A portable compiler-integrated approach to permanent checking

Nic Volanschi (`nic.volanschi@free.fr`)
my**gcc**

Abstract. Program checking is now a mature technology, but is not yet used on a large scale. We identify one cause of this gap in the decoupling of checking tools from the everyday development tools. To radically change the situation, we explore the integration of simple user-defined checks into the core of every development process: the compiler. The checks we implement express constrained reachability queries in the control flow graph taking the form "from x to y avoiding z", where x, y, and z are native code patterns containing a blend of syntactic, semantic and dataflow information. Compiler integration enables continuous checking throughout development, but also a pervasive propagation of checking technology. This integration poses some interesting challenges, including tight bounds on the acceptable overhead, but in turn opens up new perspectives. Factorizing analyses between checking and compiling improves both the efficiency and the expressiveness of the checks.

Minimalist user properties and language-independent code pattern matching ensure that our approach can be easily integrated in any compiler for any language. We illustrate this approach with a full-fledged checking compiler for C. We demonstrate the need for permanent checking by partially analyzing two different releases of the Linux kernel.

Keywords: extensible compilers, user-defined checks

1. Introduction

Checking programs with respect to user-specified properties is an important aspect of automated software engineering. Recent years have seen many advances in software checking materialized in the apparition of many different checking tools performing various levels of checks, ranging from purely syntax checks [7, 31, 1, 22, 13, 11], going through lightweight model checking [20, 30, 10, 14, 3, 8, 26, 21] up to sound software model checking [12, 4, 23]. Despite this apparently very encouraging trend, the use of these tools in everyday software practice is still marginal. Face to this situation, a legitimate question is: why?

There are several possible reasons of this discrepancy, among which:

– efficiency: most of the tools are not fast enough for everyday use. This is especially the case for more formal verifiers.

– usability: the learning curve might be too steep for some tools; some other tools are not sufficiently easy to use even for trained users

- integration: most of these tools are not integrated with familiar development tools

As a consequence of these and maybe other reasons, checking tools are not used at all in most software projects. At best, checking is performed sporadically. However, occasional checking has two major shortcomings. Firstly, errors are not instantly caught, meaning that they may be found in later stages when fixing them is more expensive. Secondly, errors that were fixed in some phases of the project may be re-introduced later on.

This paper explores a pragmatic approach to code checking aiming to incorporate some minimal amount of checking *throughout* the development process. Our approach to permanent checking is based on extending the central tool of the development process, the compiler, with user-defined properties that are checked in addition to compilation.

Achieving integration of user-defined checks within a compiler requires a new balance between power and precision on one hand versus speed and usability on the other hand. We propose here to favor speed and usability based on a minimalist class of user-defined properties, trivial to define by any programmer and checkable very efficiently, yet covering many well-known checks on systems code. The checks we propose are simply expressed as reachability queries in the control flow graph, constrained by syntactic, semantic, and dataflow information.

Our approach aims specifically to be easy to integrate in existing compilers, in order to allow a really widespread use of checking technology by every programmer. To achieve this portability goal, we use a language-independent pattern matching technique that can be implemented very concisely in virtually any compiler.

We practically demonstrate our approach by presenting a full-fledged checking compiler prototype built as a customizable version of the popular gcc compiler, and called mygcc. This prototype checks and compiles full C while adding only about 1000 lines of C source code to the gcc compiler. First performance measures indicate that the overhead of checking does not exceed compilation time even on rather complex checks, and may be as small as 15% for basic checks. Thus, it now becomes possible to perform checking at every compilation.

As a price to pay for this efficient execution, our checking tool is neither sound nor complete, as opposed to full model checkers. Nevertheless, it aims at significantly helping developers by constantly pointing them many likely bugs. Sources of incompleteness include its inability to detect inter-procedural bugs and the overlooking of alias information. Sources of unsoundness include the overlooking of most (but not all) dataflow information, which means that some false positives may be reported that correspond to unexecutable paths. It is important to note that none of these limitations of the tool constitute fundamental limitation of our compiler-integrated approach, nor of our checking language. For instance, alias information already computed by the compiler might be taken into account without changing the checks, and probably without significantly slowing down the checking. One of the main goals of this paper is to encourage compiler developers to experiment with more powerful analyses that might be efficient enough to do at every compilation.

Notwithstanding the limitations of our concrete implementation, we were able to estimate the potential benefits of compiler-integrated checking on the maintenance of a significant code base — part of the Linux kernel. This experiment shows that some limitations of the sporadic checking approach can be overcome using permanent checking.

The main contributions of this paper can be summarized as follows:

– we present an approach to closely integrate checking and compiling, and we illustrate it with a full-fledged compiler (mygcc) able to check user-defined properties in addition to usual compilation; based on this integration, we introduce the concept of permanent code checking during the whole development process

– we present the Condate declarative language for expressing simple user-defined checks integrating in a very concise form syntactic, semantic, control flow, and data flow information

– we validate the expressiveness, the precision, and the efficiency of Condate and mygcc by applying them to successfully check some part of the Linux kernel.

The rest of this paper is organized as follows. Section 2 presents the approach of compiler-integrated checking. Section 3 discusses the design principles of the Condate language, then defines its syntax and semantics. Section 4 describes how condates can be checked efficiently. Section 5 discusses further implementation issues on a real case. Section 6 presents the permanent checking approach enabled by compiler-integrated checking, and experimentally demonstrate its usefulness. Section 7 discusses related work, and Section 8 concludes.

2. Compiler-integrated checking

Existing tools for user-defined program checking use various approaches, but share a common, apparently minor design choice: they are specialized tools, doing *only* program checking. There are several important consequences of this design:

- most of the tools are completely decoupled from existing development environments
- they duplicate a considerable amount of work on program parsing and analysis; this is true even for tools that achieve a superficial level of integration by being called automatically from existing IDEs or makefiles
- they afford to perform costly analyses, which make them unsuitable for daily use throughout development; at best, existing tools aim only at *scalable* analyses
- last but not least, the vast majority of programmers completely ignore their existence.

We propose to explore the challenge of integrating user-defined checks into the tool which constitutes the core of every development process: the compiler. The goal is not just to build one experimental checking compiler, but to define a methodology to integrate user checking into any existing compiler.

This design decision solves the above problems, and should allow a really widespread use of program checking, in order to deliver a small yet useful part of present checking technology to every programmer.

However, this design decision is indeed a challenge to put into practice, because it imposes a number of severe constraints on the implementation:

- checking should be really fast, which means not only scalable, but comparable to compilation time
- the interface should be smoothly integrated into the compiler interface, and trivial to use; ideally, the user should express many useful checks using only a few compiler options
- the implementation should not contain overly complex tools such as theorem provers, expression simplifiers, or language interpreters, so that compiler implementers could practically accept it.

