verifier for C source code that supports symbolic linking, and initial experiments indicate that the modified preprocessor allows most common use cases. To the extent of our knowledge, we are the first to support both modular and sound verification of annotated C source code.

Keywords: modular program verification, verification of C programs, C preprocessor

1 Introduction

One of the means to create safe and secure software is the formal verification of source code. Formal verification allows a developer to prove certain properties of his source code, so that he in turn can rigorously provide guarantees to the users of his software. There are many different tools available to verify source code. Tools that allow to show arbitrary properties of code, require some hints to be provided. Indeed, the validity of arbitrary properties of code is undecidable. For many verifiers these hints must be provided as annotations added to the source code. We only consider annotation-based verifiers [1–5] in this paper.

Proving security properties of code is most often a daunting task. A modular verification approach allows a developer to concentrate on the security sensitive parts of his application. We propose symbolic linking as such a modular approach.

After having verified different source files in isolation, symbolic linking allows to show whether or not the source files can be safely (i.e. earlier proven properties remain valid) linked together into an application without having to reverify them. As far as we can see, we are the first to report on a both modular and sound verification technique for annotated C source code. To support sound modular verification with symbolic linking, we modified the lexical and semantical analysis phases of verification. However, we show that the resulting semantics of these modifications are equivalent to the standard C semantics.

The first modification impacts the type checking procedure of the semantical analysis phase. Symbolic linking requires that for each C source file there is a header file containing forward declarations that describe the functionality from the source file. Besides that, it also requires that the verification of a single source file is performed with the assumption that the annotations in the header files of other source files are valid. Since header files can textually include each other using the C preprocessor, this last requirement implies, as explained further on, that the type checking procedure of the verifier must be recursive to type check every header file in isolation.

The second modification is concerned with lexical analysis, in particular preprocessing. Allowing the full functionality of the C preprocessor during verification renders symbolic linking unsound. Indeed, the result of including a header file depends on the context (i.e. the set of defined macros) at the point of inclusion. This can trick symbolic linking into thinking that earlier verified files can be safely linked together, while it is not safe to do so. A possible solution is to modify the behavior of the C preprocessor: if a header inclusion is encountered during preprocessing, expand it with an empty set of defined macros so that a header is always expanded in the same way. This resolves the verification hazard, but this context-free preprocessor differs from the normal C preprocessor. This can be solved by running both preprocessors in parallel and signaling an error if they diverge. To define this parallel preprocessor, precautions must be taken to support header files that use a macro as guard to prevent double inclusions.

In Section 2, we discuss symbolic linking and recursive type checking in more detail. The unsoundness problem caused by preprocessing is clarified in Section 3. Our solution to this problem is presented in Section 4 and in Section 5 we show that for the verification of a given source file, the semantics of recursive type checking after parallel preprocessing are equivalent to the standard C semantics. We implemented symbolic linking and the parallel preprocessing technique in VeriFast [4], an annotation-based verifier for single- and multi-threaded C programs. The implementation and some findings of initial experiments are described in Section 6. Finally, we end this paper with a discussion on related work and some conclusions in Section 7.

2 Symbolic Linking

Verifying an existing application with an annotation-based verifier is a nontrivial task. If a lot of source code must be annotated to prove correctness of a small

piece of security-sensitive functionality, the verifier is not modular and verification would become unmanageable for applications with a large code base. A modular verifier allows the verification of smaller pieces of code in isolation.

To build a modular application, a linker can be used to combine different compiled object files into an executable program. Symbolic linking is the verification counterpart of this compilation stage. Instead of object files, manifest files are created during verification. These manifest files describe the essential contents of the source files from the point of view of the verifier. During the symbolic linking process the manifest files are inspected to decide whether or not the original verified source files can be safely linked together.

This process imposes some conditions on the structure of the source files. First, we describe these conditions. Then we elaborate on the contents of the generated manifest files and how they are used by the symbolic linking process. We finish our discussion on symbolic linking with the necessity of a recursive type checking process for the verifier.

2.1 Source File Structure

An abstract view of a source file is one in which the file simply declares some program elements and these declarations may use program elements from other files. Symbolic linking requires that these dependencies are explicitly recorded in interfaces. In the context of the C language, for each source file a header file is used as its interface. The function headers of the functions implemented in a C source file are recorded in its corresponding header file together with annotations. Without going into the details of the VeriFast annotation language, here is an example of a C source file `abs.c` implementing the function `abs()`, which returns the absolute value of its only argument, together with its interface `abs.h`:

abs.c

```
#include "abs.h"

int abs(int x)
//@ requires true;
//@ ensures 0 <= x ?result == x:
                  result == 0 - x;
{ if(0<=x) { return x;}
  else  {return 0 - x;} }
```

abs.h

```
#ifndef ABS_H
#define ABS_H

int abs(int x);
//@ requires true;
//@ ensures 0 <= x ?result == x:
                  result == 0 - x;
#endif
```

The verification of another source file that uses the function `abs()`, has to include the header `abs.h` (as would be necessary for compilation too) so the verifier can find the annotation. That source file must then be verified with the assumption that the annotation found in the header `abs.h` is true.