In order to fulfill the above constraints, the present approach chooses a new balance between checking power and precision on one hand, versus speed and usability on the other hand. This new balance is achieved by a *minimalist interface* consisting of constrained reachability queries, able to express a small class of user properties, checkable in linear time, but covering nevertheless many useful checks.

Furthermore, to ensure widespread adoption of the methodology into any compiler, the implementation should be easily ported to any language without the need to develop complex language-specific front-ends; ideally, the whole implementation should be completely language independent. These language-independence is achieved by a *minimalist implementation* of pattern matching, using unparsed patterns.

2.1. Unparsed patterns

Traditionally, source code pattern matching has been reduced to tree matching, following two different approaches.

According to the first approach, patterns are expressed directly as ASTs (abstract syntax trees) [22, 31, 8], so that any algorithm for tree matching can be used to match them with the program AST. This approach is simple to implement, but writing patterns in AST form requires the user to be aware of both the AST representation of programs and a specific textual notation for it.

According to the second approach, patterns are expressed directly in the concrete syntax of the programing language extended to contain pattern variables (also called meta-variables to distinguish them from the variables of the underlying programming language) [14, 3, 23]. Writing patterns in concrete syntax is trivial for any programmer, but this approach is difficult to implement, because it requires to build a pattern parser, implementing an extended version of the programming language's grammar. Extending the grammar of a real programming language to allow for pattern variables is a difficult task, because it requires adding many new productions both to allow meta-variables in different places and to allow parsing of incomplete programs. These new productions usually introduce many conflicts in LALR grammars, which may be tedious to solve. As an alternative to solving the conflicts in LALR grammars, one may use GLR parsing or other variants of backtrack-based parsing. However, both parsing techniques are much less efficient that LALR parsing when the number of conflicts is significant (which is the case for code patterns). As a result, pattern parsers using either parsing technique usually implement a limited pattern grammar, allowing meta-variables to occur only in certain positions, and also restricting the kind of program fragments that may be matched.

Thus, abstract syntax patterns are easy to implement but difficult to use, while concrete syntax patterns are easy to use but difficult to implement efficiently. To solve this apparent contradiction without sacrificing any of the terms, we designed a new technique of pattern matching based on unparsed patterns.

Unparsed patterns are unrestricted program fragments written in the concrete syntax of a programming language where meta-variables may replace any construct that is represented as a subtree in the AST. However, unparsed patterns can be matched with ASTs without being parsed [37]. The key insight behind unparsed pattern matching is that when matching a program AST with a pattern represented as a string, there is enough structure information in the AST so that the pattern needs not be parsed. In fact, the pattern matching algorithm works by unparsing the AST to compare it with the pattern, rather than parsing the pattern.

In our notation, unparsed patterns are represented as quoted strings, in which pattern variables are preceded by an escape character. In C, the escape character used in the rest of this paper is "%", in order to adhere to the familiar convention used for C "format strings".

For example, "buf = malloc(sizeof(int));", "%X = malloc(%Y);", "%L = %L->next;" are unparsed patterns representing C statements, and "%X = malloc(%Y)" (without the ending semicolon), "%X >= threshold", and "p == NULL" are unparsed patterns representing C expressions.

Note that there is no distinction at the formal level between statement patterns and expression patterns. It just happens that some patterns may match only statements, while some other may only match expressions.

An unparsed pattern matches a source code fragment c if there is a substitution mapping variables to subtrees in the AST of c that makes the pattern equal to c. (We also say sometimes that c matches the pattern.) This implies that the same variable occurring several times in a pattern must stand for an equivalent subtree. For cases where the value of the variable is not important, there is an anonymous variable, noted "%_", that is always free.

For instance, the pattern "%L = %L->next;" matches the statement "list = list->next;" under the substitution $\{l \rightarrow \mathit{list}\}$, but it does not match the statement "p = buf[0]->next;". In turn, this last fragment is matched by the pattern "%_ = %_->next;".

It may be useful to consider that a pattern matches a code fragment if it matches any subtree of the fragment, as opposed to considering only "exact" matches. In this case, the pattern "%X = malloc(%Y)" would match both C expressions and C statements containing such expressions. We choose to adopt this convention in the rest of this paper.

By avoiding to implement a pattern parser, unparsed pattern matching is completely language-independent, except the part that unparses an AST. Unparsers for any language can be generated automatically based on the grammar of the language. Moreover, most compiler already include an unparser for debugging purposes. As a result, unparsed pattern matching can be easily implemented in any compiler.

Based on unparsed patterns, we can define our language for user-defined checks, called Condate.

3. The Condate language

Before giving the precise definition of the Condate language, this section starts by informally describing its main design principles.

3.1. Design principles

Condate is a minimalist declarative language for expressing user-defined code checks. The name of the Condate language (meaning “confluence” in Latin) refers to the confluence between compilers and checkers that it is prototyping. It also stands as a contraction of “control” and “data” because user-defined properties integrate these and other levels of programs properties, described in the following. Therefore, we sometime refer to these properties as “condates”.

3.1.1. *The syntax level*

By allowing meta-variables to stand for any subtree in the AST, unparsed patterns provide a powerful tool to express syntax information in user-defined properties. This is already sufficient to define a large class of properties related to code inspection. To go beyond that, we need to integrate control-flow information in our interface.

3.1.2. *The control flow level*

It is well known that many dataflow analyses and program checks can be expressed as reachability queries over an “exploded program graph” [32], which is the product of the program CFG and another graph (a value-flow graph, or an automaton, for instance).

Integrating control-flow in our minimalist interface is based on the observation that many sequencing properties that were successfully used in the literature to find bugs in real code can be expressed as one or several instances of reachability queries *directly* on the program CFG. This form of reachability problems, that may be called “constrained reachability queries” have the form: ‘Is there a path from a statement f to a statement t avoiding statements v?’, where f, t, and v are unparsed patterns.

For example, looking for memory leaks can be expressed as ‘Is there a path from a “malloc(%X)” statement to the exit node avoiding statements "free(%X)"?'. Similarly, many other common checks may be expressed as constrained reachability queries. For instance:

– reading a closed file: from "close(%F)" to "read(%F,%_)" avoid "%F=open(%_,%_)"

– double lock: from "lock(%X)" to "lock(%X)" avoid "unlock(%X)"

– blocking operation with interrupts disabled: from "disable_interrupts()" to "blocking_function()" avoid "enable_interrupts()"

3.1.3. *The data flow level*

Using reachability in the CFG and unparsed patterns, an unexpected number of useful checks can be encoded. However, the properties thus defined lack any information on the values of program variables.

As a concrete example of this limitation, consider a check for potential null dereferences of dynamically allocated pointers. This check can in principle be expressed as a reachability query: 'Is there a path from "%X=malloc()" to "*%X" avoiding "if(%X!=0)"?'. However, as it is written, the reachability query ignores the outcome of the test, while in fact, paths going through the "else" branch could still contain potential null dereferences.

In order to take into account the result of the test, the query should avoid only *successful* tests matching the pattern "%X!=0". That is, the query has to be written more precisely as: 'from "%X=malloc()" to "*%X" avoiding *successful* tests "%X!=0" and *unsuccessful* tests "%X==0"'. This way, dataflow information can be integrated very naturally in our minimalist language.

Other dataflow information can be transparently integrated in our language. For instance, after binding variable x in the "from" pattern above ("%X=malloc()") to an expression, when recognizing the "to" pattern ("*%X"), the matcher can verify if the new expression is not only syntactically identical to the bound one, but also that the (program) variables it contains have not changed. For this, the matcher can use existing reaching definition information computed by a previous pass of the compiler (which indirectly takes into account other dataflow analyses such as pointer information) to verify that the reaching definitions of all these program variables are the same in the "from" and "to" expressions. This idea can be further extended, for instance by allowing the matcher to recognize an expression for x that is textually different to the bound one, but having the same meaning using aliases.

3.1.4. *The semantics level*

Taking advantage of the compiler-integrated approach, semantic information can be easily added by complementing patterns with calls to executable predicates internal to the compiler. For instance, a pattern such as the following one could match statements that allocate not enough space for the destination variable:

"%X=malloc(%Y)" | TREE_INT_CST(Y) &&
    !INT_CST_LT(Y,size(TYPE_POINTER_TO(X)))

where `TREE_INT_CST`, `INT_CST_LT`, and `TYPE_POINTER_TO` are macros in the compiler testing whether an AST represents an integer constant, respectively whether the value of such an integer constant is less than a given value, and returning the pointer type for a given node[1].

3.2. Condate Syntax

Putting all the previous pieces together, the grammar of the minimalist Condate language, in BNF form, is as follows:

$$S \rightarrow \textbf{from } D \; [\textbf{to } D \; [\textbf{avoid } D]] \qquad (1)$$

$$D \rightarrow E \mid E \textbf{ or } E \qquad (2)$$

$$E \rightarrow P \mid +P \mid -P \qquad (3)$$

$$P \rightarrow \texttt{"}(\%V \mid lit)^{*}\texttt{"} \; [\texttt{|}\, expr] \qquad (4)$$

In the above rules, S is the start symbol, D represents a disjunctive pattern, E an edge pattern, P an unparsed pattern, V a pattern variable, *lit* a literal code fragment, and *expr* an expression executable within the compiler. Note that in the last production, the first vertical bar denotes alternatives in the grammar, while the second vertical bar is part of the Condate language, read as "such that" and used to add an optional semantic constraint as shown in section 3.1.4.

As can be seen in the grammar, some patterns can be omitted in a condate, to express plain reachability or even purely syntactic queries:

– integer division: [is there a path] from "int %X;" to "%X/%_"? The "avoid" patterns missing altogether, we have a pure (or unconstrained) reachability query.

– undefined side-effect constructs: [is there a path] from "%X[%I++] = %Y[%I++]" [to anywhere]? As the "to" patterns is missing, this represents a purely syntactic query looking for statements matching the pattern.

[1] These exact macros exist in gcc, for instance.

Thus, positive patterns such as the "to" pattern default to the "%_" pattern matching anything, while negative patterns such as the "avoid" pattern default to "", the empty pattern matching nothing. The "from" pattern cannot be omitted.

This minimalist language has the following advantages:

- a large number of useful checks can be expressed
- checks are encoded very compactly, grouping together syntactic, semantic, control flow and data flow information
- checks are expressed very naturally from a programmer point of view
- user properties written as condates can be checked in linear time and space, as shown in section 4.

3.3. Condate semantics

The semantics of condates can be formalized as a particular class of regular expressions ranging over paths in the CFG. In principle, we could consider both intra-procedural and inter-procedural paths without changing the syntax of the Condate language. However, exploring inter-procedural paths may be considerably less efficient, and we could not yet prove experimentally that this would lead to an acceptable overhead compared to compilation times. For this reason, we will limit ourselves in what follows to the CFG of a single function. Extension to an inter-procedural setting should be possible to achieve by implementing graph reachability in a global CFG using for example the standard method of context-free language reachability [32], but this is beyond the scope of this paper.

In order to define the particular class of regular expressions that constitute the meaning of condates, we first have to define what regular expressions are. Let a CFG be a graph consisting of a set of nodes and a set of edges representing control flow transitions between nodes. There are two types of nodes in the CFG:

- Action nodes, labeled with program statements that do not contain internal control flow. There is always a single unlabeled edge flowing out of an action node.
- Decision nodes, labeled with expressions in the program used to decide on control flow. There are exactly two outgoing edges, one is labeled with "+" and corresponding to a successful test and the other labeled with "−".

Additionally, there is an EXIT node, with no outgoing edge.

We say that a CFG node matches an unparsed pattern if the statement or expression labeling the node matches the pattern.

Edge patterns are defined as follows:

– (unconstrained) an unparsed pattern p is an edge pattern matching any edge leaving any node matched by p

– (successful) if p is an unparsed pattern, $+p$ is an edge pattern matching the edges labeled "+" leaving any node matched by p

– (unsuccessful) if p is an unparsed pattern, $-p$ is an edge pattern matching the edge labeled "$-$" leaving any node matched by p.

For example, "%X = malloc(%Y)", $+$"%P != NULL", and $-$"%X > val" are edge patterns.

A regular path expression over CFG is a regular expression over an alphabet consisting of edge patterns. Regular expressions are recursively defined as:

– (atomic) if e is an edge expression, e is a regular path expression matching any edge matched by e

– (disjunction) if e_1, ... e_n are edge patterns, $[e_1 ... e_n]$ is a regular path expression matching any edge matched by any of the edge patterns

– (complement) if e_1, ... e_n are edge patterns, $[\hat{}e_1 ... e_n]$ is a regular path expression matching any edge that is not matched by any of the edge patterns

– (repetition) if r is a regular path expression, r^* is a regular path expression matching any path in the CFG that concatenates paths matched by r

– (concatenation) if r_1 and r_2 are regular path expressions, $r_1 r_2$ is a regular path expression matching any path in the CFG that concatenates two paths matched by r_1 and r_2, in this order.

For example, if a and b are unparsed patterns, $a^* b [\hat{}ab]^* b$ is a regular path expression.

By the above definitions, a regular path expression matches a path in the CFG if there is a substitution that makes the edge patterns match all edges in the path. This means that unparsed patterns occurring in a regular path expression may share variables, and one such variable stands for equivalent expressions everywhere.

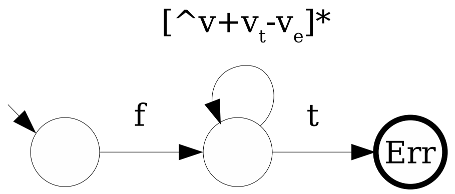

Figure 1. Automaton equivalent to the condate ‘from f to t avoid v or $+v_t$ or $-v_e$’.