2.2 Manifest Files

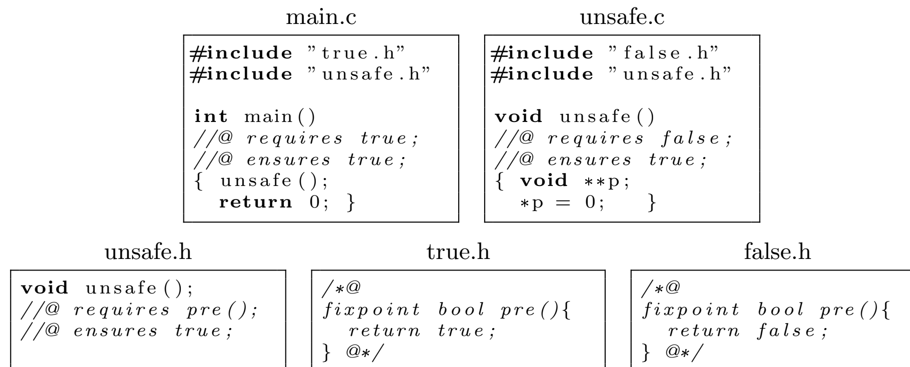
The implementation in a source file is thus verified with the assumption that annotated declarations (without implementation) in interface files are valid. During verification of a source file, all the declarations that were used but not

implemented by the file, should be recorded in the manifest. These *requires*-records from the manifest file contain the name of the interface where the declaration was found and the name of the declaration itself. Besides the *requires*-records, there are also *provides*-records included in the manifest. These records describe all implemented constructs from the source file, together with the name of the interface where the construct was declared. Of course the verifier must check that an annotation of a declaration in an interface conforms with the annotation of its implementation in the source file.

The symbolic linking process uses the manifest files to decide if the earlier verified files can be safely linked together. The process checks that there exists some matching *provides*-record for each *requires*-record in the manifest files of the earlier verified files. This ensures that all necessary functionality is correctly implemented somewhere. Due to the annotations in the interface, the linking process knows that a declaration has the same meaning in all source files (ignoring preprocessing for now) and thus earlier proven properties remain valid.

2.3 Recursive Type Checking

An important issue was ignored when describing manifest files and the symbolic linking process. For the verification of C source code with VeriFast, header files (i.e. interfaces) can contain auxiliary constructs (inductive data types, pure functions over these data types, predicates, . . .) for specifying annotations. These constructs can then be used in the annotations in the rest of the header file as well as in the source file. Of course a header file can also use annotation constructs from another header. If this is the case, it must be made sure that the first header file includes the second. Otherwise the semantics of an inclusion depends upon the type checking context in which the header file was included and thus its meaning can be different for different source files that included it. The following unsound example illustrates this requirement:



The `fixpoint` functions from `true.h` and `false.h` specify pure functions that can be used in other VeriFast annotations and can be considered to be synonyms for `true`, respectively `false` in the rest of this example. The file `unsafe.h` is an interface containing the forward declaration of the function

`unsafe()` annotated with a contract. In file `unsafe.c`, the header file `unsafe.h` is included after including the file `false.h`. The expansion of `unsafe.h` will be a forward declaration of the function `unsafe()` with the same contract as its implementation in `unsafe.c`. Again without going into the details of the annotation language of VeriFast, the contract in `unsafe.c` trivially holds since the precondition is false. So verification succeeds even if there clearly is a memory violation in the function `unsafe()`. A function with `false` as a precondition may never be used of course or verification will fail. Verification of the function `main()` in `main.c` also succeeds since by including the file `true.h` before including `unsafe.h`, the precondition of `unsafe()` is trivially satisfied (i.e. it is simply `true`) so the function may be used anywhere. These files verify correctly in isolation, but they should not be compiled together into an application. It is clear from this example that the semantics of the forward declaration of the function `unsafe()` from the included header `unsafe.h` is dependent upon previous includes.

A way to ensure that includes are independent from previous includes, is to type check each (directly or indirectly) included header file recursively in isolation: type check an included header with an empty set of declarations before using its declarations to type check the file that included the header. If an included header file is well-typed in isolation, we know it includes all the necessary constructs for the semantics of its contents. In our previous example the recursive type checking of `unsafe.h` will fail, since the pure function `pre()` is not defined there. To ensure that this kind of type checking preserves the semantics of a language, the language must exhibit the following property which we consider an axiom for the C language:

Axiom 1
A declaration that is well-typed according to two sets of type checked declarations, has the same semantics relative to the two sets if one is a subset of the other.