Based on these definitions, we can now define the semantics of a condate as a regular path expression. A condate of the form ‘from f to t avoid v or $+v_t$ or $-v_e$’ corresponds to the regular path expression $f[\hat{}v + v_t - v_e]^*t$. The automaton equivalent to a condate is shown in Figure 1, and has a fixed size. We already saw that in spite of this limitation, many useful properties can be expressed.

To ensure an efficient checking of condates, we furthermore impose the restriction that all pattern variables occurring in a condate must be instantiated in the pattern f that recognizes “from” statements, which must be a positive pattern. This not only guarantees that pattern variables are always instantiated in a positive pattern, but also that any substitution instantiating a regular path expression to some path in the CFG is a substitution instantiating its first edge. Each such substitution that instantiates the f pattern to “from” statements in the program is called an *instance* of the condate.

This restriction does not seem to be a severe limitation in practice, because variable instantiated in negative patterns usually count for “don’t cares”, which are supported in condates through the anonymous variable “%_”. Also, we believe that this restriction makes queries easier to write and understand, because users naturally tend to think in terms of problem instances: for any variable x there is a memory leak if $P(x)$; given two locks x and y there is a contention between them if $P(x, y)$; etc.

4. Checking condates

Condates can be checked by the following algorithm, which takes a CFG and a $condate = \langle from, to, avoid \rangle$ consisting of a triple of (possibly disjunctive) patterns:

```
proc $check(CFG, from, to, avoid)$
  $substs \leftarrow \emptyset$                              // collect condate instances
  foreach node $t \in CFG$ do
```

```
      $global\_store \leftarrow \emptyset$                              // empty substitution
      if $match(t, from)$                              // instantiating global store
        then $substs \leftarrow substs \cup \{global\_store\}$
      fi
    end
    for $subst \in substs$ do                          // check one condate instance
      $global\_store \leftarrow subst$                              // instantiate variables
      $list \leftarrow []$
      foreach node $t \in CFG$ do
        // put "from" nodes of the instance on the list
        if $match(t, from)$ then $list \leftarrow [t \mid list]$ fi
      end
      // traverse instantiated CFG
      while $list = [t \mid rest]$ do
            $list \leftarrow rest$
            if $\neg visited(t)$
              then $visited(t) \leftarrow$ true
                if $match(t, to)$
                  then print "reached t"
                   else foreach edge $e = t \rightarrow t'$ do
                          if $\neg match(e, avoid)$
                            then $list \leftarrow [t' \mid list]$
                          fi
                        end
                fi // $match(t, to)$
            fi // $\neg visited(t)$
      end // while
    end // for
  end // proc
```

In a first traversal, the algorithm scans all the program for "from" statements, and collects the number of different substitutions associated to them. The global variable called *global_store* contains a mapping of meta-variables to their current values (subtrees of the program AST). By initializing this mapping to the empty substitution, all meta-variables are reset to free (or unbound). Function *match*() takes into account the values of meta-variables in *global_store*, and upon successful match, may side-effect *global_store* by further instantiating other meta-variables. This way, *global_store* serves to pass meta-variable values between matches along the CFG traversal. To compute the set of all instances of a condate, a set of global stores is used, called *substs*. Each instantiation of the "from" pattern may add a new substitution to this set.

Each substitution collected in this set, representing an instance of the condate, is then checked in two subsequent passes. The first pass over an instance collects the "from" nodes of this instance. These constitute the initial worklist for the second pass over an instance, which traverses the (correspondingly instantiated) CFG from "from" nodes to "to" nodes using only edges not matching the "avoid" pattern. For any "to" node that is reached, there exists a path matching the condate, so this is signaled to the user.

As unparsed pattern matching works in linear time [37], a condate *instance* is checkable in time $O(\mid CFG \mid \times (pvars + labelsize))$, where *pvars* is the number of pattern variables, and *labelsize* is the size of the statements in the program. Considering that the number of meta-variables is a constant in practice (frequently 2 or 3), the only super-linear factor comes from *labelsize*. Section 5.6 will show how this factor can be improved.

The space used by the algorithm is of only one bit per CFG node, to recognize already visited nodes, plus variable *global_store* containing a pointer for each pattern variable. In addition, during the first traversal of the algorithm, a set of global stores is needed; the maximum size of this set is the number of condate instances.

5. Implementation

We implemented the approach described in the previous section in a prototype called mygcc. During this process, we encountered some interesting challenges, described below.

Mygcc was built by integrating into the open-source gcc compiler the above intra-procedural checking algorithm, and the Condate language with the following restrictions:

- pattern matching takes into account only dataflow information resulting from matching positive and negative "to" patterns; reaching definitions are only taken into account for temporary variables, as described in section 5.5; all other dataflow information such as aliases is currently ignored;

- user-defined patterns cannot currently specify semantic information (the "such-that" part).

Mygcc is not just a checker, it is a fully functional gcc version that performs checking as an additional compiler pass. In terms of user interface, Mygcc just adds a new gcc flag "–tree-checks=file" to specify a file containing condates to be checked while compiling the given programs.

Due to our minimalist approach, mygcc implementation consists of only about 250 lines of modifications to existing gcc code, plus about 1000 lines of added C code, among which 600 lines implement the pattern matcher and 400 lines implement the checking engine. Mygcc is freely available [29], and we are currently working with the gcc development team to incorporate the corresponding source changes into an upcoming gcc release.

5.1. Why gcc?

To demonstrate that the compiler-integrated approach can be easily incorporated into *any* existing compiler, we eliminated from the start the idea of using a research-oriented open compiler that would have eased the task by already providing some infrastructure for extensibility and cutting-edge program analyses. As opposed to this open architecture, gcc was not meant to be extensible with user checks, and implements rather well-established analyses.

Other important advantages of choosing gcc are its large user base and its multiple language frond-ends for C, C++, Java, etc. By choosing gcc, we also aim at proving that user-defined checks can be adopted for different programming languages.

5.2. Choosing the intermediate representation

The first implementation choice that we faced was that gcc has no less than four internal representations of the code. First, there are language-specific ASTs in the front-ends (C, C++, Java...). Second, these ASTs are translated by each front-end to a language-independent tree representation called GENERIC. Third, GENERIC trees are simplified to a subset of GENERIC called GIMPLE [28], that keeps some high-level information about the code (lexical scopes, control constructs such as if-then-else) but factorizes most of the syntactic variations (e.g., loops are translated into gotos). Fourth, GIMPLE is translated to a register transfer language called RTL.

Given the high-level patterns specified in user properties, the easiest choice would have been to implement checking on the first representation: language-dependent ASTs.

We chose not to do so, primarily because we aimed at a tight integration between checking and compiler analyses and transformations. Or, in order to factorize analysis code between different front-ends, most of the high-level analyses (building of the CFG, aliases, use-defs) and high-level optimizations (tail recursion elimination, constant propagation, strength reduction, etc) are performed on GIMPLE. Therefore, to take the best advantage of being inside the compiler, we chose to implement checking on the GIMPLE representation.

A second reason for choosing GIMPLE is that various C forms of a statement are factored into a single GIMPLE form. For instance, both C statements "i = i + 1;" and "i++;" are reduced in GIMPLE to the form "i = i + 1;", so both can be matched by the user pattern "%x = %x + 1".

Finally, a third, very important advantage of choosing GIMPLE is that checking can then be used on any language parsed by gcc's front-ends. Indeed, mygcc was initially tested only on C code, as reported in [36]. Since that publication, we also included the C++ and Ada front-ends in the mygcc distribution. After just re-compiling the package, C++ and Ada pattern matching became possible by the combination of our language-independent matching algorithm with the language-dependent unparsers provided by the different front-ends. Of course, user patterns have to be written as they are unparsed by gcc. For instance, a C++ assignment of the form "%_ = %X[%Y]" in which the indexing operator has been redefined is dumped by gcc from GIMPLE form as "%_ = operator[](%X,%Y)", so the user pattern has to be written in this latter form, rather in the former.

Thus, choosing to perform user-defined checks on GIMPLE: allows for a closer integration with the compiler, factors some syntactic variations in user patterns, and enables multi-language checks. However, choosing GIMPLE significantly augments the complexity of matching high-level user patterns with de-sugared code.

A first slight complication is that the GIMPLE representation introduces many explicit casts not present in the source. We easily adapted pattern matching so as to ignore any casts in the AST. One consequence of this is that user patterns must not include any explicit casting.

5.3. Dealing with temporaries

The main difficulty when matching user-level patterns with GIMPLE code concerns the fact that expressions are broken down in GIMPLE to a three-address form, using temporary variables to store intermediate values. We had to adapt the pattern matching mechanism such that when encountering a temporary variable in the matched code to conceptually inline its definition before continuing the matching process.

Let us illustrate this method by the following code fragment, to be checked for unreleased locks 'from "lock(%X,%Y)" to "return" avoid "unlock(%X,%Y)"':

```
lock(step[i+1], NOWAIT);
```

```
critical_section(...);
unlock(step[i+1], NOWAIT);
return;
```

In GIMPLE, the same code may look as follows:

```
T.2 = i + 1;
T.3 = step[T.2];
lock (T.3, 0);
critical_section(...);
T.4 = i + 1;
T.5 = step[T.4];
unlock (T.5, 0);
return;
```

Matching the "from" pattern "lock(%X,%Y)" with the statement "lock(T.3, 0)" bounds the pattern variable x to the temporary variable "T.3", so then the "to" pattern "unlock(%X,%Y)" will not match the statement "unlock(T.5, 0)", because "T.5" is a different variable than "T.3". Therefore, a checking algorithm not taking into account temporary variables does not recognize the correct unlock statement, so erroneously reports that the lock is not released before the return.

This problem can be solved by observing that it is possible to reconstitute the syntax in the original program. Indeed, in the GIMPLE form before optimizations, each syntactically complex expression (containing no internal control flow) is broken down into a block of straight-line code where each use of a temporary corresponds to a definition in the same block. Of course, there may be several subsequent uses for a same definition. Note that this local definition property is not specific to GIMPLE, but is rather typical for how compiler introduce temporary variables, before optimization passes. The original expression can then be reconstituted by inlining temporaries, that is, by substituting every temporary variable with its corresponding definition above in the block. This inlining can be integrated in the matching algorithm, by systematically considering the definition of a temporary instead of the temporary itself.

5.4. Matching lvalues

However, inlining temporaries does not solve by itself the whole problem of matching high-level patterns with simplified code. Consider the following code fragment, which represents a downsized version of a real bug in the Linux kernel:

```
ps->t->table = malloc(sizeof(pixmap));
```

```
  memcpy(ps->t->table,
         pixmap, sizeof(pixmap));
```

This fragment contains a possible null pointer dereference: the allocated pointer expression “ps->t->table” is passed unchecked to the *memcpy* function that uses this pointer as the destination of a copy. However, when the checking algorithm is performed on the GIMPLE form to verify the condate ‘from “%X = malloc(_)” to “memcpy(%X,_,_)” avoid (“%X=_” or +“%X!=0” or −“%X==0”)’, the code fragment has been rewritten as follows:

```
1.  D.2208 = ps->t;
2.  D.2209 = malloc (40);
3.  D.2210 = (char *) D.2209;
4.  D.2208->table = D.2210;
5.  D.2208 = ps->t;
6.  D.2211 = D.2208->table;
7.  memcpy (D.2211, &pixmap, 40);
```

In the above GIMPLE form, the destination of the *malloc()* call is a temporary variable (*D.2209*) that does not occur at all in the *memcpy()* call, even when inlining the definition of temporary *D.2211*.

In fact, the pointer expression “ps->t->table” that was shared in the original program between the calls to *malloc()* and *memcpy()* can be found as the LHS of line 4. It is precisely this expression that should be caught by the pattern variable x, rather than the temporary *D.2209*.

The key observation here is that a GIMPLE temporary definition never represents a definition in the original program, but always represents a use in the original program. Definitions in the original program are always translated in GIMPLE as assignments to a *non-temporary*. Thus, by applying inlining of temporary uses to every statement that is not a temporary definition, one can reconstitute all the statements in the original program. In our example, the original assignment involving *malloc()* can be reconstituted from the definition on line 4, the only non-temporary definition.

Therefore, we adapted the matching algorithm to skip temporary definitions when looking for user-defined statement patterns.

5.5. Binding context

There is another interaction between temporary variables and patterns, which comes from the fact that temporary variables only have meaning when associated with the statement where they occur. When a meta-variable x in a “from” pattern is bound to a temporary, the binding should include not only the temporary, but also the context statement where it was bound. Thus, when the bound meta-variable x is subsequently used in a "to" or "avoid" pattern for matching another statement, inlining the temporary bound to x must be done with respect to its original reaching definition, and not with respect to the current statement. To implement this mechanism, the global store was extended to include for every pattern variable both its value and its "binding context", i.e., the program location where the bind occurred; the pattern matcher was adapted to take into account this information when inlining temporaries.

In principle, this binding context could be used by the matcher more generally to access various kinds of *flow-sensitive* dataflow information computed by previous analyses in the compiler, such as reaching definitions for all variables (not only for temporary variables) flow-sensitive alias information, etc. However, the use of such dataflow information is not implemented in our current prototype.