Since in the C programming language a declaration cannot be hidden by a subsequent declaration in the same scope and we only allow includes at global scope (this follows from the definitions of a preprocessor tree in Section 4 and the fact that a proper prefix of a declaration in C is itself not a valid declaration), the C language has this property.

3 Unsoundness Caused by Preprocessing

In the context of the C programming language, header files are used as interfaces for symbolic linking and these have to be included in a source file by using the preprocessor. However, an annotation-based verifier for C source code that supports symbolic linking, cannot allow full C preprocessor functionality[1] without becoming unsound.

The problem is that the C preprocessor performs textual inclusion and also allows to define textual macros. The earlier described symbolic linking process

[1] C11 - ISO/IEC 9899:2011: standard for the C programming language

implicitly assumes that interfaces contain the same annotations for different source files being verified. But the result of a header file inclusion, depends on the context (i.e. defined macros) at the point of inclusion. We clarify this problem by an example and end this discussion with possible alternative solutions.

3.1 Unsound Example

The unsound example presented here is quite similar to the one from Section 2. Consider the following files annotated for verification with VeriFast:

main.c

```
#define PRE true
#include "unsafe.h"

int main()
//@ requires true;
//@ ensures true;
{ unsafe();
  return 0; }
```

unsafe.c

```
#define PRE false
#include "unsafe.h"

void unsafe()
//@ requires false;
//@ ensures true;
{ void **p;
  *p = 0;    }
```

unsafe.h

```
#ifndef UNSAFE_H
#define UNSAFE_H

void unsafe();
//@ requires PRE;
//@ ensures true;

#endif
```

The file `unsafe.h` is an interface containing the forward declaration of the function `unsafe()` annotated with a contract. In file `unsafe.c`, the header file `unsafe.h` is included after defining the preprocessor symbol `PRE` to `false`. After preprocessing, the expansion of `unsafe.h` will be a forward declaration of the function `unsafe()` with the same contract as its implementation. So verification `unsafe.c` of will succeed since the precondition is false. Verification of the function `main()` in `main.c` also succeeds since by defining preprocessor symbol `PRE` to `true` before including `unsafe.h`, the precondition of `unsafe()` is trivially satisfied. Like before, these files verify correctly in isolation, but they should not be compiled together into an application.

Unfortunately, the symbolic linking process concludes that they can be safely linked together. This is clear from the manifest files generated for `main.c` and `unsafe.c`, i.e. `main.vfmanifest`, respectively `unsafe.vfmanifest`:

main.vfmanifest

```
.requires .\unsafe.h#unsafe
.provides main : prelude.h#main()
```

unsafe.vfmanifest

```
.provides .\unsafe.h#unsafe
```

The manifest file for `unsafe.c` only contains a *provides*-record for function `unsafe()` and the manifest file for `main.c` only contains a *requires*-record for the same function and a *provides*-record for the function `main()`. Since all required functionality is provided, symbolic linking decides that the files can be linked together.

The problem of combining preprocessing with symbolic linking, was shown here for the contract of a simple function inside a header file. The same problem can emerge if macro symbols are used for function names, function parameters or other parts of declarations.

3.2 Alternative Solutions

There are different possible solutions for the unsoundness problem introduced by preprocessing, each with their advantages and disadvantages. One solution could be to include annotations after preprocessing in the manifest files and check during symbolic linking that corresponding annotations are identical. However, in VeriFast annotations can be specified using inductive data types, primitive recursive pure functions over these data types and predicates. For this solution to work, these construct also have to be included in the manifest file and will make it bloated.

Another possible solution would be to reverify the source files during symbolic linking. In many cases this solution is unacceptable (e.g. it deteriorates modularity) or even impossible (e.g. linking with a library when only the header files of that library are available and not the source code).

Finally, the solution presented in the rest of this paper makes use of a modified (context-free) preprocessor. This context-free preprocessor does the trick by processing each included header with an empty set of defined macros. Thus the inclusion is not dependent on the context in which the include occurs (i.e. context-free). The context-free preprocessor should then be executed in parallel with the normal preprocessor and if their outputs diverge, an error is reported. This ensures that a correct execution of the resulting parallel preprocessor is context-free and compliant with the normal C preprocessor.

4 Preprocessing for Sound Symbolic linking

Here, we describe our solution to the unsoundness problem. First, we formalize the preprocessor by describing its behavior with a set of inference rules. Then, based on this formalization, a parallel preprocessing process is explained that resolves the unsoundness by construction. We conclude this discussion of our solution with a formal definition of the resulting semantics of the verification process as compared to a normal compilation process.