5.6. On-demand temporary inlining

Interestingly, some inlining can be avoided if the intermediate representation has been constructed using a form of global value numbering (GVN) [33]. In such a representation, a given temporary variable represents always the same expression, or an equivalent one. Actually, gcc does use such an algorithm for building the GIMPLE form, which means that in reality the code fragment in section 5.3 really looks as follows:

```
T.2 = i + 1;
T.3 = step[T.2];
lock (T.3, 0);
critical_section(...);
T.2 = i + 1;
T.3 = step[T.2];
unlock (T.3, 0);
return;
```

In this form, inlining of T.3 can be avoided by binding the pattern variable x to "T.3" instead of to "step[i+1]". Thus, inlining a temporary can be avoided by binding a pattern variable that is shared between two patterns to the temporary instead of to the expression it represents. However, when the pattern portion that is matched with the temporary is not a free variable, the temporary must be inlined to check that its definition corresponds to the pattern. For example, if the above code fragment is checked against the condate 'from "lock(%X+1, %Y)" to "return" avoding "unlock(%X+1, %Y)"', the definition of "T.3" has to be recursively inlined to retrieve the '+' operator in the pattern.

Thus, this on-demand inlining is driven by the pattern, not by the original statement, which means that its recursive application is bounded by the size of the user pattern.

When eager inlining is used, the matching time is linear in the size of the statements in the original program and in the size of user patterns. When on-demand inlining is used, the matching time is linear in the size of GIMPLE statements and in the size of user patterns. The size of GIMPLE statements is constant, and the size of patterns is under user control, and may be eventually bounded by a constant. Therefore, on-demand inlining changes the complexity of the checking algorithm by replacing the factor related to the size of program statements, *labelsize*, with the size of the patterns, *patternsize*.

6. Experimental evaluation

The compiler-integrated approach, as implemented in mygcc, makes it possible to perform *permanent checking* of a restricted class of user-defined properties during all the development process. This technique was not possible using previous checking tools.

In order to assess and validate both our class of user-defined checks and more largely the compiler-integrated approach, we applied mygcc to reproduce the detection of some previously reported bugs in the Linux kernel [9]. That previous study applied 12 different user-defined checkers written in Metal for the MC tool to detect over 500 bugs in kernel version 2.4.1. All these bugs were manually inspected and/or confirmed by kernel developers. A summary of the MC results is freely accessible as an on-line database [27]. As MC is a basically intra-procedural tool, this study proved that it is indeed worth to carefully check intra-procedural properties in real system software. This also meant that even our current implementation of mygcc could have a real potential usefulness. However, checks in Metal are considerably more powerful than condates — they are expressed as arbitrary automata mixed with executable C code. Interestingly, MC was also developed as an extensible version of gcc, but using the traditional design of a standalone checker: it was only meant to do program checking, thus loosing its initial compiler features.

Using this excellent and well-established testbed, our approach had to be validated in several respects:

– test the expressiveness of condates by expressing a maximum number of checks

- test the precision of condates with respect to MC by reproducing a maximum number of bugs with the smallest possible number false positives
- test the scalability and performance of mygcc on a large code base
- test the usefulness of permanent checking with respect to sporadic checking by finding practical evidence that it addresses some limitations of the latter

6.1. Expressiveness

To assess the expressiveness of condates, we expressed as condates as much as possible of the 12 MC checkers cited above. The results are given in Table I.

As can be seen in the table, 11 checkers out of the 12 can be expressed partially or completely in Condate, only two of them (SIZE and VAR) using semantic patterns. A single checker (INULL) cannot be expressed conveniently as a condate because it encodes complex checks about pointer uses and their implied assumptions.

An interesting comment about the expressiveness of condates is that this framework allows to optimize the checkers by factorizing similar user properties in the same condate. For instance, many similar checks written for the Linux kernel are different versions of the LOCK checker. They all check for leaving a function with an active lock, but differ only on the names of the functions used to lock (l_1, l_2, ..., l_N) and unlock (respectively: u_1, u_2, ..., u_N). These similar checks can be grouped in the following condate:

from "l_1(%X)" or "l_2(%X)" or ... "l_N(%X)" to "return" avoid "u_1(%X)" or "u_2(%X)" or ... "u_N(%X)".

This factorization trades some precision for speed, because it would miss the bug in the code fragment: "l_1(a); u_2(a); return;", where an incorrect function (u_2 instead of u_1) is used to release the lock. In practice, different lock functions are usually type-incompatible, so the situation may never occur. Anyways, it is up to the user to evaluate if such a factorization is safe or not.

6.2. Precision

To test the precision of mygcc with respect to MC, we chose one Metal checker called "NULL" that checked possible null pointer dereferences and reported 124 bugs in 89 source files in the kernel. We rewrote this Metal checker in Condate, and we tried to reproduce as much as possible of the bugs that were reported by it.

Table I. Expressiveness comparison between Metal and Condate.

Checker	Tool	Specification
BLOCK	MC	To avoid deadlock, do not call blocking functions with interrupts enabled or a spinlock held
	mygcc	from “lock()” to “blocking_function()” avoid “unlock()”
NULL	MC	Check potentially null pointers returned from routines
	mygcc	from “%X=malloc(%_)” to “*%X” or “%X->%_” avoid +“%X != 0” or −“%X == 0”
VAR	MC	Do not allocate large stack variables (>1K) on the fixed-size kernel stack
	mygcc	“%T %X” | TYPE_P(T) && TREE_CODE(X)==VAR_DECL && DECL_SIZE(X)>1024
INULL	MC	Do not make inconsistent assumptions about whether a pointer is null
	mygcc	N/A
RANGE	MC	Check bounds of array indices derived from user data
	mygcc	from “copy_from_user(&%X, %_, %_)” to “malloc(%X)” or “%_[%X]” avoid “%X < %_” or “%X <= %_”
LOCK	MC	Release acquired locks
	mygcc	from “lock(%X)” to “return” avoid “unlock(%X)”
	MC	Do not double-acquire locks
	mygcc	from “lock(%X)” to “lock(%X)” avoid “unlock (%X)”
INTR	MC	Restore disabled interrupts
	mygcc	from “cli()” to “return” avoid “sti()”
FREE	MC	Do not use freed memory
	mygcc	from “free(%X)” to “*%X” or “%X->%_” avoid “%X=%_”
FLOAT	MC	Do not use floating point in the kernel
	mygcc	from “float %X” or “double %X”
REALLOC	MC	Do not lose a pointer if realloc fails and returns null
	mygcc	from “%X = realloc(%X)”
PARAM	MC	Do not dereference user pointers
	mygcc	from “copy_from_user(&%X, %_, %_)” to “*%X” or “%X->%_” avoid +“%X != 0” or −“%X == 0”
SIZE	MC	Allocate enough memory to hold the destination type.
	mygcc	“%X = malloc(%Y)” | TREE_INT_CST(Y) && !INT_CST_LT(Y, size(TYPE_POINTER_TO(X)))

These 89 source files also contained bugs reported by other Metal checkers. We also rewrote two of these checkers in Condate and tried to reproduce all the corresponding bugs in these 89 files. These two Metal checkers were looking for uses of freed pointers (the "FREE" checker) and calls to blocking functions with interrupts disabled or while holding a spin lock (the "BLOCK" checker).

The three checkers rewritten in Condate (completely included in Appendix A) are compact: 49 lines for NULL, 4 lines for FREE, and 16 lines for BLOCK. The NULL checker is the largest because we aimed at reproducing all the bugs found by the Metal checker in the whole kernel, so we had to include all the syntactic patterns it checked for. For the other two checkers, we wanted to reproduce only the bugs in the selected 96 files, so we could include only the particular patterns that appeared in these files.

Within these 89 files, we successfully found 117 NULL bugs out of the 121 found by MC. Only four NULL bugs were missed by mygcc, in spite of its minimal interface and implementation. The FREE and BLOCK checkers found a total of 13 bugs out of 13 reported by MC on these files. Among the bugs found by both MC and mygcc, two were diagnosed slightly differently. More surprisingly, in addition to the bugs previously reported, mygcc found two new bugs.

6.2.1. *Missed bugs*

Two of the bugs missed by mygcc are of the same type, one in file "namei.c", function "udf_add_entry", and another in file "upcall.c", function "coda_upcall". The latter for instance occurs in the following code fragment:

```
CODA_ALLOC(sig_req, struct upc_req *,
           sizeof (struct upc_req));
CODA_ALLOC((sig_req->uc_data), char *,
           sizeof(struct coda_in_hdr));
sig_inputArgs = (union inputArgs *)sig_req->uc_data;
sig_inputArgs->ih.opcode = CODA_SIGNAL;
```

In this fragment, "CODA_ALLOC(pointer, size)" is a macro that allocates memory of the given size into the given pointer using function "kmalloc()", without checking the returned pointer against null. There are two errors in this fragment. The first one, found by both MC and mygcc is the dereference "sig_req->uc_data" on the third line, without checking "sig_req" for null. The second error, found by MC but not by mygcc is the dereference "sig_inputArgs->ih.opcode" on the last line, without checking the pointer "sig_inputArgs" for null. Mygcc does not find the error because this pointer comes from the allocation in the second line but only indirectly through variable "sig_req->uc_data". Mygcc does not track equalities between different variables (other than temporary inlining), so it misses this second error. As opposed to this, MC finds the error because the user property is expressed as an executable automaton that can carry values between different states.

Two other missed bugs are of a different type. They occur in file "aironet4500_card.c", for instance in function "awc4500_isa_probe", in the following code:

```
if (!dev) {
        dev = init_etherdev(dev, 0 );
}
dev->priv = kmalloc(sizeof(struct awc_private),
                    (0x02 | 0x01 | 0x04) );
```

In this code fragment, the pointer "dev" is allocated by function "init_etherdev()" and then dereferenced without being checked, which is a bug. The problem seems trivial to detect at first sight, but the subtlety comes from the semantics of the function "init_etherdev()", which depending on the value of its first argument either initializes an existing structure (if the argument is not null) or allocates memory (if it is null). Only in the latter case the function may return a null pointer. To avoid plenty of false positives, the "to" pattern had to be written "%X=init_etherdev(0, %_)". With this pattern, mygcc finds some real bugs, in cases when an explicit null pointer is passed. However, in the present example, the pointer is known to be null only because it has just been tested. MC is able to deduce this information, but mygcc is not.

6.2.2. *False positives*

With respect to MC, mygcc found only three additional false positives, both in file "namei.c", and both of the same type. For instance, one is found in function "udf_find_entry", which contains the following code fragment:

```
if (!(fibh->sbh = fibh->ebh = udf_tread(...)))
{
  udf_release_data(bh);
  return NULL;
}
...
nameptr = (Uint8 *)(fibh->ebh->b_data + poffset
                                      - lfi);
```

In the addition on the last line, the dereferenced pointer "fibh->ebh" is guaranteed to be non-null, but the test against null on the first line is done on another variable, “fibh->sbh”. This is directly visible in the GIMPLE form below. Again, as mygcc does not track variable equality, it incorrectly signals the dereference on the last line as an error.

```
  T.945 = udf_tread (T.916, block, T.944);
  fibh->ebh = T.945;
  T.946 = fibh->ebh;
  fibh->sbh = T.946;
  T.947 = fibh->sbh;
  if (T.947 == 0B)
    {
        bh.943 = bh;
        udf_release_data (bh.943);
        return 0B;
    }
...
    T.946 = fibh->ebh;
    T.959 = T.946->b_data;
```

On the same example, MC does not signal the false positive because the user property automaton carries values between different user variables.

To circumvent mygcc’s false positive on this example, it is sufficient to inverse the test on the first line from:

```
if (!(fibh->sbh = fibh->ebh = udf_tread(...)))
```

to:

```
if (!(fibh->ebh = fibh->sbh = udf_tread(...)))
```

This kind of turnaround is easy to find, and in fact is frequently used by programmers to circumvent false warnings of traditional compilers, e.g., false warnings about uninitialized variables.

Yet another kind of false positive was found in file “slram.c” in function “init_slram”:

```
mymtd->priv = (void *)kmalloc(sizeof(struct mypriv),
                              GFP_KERNEL);
if (!mymtd->priv)
{
  kfree(mymtd);
  mymtd = NULL;
}
memset(mymtd->priv, 0, sizeof(struct mypriv));
```

In this code fragment, the allocated pointer “mymtd->priv” is checked for nullity, but even if it is null, the code runs into a dereference of it in the last line, signaled by mygcc. This is a false positive, because the variable "mymtd" has been re-assigned in the mean time. Mygcc does not currently verify whether variables occurring in a matched expression are being re-assigned. This should be fixed in a future version, by treating such assignments as implicit "avoid" nodes.

In reality, there is another, undetected bug here, because when the pointer "mymtd" is assigned to null, it is immediately de-referenced. Our checker does not take into account explicit assignments of pointers to null. This would be very easy to fix by adding the pattern "%X = 0" to the condate implementing the NULL checker. We do not know why MC overlooked this bug, too.

6.2.3. *Different diagnostics*

For two bugs that were found by both MC and mygcc, there are some differences in their diagnostics. The first difference concerns a bug in file "inode.c", in 'function "bfs_read_super":

```
inode = iget(s,i);
if (inode->u.bfs_i.i_dsk_ino == 0)
  s->u.bfs_sb.si_freei++;
else {
  set_bit(i, s->u.bfs_sb.si_imap);
  s->u.bfs_sb.si_freeb -= inode->i_blocks; ... }
```

The bug is that pointer "inode" is allocated by function "iget" and dereferenced with no check. MC correctly signals the errors on the second line, when the pointer is dereferenced in the condition of the if statement. Due to a limitation in its current implementation mygcc searches "to" patterns only in elementary statements, so it overlooks the condition of the "if", and signals the error on the last line instead. This limitation is easy to eliminate in a future release.