4.1 Preprocessing Formalized

Before presenting our solution, it is instructive to formalize the behavior of the preprocessor. Using the unspecified sets W and H we define in Definition 1 a token which represents the contents of a source file. Representing a source file by a single token simplifies the definitions that follow. A token can be a list of words

Definition 1.
$w \in W$ *and* $h \in H$
$t \in T ::=$
$| \; \bar{w} \quad | \; \textbf{\textit{def}} \; w \; \bar{w}$
$| \; t \; t \quad | \; \textbf{\textit{undef}} \; w$
$| \; h \quad | \; \textbf{\textit{ifdef}} \; w \; t \; \textbf{\textit{else}} \; t \; \textbf{\textit{endif}}$

Definition 2.
$m \in \mathrm{H} \rightarrow \mathrm{T}$

Definition 3.
$d \in \mathrm{W} \rightharpoonup \mathrm{W}^*$

Definition 4.
$\tau ::=$
$| \; []$
$| \; w :: \tau$
$| \; (h, \tau) :: \tau$

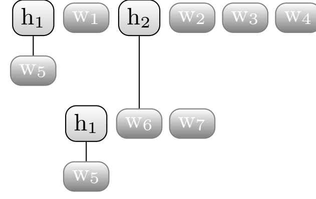


Fig. 1: Graphical representation of the preprocessor tree $[(h_1, [w_5]), w_1, (h_2, [(h_1, [w_5]), w_6, w_7]), w_2, w_3, w_4]$

($\bar{w} \in \mathrm{W}^*$), a sequence of other tokens ($t\ t$) or a preprocessor directive. In this simplified setting there are directives for header inclusion (h), macro definition ($\mathbf{def}\ w\ \bar{w}$), macro removal ($\mathbf{undef}\ w$) and conditional compilation ($\mathbf{ifdef}\ w\ t\ \mathbf{else}\ t\ \mathbf{endif}$). During preprocessing, a header map m is used to retrieve the contents of header files and the partial function d is used to remember the defined macros. The output of the preprocessor is a preprocessor tree τ, which is in fact a list of words augmented with the original include structure of the source file as illustrated in Fig. 1.

The behavior of the C preprocessor can then be captured by the (incomplete) inference rules rules in Definition 5 using big-step semantics. The complete set of rules can be found in a technical report [6]. Only the rules for macro definition, macro expansion and header expansion are shown here. The judgment $m \vdash (d, t) \Downarrow (d', \tau)$ defined by Definition 5, indicates that, given a certain header map m, the formal preprocessor will accept a token t and a set of defined macros d and it returns a resulting preprocessor tree τ and an updated set of defined macros d'.

In rule P-DEFINE the function update is used to add the defined macro to d. Since a premise in the rule P-WORDS-UNDEFINED states that w is not in the domain of d, the word is just copied to the resulting preprocessor tree. In the rule P-WORDS-DEFINED on the other hand, w is in the domain of d and its expansion $\bar{w}$ is preprocessed before adding it to the resulting preprocessor tree. The domain

Definition 5. *Inference rules for preprocessing*

$$\frac{}{m \vdash (d, \mathbf{def}\ w\ \bar{w}) \Downarrow (d[w := \bar{w}], [])}\ \textsc{P-define}$$

$$\frac{w \notin \mathit{dom}(d) \qquad m \vdash (d, \bar{w}) \Downarrow (d, \tau)}{m \vdash (d, w :: \bar{w}) \Downarrow (d, w :: \tau)}\ \textsc{P-words-undefined}$$

$$\frac{\begin{array}{c} w \in \mathit{dom}(d) \\ d(w) = \bar{w}_1 \qquad m \vdash (d, \bar{w}_2) \Downarrow (d, \tau_2) \\ m \vdash (d|_{\mathit{dom}(d) \setminus \{w\}}, \bar{w}_1) \Downarrow (d|_{\mathit{dom}(d) \setminus \{w\}}, \tau_1) \end{array}}{m \vdash (d, w :: \bar{w}_2) \Downarrow (d, \tau_1\ \tau_2)}\ \textsc{P-words-defined}$$

$$\frac{m \vdash (d, m(h)) \Downarrow (d', \tau)}{m \vdash (d, h) \Downarrow (d', [(h, \tau)])}\ \textsc{P-header-exp}$$

restriction of a function is used there to indicate that the expansion of a macro is preprocessed without the macro itself as a defined macro. Finally, the problematic rule P-words-exp describes how a header is expanded. It is clear from this rule that the header is preprocessed with the current partial function of defined macros d before the expansion is returned.

4.2 Parallel Preprocessing

As mentioned before, it is the context-dependency of the inclusion of a header in rule P-header-exp that renders symbolic linking unsound. To overcome this problem we define the context-free preprocessor. The context-free preprocessor works exactly the same as the normal preprocessor except that when it encounters an include directive, the expansion of the included file is calculated by a recursive call with an empty set of defined macros. This is done by replacing the rule P-header-exp with the rule CFP-header-cf-exp from Definition 6 (for the complete set of rules see [6]), which results in the definition of the judgment $m \vdash (d, t) \Downarrow_{cf} (d', \tau)$. Since the defined macros at the point of inclusion are the only source of variability in the resulting expansion, the context-free preprocessor always expands an included file in the same way. So the context of defined macros does not influence the result of an inclusion. Note that the macros that are defined during preprocessing of the header file, are added to the preexisting macros.

Definition 6. *Inference rule for context-free header expansion*

$$\frac{m \vdash (\varnothing, m(h)) \Downarrow_{cf} (d', \tau)}{m \vdash (d, h) \Downarrow_{cf} (d \cup d', [(h, \tau)])} \text{ CFP-header-cf-exp}$$

The context-free preprocessor clearly behaves differently from the normal preprocessor, but its context-freeness ensures that symbolic linking is sound. However, the compliance to the C standard of the normal preprocessor is also required. Our solution is to run both preprocessors in parallel and signal an error if they diverge. Care must be taken with the inclusion of header files that protect themselves from double inclusions by using a macro as guard (i.e. guarded headers). Since the inference rule CFP-header-cf-exp calls the preprocessor recursively with an empty set of defined macros, the macro guarding a header file is never defined at that point during preprocessing. Thus the second time a guarded header is included, it is expanded anyway by the context-free preprocessor. The normal preprocessor will not expand the second include of that guarded header. So a naive parallel preprocessing technique, would fail here.

To make parallel preprocessing succeed for guarded headers, we remove the secondary occurrences of header includes during context-free preprocessing. Only thereafter are the produced normal preprocessor tree and context-free preprocessor tree checked for equality during parallel preprocessing. This is safe to do, since the context-free preprocessor always expands a header to the same parse tree. To formalize this we first need the function I_h that does nothing more than collecting all the header names that occur in an outputted preprocessor

tree τ or in a set $\bar{\bar{h}}$ containing header nodes (i.e. ordered pairs of header names and preprocessor trees).

Definition 7. *Function I_h collects header names*

$$\begin{aligned} I_h([]) &= \varnothing \\ I_h(b :: \tau) &= I_h(\tau) \\ I_h((h, \tau_h) :: \tau) &= \{h\} \cup I_h(\tau_h) \cup I_h(\tau) \\ &\textit{and} \\ I_h(\bar{\bar{h}}) &= \bigcup_{\tilde{h} \in \bar{\bar{h}}} I_h([\tilde{h}]) \end{aligned}$$

Having defined the function I_h, we now can specify the function RSO which removes secondary occurrences from a preprocessor tree. As shown in Definition 8, this function expects a preprocessor tree τ and a set of already encountered header names $\bar{h}$. Only the case for $\tau = (h, \tau_h) :: \tau_r$ and $h \notin \bar{h}$ is worth discussing. Secondary occurrences from the tree τ_h are removed first and only then are they removed from τ_r. Note that the header names encountered while processing τ_h must be added to the set of already encountered headers names $\bar{h}$ before processing τ_r. This is the reason for adding the set $\mathrm{I}_h([(h, \tau_{h'})])$ to $\bar{h}$ before RSO is called recursively.

Definition 8. *Function RSO removes secondary occurrences*

$$\begin{aligned} \mathit{RSO}([], \bar{h}) &= [] \\ \mathit{RSO}(b :: \tau_r, \bar{h}) &= b :: \mathit{RSO}(\tau_r, \bar{h}) \\ \mathit{RSO}((h, \tau_h) :: \tau_r, \bar{h}) &= (h, []) :: \mathit{RSO}(\tau_r, \bar{h}) && (\textit{if } h \in \bar{h}) \\ \mathit{RSO}((h, \tau_h) :: \tau_r, \bar{h}) &= \mathbf{let}\ \tau_{h'} = \mathit{RSO}(\tau_h, \bar{h} \cup \{h\})\ \mathbf{in} && (\textit{if } h \notin \bar{h}) \\ &\quad\ (h, \tau_{h'}) :: \mathit{RSO}(\tau_r, \bar{h} \cup I_h([(h, \tau_{h'})])) \end{aligned}$$

Finally, we can formalize the parallel preprocessing technique. Let the judgment $m, t \blacktriangleright \tau_p, \tau_{cfp}$ as defined in Definition 9 indicate that parallel preprocessing succeeded and produced the normal preprocessor tree τ_p and the context-free preprocessor tree τ_{cfp} for a specific token t and header map m (and an empty set of defined macros). Thus the implementation of the parallel preprocessing technique (see Section 6) must ensure that if it was successful, then $\tau_p = \mathrm{RSO}(\tau_{cfp}, \varnothing)$ holds. Notice that this means that we do not support unguarded headers.

Definition 9. *Semantics of parallel preprocessing*

$$\forall\, m, t, \tau_p, \tau_{cfp}.\ \ m, t \blacktriangleright \tau_p, \tau_{cfp} \Leftrightarrow \exists\, d_p, d_{cfp}.\ \begin{cases} m \vdash (\varnothing, t) \Downarrow (d_p, \tau_p) \ \wedge \\ m \vdash (\varnothing, t) \Downarrow_{cf} (d_{cfp}, \tau_{cfp}) \ \wedge \\ \tau_p = \mathit{RSO}(\tau_{cfp}, \varnothing) \end{cases}$$

4.3 Resulting Semantics

Our sound approach to modular verification we call symbolic linking requires a modified (i.e. context-free) preprocessing phase and a modified semantic analysis phase (i.e. recursive type checking). While the behavior of each phase separately

differs from the C standard, we prove in Section 5 that their combined semantics are equivalent to the semantics defined by the C standard.

In order to state the soundness theorem for our approach, we need a way to specify the semantics of normal compilation and the semantics of a verification process with parallel preprocessing and recursive type checking. For this reason we introduce the following concepts:

- a declaration block ($b \in W^*$):
  a list of declarations that is not interrupted by an include directive
- a type checked declaration block ($b_{tc} \in W^* \times E$):
  an ordered pair of a declaration block and its type checking environment
- a type checking environment ($e \in E = \mathbb{N}_0^{W^* \times E}$):
  a multiset of declaration blocks

Strictly speaking E is the smallest set for which $E = \mathbb{N}_0^{W^* \times E}$ holds. Now we can express the semantics of compilation and verification in terms of a type checking environment. We will discuss these in turn.

Compilation. Let the function CP from Definition 10 represent the normal type checking procedure in the compilation process. The input to CP is a normal preprocessor tree τ and a current global type checking environment e_g, and the output is the resulting type checking environment which represents the semantics of the source file during compilation. This resulting type checking environment is a multiset containing all the declaration blocks found in the preprocessor tree together with the environment in which they are to be type checked. For a one-pass compile language like C every declaration is type checked given all the previous encountered declarations. This is the reason that e_g is called the global environment. So the rule for $\mathrm{CP}(b :: \tau, e_g)$ in Definition 10 correctly includes the previous global environment as the type checking environment of the encountered declaration block b. Note that b is used here to implicitly indicate the longest match of consecutive words in the tree that is not interrupted by an include.

Definition 10. *Function CP computes semantics of compilation*

$$\begin{aligned} CP([], e_g) &= e_g \\ CP(b :: \tau, e_g) &= CP(\tau, e_g \uplus \{|(b, e_g)|\}) \\ CP((h, \tau_h) :: \tau, e_g) &= CP(\tau, CP(\tau_h, e_g)) \end{aligned}$$

Verification. The function VF specified in Definition 11 (and named VF_s in [6]) represents the recursive type checking process of a verifier that supports our solution for sound symbolic linking. The output of VF is again a type checking environment and represents the semantics of the corresponding source file as seen by the verification process.

In contrast to CP, VF does recursive type checking and so the type checking environment e_d expected by VF is not global. It only contains declarations directly declared in the current expansion. Besides a context-free preprocessor tree τ and a direct type checking environment e_d, the function VF also expects

Definition 11. *Function VF computes semantics of verification*

$$\begin{aligned} VF([], \bar{\tilde{h}}_t, e_d) &= e_d \\ VF(b :: \tau, \bar{\tilde{h}}_t, e_d) &= \textbf{let}\ e := e_d \uplus MH(\bar{\tilde{h}}_t)\ \textbf{in} \\ &\quad\ VF(\tau, \bar{\tilde{h}}_t, e_d \uplus \{|(b, e)|\}) \\ VF(\tilde{h} :: \tau, \bar{\tilde{h}}_t, e_d) &= VF(\tau, \bar{\tilde{h}}_t \cup I_\tau([\tilde{h}]), e_d) \end{aligned}$$

Definition 12. *Function I_τ collects header nodes*

$$\begin{aligned} I_\tau([]) &= \varnothing \\ I_\tau(b :: \tau) &= I_\tau(\tau) \\ I_\tau((h, \tau_h) :: \tau) &= \{(h, \tau_h)\} \cup I_\tau(\tau_h) \cup I_\tau(\tau) \\ &\textit{and} \\ I_\tau(\bar{\tilde{h}}) &= \bigcup_{\tilde{h} \in \bar{\tilde{h}}} I_\tau([\tilde{h}]) \end{aligned}$$

a set of transitively encountered header nodes $\bar{\tilde{h}}_t$. In this set the occurrences of included headers are collected together with their transitive includes. As for e_d, this multiset only contains headers from the current expansion. To calculate the transitive includes of the encountered header in the rule for VF($\tilde{h} :: \tau, \bar{\tilde{h}}_t, e_d$), the function I$_\tau$ from Definition 12 is used before adding the result to $\bar{\tilde{h}}_t$.

The rule for VF($b :: \tau, \bar{\tilde{h}}_t, e_d$) does all the work to get the correct recursive type checking environment for type checking the encountered declaration block. The recursive type checking environment is the direct type checking environment together with all the declaration blocks occurring in $\bar{\tilde{h}}_t$. But the declaration blocks from $\bar{\tilde{h}}_t$ must be type checked before they are to be added to the type checking environment of the encountered declaration block. So the auxiliary function MH from Definition 13 is used in the let expression of rule VF($b :: \tau, \bar{\tilde{h}}, e_d$) to calculate these type checked declaration blocks from $\bar{\tilde{h}}_t$. Function MH simply calculates the resulting type checking environment of all the header nodes in $\bar{\tilde{h}}_t$ using the function VF. These type checking environments are then merged and the final resulting type checking environment is returned.

Definition 13. *Function MH merges header nodes into type checking environment*

$$MH(\bar{\tilde{h}}) = \biguplus_{(h,\tau) \in \bar{\tilde{h}}} VF(\tau, \varnothing, \varnothing)$$

5 Proof of Equivalence

In the previous section we formalized the semantics of compilation and verification with sound symbolic linking. We now must make sure that their semantics are equivalent. Otherwise a successful verification would be meaningless, since the verification is then performed on a semantically different program. So we need a way to compare type checking environments which represent the semantics of compilation and verification.

Comparing type checking environments can be done using the two mutually recursive judgments from Definition 14 and Definition 15. The (asymmetric) judgment $\gtrapprox$ from Definition 14 means equivalence between two type checking

Definition 14. *Equivalence of type checking environments*

$$\frac{}{\varnothing \succapprox \varnothing}\ \textsc{Env-eq-empty}$$

$$\frac{e_1 \succapprox e_2 \qquad e_{11} \succeq e_{21}}{\{|(b, e_{11})|\} \uplus e_1 \succapprox \{|(b, e_{21})|\} \uplus e_2}\ \textsc{Env-eq-not-empty}$$

Definition 15. *Subsumption of type checking environments*

$$\forall\, e_1, e_2.\ (e_1 \succeq e_2 \Leftrightarrow \exists e_3. e_1 \succapprox e_2 \uplus e_3)$$

environments. Clearly two empty environments are equivalent. If two environments are equivalent, adding a type checked declaration block to each of them where the type checking environment of the first subsumes the one of the second as defined by the judgment $\succeq$ from Definition 15, preserves this equivalence.

To see why the judgment from Definition 14 indeed implies that equivalent type checking environments have the same semantics according to the C language, note that the C language has the property mentioned in Axiom 1: declarations can not be hidden by subsequent ones.

If we can prove that (when parallel preprocessing succeeds) for a preprocessor tree τ_p generated from a specific source file by the normal preprocessor and a preprocessor tree τ_{cfp} generated form the same file by the context-free preprocessor, the semantics of $CP(\tau_p, \varnothing)$ are the same as that of $VF(\tau_{cfp}, \varnothing, \varnothing) \uplus \mathrm{MH}(\mathrm{I}_\tau(\tau_{cfp}))$, we know that the verification has the same semantics as compilation. This main property of our approach is expressed in Theorem 1.

Theorem 1. *Soundness theorem*

$$\forall\, m, t, \tau_p, \tau_{cfp}.\ \ m, t \blacktriangleright \tau_p, \tau_{cfp} \ \ \Rightarrow\ CP(\tau_p, \varnothing) \succapprox VF(\tau_{cfp}, \varnothing, \varnothing) \uplus MH(I_\tau(\tau_{cfp}))$$

We proved this theorem by first showing the validity of Lemma 1 from which Theorem 1 can be straightforwardly deduced. The proof of Lemma 1 is omitted here for space reasons but can be found in [6].

Lemma 1. *Main lemma*

$$\forall\, \tau_p, \tau_{cfp}, e_g, e_d, e_o, \bar{\bar{h}}_t, \bar{\bar{h}}_o.\ \begin{cases} \bar{\bar{h}}_t = I_\tau(\bar{\bar{h}}_t) \wedge \\ \bar{\bar{h}}_o = I_\tau(\bar{\bar{h}}_o) \wedge \\ (\forall h, \tau_h.\ \ (h, \tau_h) \in I_\tau(\tau_{cfp})\ \Rightarrow (h, \tau_h) \notin I_\tau(\tau_h)) \wedge \\ (\forall h, \tau_1, \tau_2.\ \ ((h, \tau_1) \in I_\tau(\tau_{cfp}) \wedge \\ \qquad\qquad (h, \tau_2) \in I_\tau(\tau_{cfp}) \cup \bar{\bar{h}}_t \cup \bar{\bar{h}}_o) \Rightarrow \tau_1 = \tau_2) \wedge \\ \tau_p = RSO(\tau_{cfp}, I_h(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o)) \wedge \\ e_g \succapprox e_d \uplus MH(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o) \uplus e_o \end{cases}$$

$$\Rightarrow\ CP(\tau_p, e_g) \succapprox VF_s(\tau_{cfp}, \bar{\bar{h}}_t, e_d) \uplus MH(\bar{\bar{h}}_t \cup \bar{\bar{h}}_o \cup I_\tau(\tau_{cfp})) \uplus e_o$$

6 Implementation

The recursive type checking procedure as represented by the function VF from Subsection 4.3, was already implemented in VeriFast to support symbolic linking.

When performing recursive type checking in that implementation, the problem that occurs due to the removal of secondary occurrences of guarded headers by the C preprocessor, was solved by preprocessing, parsing and type checking all header files in isolation. Only then are the declarations a header contains added to the type checking environment of the file that included the header. The unsoundness introduced by preprocessing was addressed originally by only allowing includes and header guards, but nothing else of the capabilities of the C preprocessor.

The parallel preprocessing technique from Subsection 4.2 was straightforward to implement in VeriFast. An implementation of the C preprocessor and the context-free preprocessor are run in parallel and an error is reported if their outputs diverge. If a single header is included many times, the function VF is not very efficient. For every declaration block that needs the header for type checking its declarations, the function VF is recursively called for that header through the function MH. In the actual implementation the result for each header is remembered, so when it is needed again, it does not have to be recomputed. Another issue in the implementation was the use of lemma functions. To make sure during symbolic linking that these functions are correctly implemented, lemma functions are also recorded in the manifest files and their termination is ensured.

Since the verification process itself did not have to be updated, the necessary modifications were nicely isolated. Only the preprocessing stage and the type checking stage of the verifier had to be updated. Initial tests with the modified verifier, show we support most common use cases of the C preprocessor. To support this claim, these are the use cases we currently support: the use of header guards, using macro definitions as constants and enumerations, and using macros for abbreviating repetitive code. Although not supported in the current implementation, we can extend it to support parameterized headers. This can be done by introducing a new preprocessor directive that states which macros are the parameters of a header. The definitions of these macros at the point of inclusion must then be recorded in the manifest files during verification for an equality check during symbolic linking to ensure context-freeness. A theoretical foundation for this approach still has to be developed.

7 Related work and conclusion

There are several annotation-based verifiers available for C source code including Microsoft's Verifying C Compiler (VCC) [1], the Escher C Verifier [2] from Escher Technologies, the work of Claude Marché et al. resulting in the Caduceus [5] tool, and the Frama-C [3] platform and its plug-ins (e.g. WP [3] and Jessie [3]).

Microsoft states on its website that VCC is sound and modular. VCC indeed allows the verification of files in isolation, but the problem of linking earlier verified files together is not mentioned. Since the C preprocessor can be used before verification, a header file can have a different meaning for different include sites. However, there is no way to determine if properties of source files earlier

proven by VCC remain valid if they are linked together in an application. So this seams to break modularity. The Escher C Verifier, the Caduceus tool and the Frama-C platform do not claim to be modular. The Frama-C platform does let you verify source files in isolation, but requires all the source files to be presented together if an entire application is to be soundly verified.

As for as we can see, no other verifier for C source code supports both modular verification and a mechanism for determining whether or not earlier proven properties remain valid when source files are linked together. The modular verification approach we implemented in VeriFast (i.e. symbolic linking with parallel preprocessing), does support this by limiting the capabilities of the preprocessor and these limitations are quite permissive. Moreover, we proved that the resulting semantics are equivalent to the standard C semantics; a property which is necessary when deviating from the C standard. Since our solution only impacts the lexical and semantical analysis phases, it is a valid candidate for implementation in other verifiers.

Acknowledgements. The research leading to these results has received funding from the European Union Seventh Framework Programme [FP7/2007-2013] under grant agreement n°317753, and more precisely from the STANCE project (a Source code analysis Toolbox for software security AssuraNCE).

This research is also partially funded by the EU FP7 project NESSoS, the Interuniversity Attraction Poles Programme Belgian State, the Belgian Science Policy, and by the Research Fund KU Leuven.

References

1. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 23–42, Berlin, Heidelberg, 2009. Springer-Verlag.
2. D. Crocker and J. Carlton. Verification of C programs using automated reasoning. In *Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods*, SEFM '07, pages 7–14, Washington, DC, USA, 2007. IEEE Computer Society.
3. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: a software analysis perspective. In *Proceedings of the 10th international conference on Software Engineering and Formal Methods*, SEFM '12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag.
4. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: a powerful, sound, predictable, fast verifier for c and java. In *Proceedings of the Third international conference on NASA Formal methods*, NFM '11, pages 41–55, Berlin, Heidelberg, 2011. Springer-Verlag.
5. Y. Moy and C. Marché. Inferring local (non-)aliasing and strings for memory safety. In *Heap Analysis and Verification (HAV'07)*, pages 35–51, Braga, Portugal, 2007.
6. G. Vanspauwen and B. Jacobs. VeriFast: Sound symbolic linking in the presence of preprocessing. CW Reports CW638, Department of Computer Science, KU Leuven, 2013.