The second difference concerns a bug in file "sunhme.c" in function "happy_meal_pci_init":

```
dev = init_etherdev(0, sizeof(struct happy_meal));
...
if (!strncmp(dev->name, "eth", 3)) ...
```

In this code fragment, pointer "dev" allocated by "init_etherdev" appears to be dereferenced without being checked in the expression "dev->name". Indeed, MC signals the error at this precise point. As opposed to MC, mygcc checks the following GIMPLE form:

```
T.2302 = init_etherdev (0B, 512);
dev = T.2302;
```

```
...
dev.2309 = (char *)dev;
T.2310 = strncmp (dev.2309, "eth", 3);
```

In this GIMPLE form, there is no more dereference of pointer “dev” before being passed to function “strncmp”! This is simply because the field called “name” in the “dev” structure is the first field in this structure, and is a character array. Therefore, to obtain the address of “name”, GIMPLE simply casts the pointer as a string, and passes it directly to “strncmp”. Indeed, the dereference “dev->name” in the original code exists only in the concrete syntax, but does not correspond to a real pointer dereference in the executed code. Hence, MC signals in fact a false positive in this expression! However, passing a null pointer to “strncmp” is an error anyways, which is caught by mygcc because passing a pointer to “strncmp” has been declared in the condate as a “to” pattern, like other dereferences.

6.2.4. *New bugs*

Mygcc also found two new bugs not previously reported by the MC study.

The first new bug was found in file “anode.c” in function “hpfs_add_sector_to_btree”. The bug concerns a dereference of pointer “anode”, allocated by function “hpfs_map_anode()”. The control path is too complicated to detail here. It is possible that MC found this bug but that it has not been validated by manual inspection.

The second new bug was found by mygcc in file “intrep.c” in function “jffs_scan_flash”, where pointer “read_buf” is dereferenced unchecked:

```
read_buf = (__u8 *) kmalloc (sizeof(__u8) * 4096,
                             GFP_KERNEL);
...
if(*((__u32 *) &read_buf[i]) !=
   JFFS_EMPTY_BITMASK)
  break;
```

Probably, MC overlooked this bug because its NULL checker did not provide a pattern for such a complex syntax to dereference a pointer. As opposed to MC, mygcc works on the GIMPLE form where the “if” condition has been decomposed, so the dereference of the variable has been rewritten in a more standard form:

```
i.1176 = (unsigned int)i;
i.1177 = (__u8 *)i.1176;
T.1178 = read_buf + i.1177;
T.1179 = (__u32 *)T.1178;
```

```
T.1180 = *T.1179;
if (T.1180 != 0ffffffff)
{
  goto <D11400>;
}
```

Using temporary inlining and cast skipping, mygcc recognizes on the fifth line the form "*(read_buf + %_)" that has been declared in the condate as a dereference.

6.3. Need for permanent checking

In order to estimate the usefulness of permanent checking, we verified whether all the bugs reported by the MC study have been fixed in a subsequent kernel version, v. 2.6.13, released in August 2005. The results on kernel 2.4.1 were published in 2001 and Linux kernel developers were informed about the existing on-line database summarizing the bugs. Therefore, this experiment covers a lapse of 4 years of active maintenance of a significant code base.

When re-conducting the checks described in the previous subsection on the new kernel version, mygcc found four surviving bugs:

- one previously reported bug (in file skfddi.c, function *skfp_driver_init*) and one of the 4 new bugs mentioned above (in file anode.c, function *hpfs_add_sector_to_btree*) have survived identically (interestingly, the other 3 new bugs have disappeared, even if MC did not signal them); the containing functions have only slightly changed in the mean time

- one bug (in file riotable.c, function RIOReMapPorts) remained untouched, in spite of the fact that this function has been significantly changed for other reasons

- one bug (in file inode.c function *bfs_read_super*) survived in a different form; the code has radically changed between the two versions: the containing function does not exist anymore, but a code fragment similar to the old bug can be found now in another function (*bfs_fill_super*)

The fact that almost all the bugs were fixed in the new version clearly shows that the MC report was taken very seriously into account by Linux kernel developers. The three previously reported bugs that survived in spite of this intense correction effort demonstrate that without a proper tool to enforce user properties, even well-known bugs

Table II. Performance of mygcc.

File	**Checkers**	**Time** (secs)	**Overhead** (%)
inode.c	none	0.291	
	NULL	0.503	72
	all	0.506	74
comx-proto-fr.c	none	0.626	
	FREE	0.715	14
	NULL	0.929	48
	all	0.989	58
iphase.c	none	2.026	
	FREE	2.242	11
	LOCK	2.309	14
	NULL	3.644	80
	all	4.013	98
all Linux files	none	137	
	FREE	140	2
	LOCK	140	2
	NULL	171	24

can survive for long periods of time (four years in our case), or can be re-introduced during maintenance.

6.4. Performance

The examples described in the Linux study illustrate the fact that mygcc is able to check any program that gcc can compile. Thus, the scalability of the prototype to real programs is clearly demonstrated. But mygcc aims not just at being scalable to large programs, but also to impose a reasonable overhead on compilation time.

We measured the overhead of different checkers when compiling three programs that are part of the above Linux study: a program featuring only NULL bugs, one with additional FREE bugs, and a last one with the three types of bugs we checked for. The results are summarized in Table II. The last line shows the total time for checking all the 89 files part of the Linux kernel. The benchmarks were performed on a Linux PC with an Athlon XP2800+ processor and 256MB of memory.

The checking overhead is directly related to the number of checkers used, to the number of condate instances found in the program, and to the size of the patterns. This explains the large variations between the different measuring points. However, it can be seen that the checking time never exceeds compilation time in these typical examples of the Linux study. Overheads are of the order of 10-15% for a very simple checker (FREE, containing a total of 6 patterns), 15% for a moderate checker (LOCK, including 11 patterns), and 50-80% for a complex checker (NULL, including a total of 51 patterns, among which 24 are disjuncts of a single “from” pattern). The maximum overhead when combining all the checkers is 98%. As can be seen from the last line, on average, the overhead of a simple checker is of 0-2%, and the overhead of all the combined three checkers is of 24%.

When interpreting the figures, it is important to note that we did not have the time to optimize the implementation of the current prototype. To give only one example, mygcc internally uses pattern matching to decide whether a node is an assignment; while this self-application is elegant, this check could be optimized by directly testing the AST node label.

7. Related work

At the most basic level, every compiler routinely performs some fixed checks on the program, such as detecting uninitialized variables, etc. The novel proposal in this paper is to include user-defined checks in compilers.

The most common approach to user-defined checking is to define a programming model in which users may write their own program inspection passes. Tools implementing this approach incorporate a front-end that parses the program in the form of an AST and offer either an application programming interface (API) or a domain-specific programming language (DSL) to walk the AST and implement different forms of checks.

API-based code inspectors include SoftBench CodeAdvisor from HP, in which user-defined checks have to be coded in C++, Checkstyle [7] and FindBugs [19], with checks coded in Java. More recently, some extensible code inspectors such as PMD [31] build an XML representation of the AST, on which user-defined checks can be expressed either in JAVA, or in a declarative way using Xpath patterns. API-based tools allow in theory to implement any user-defined checks. They offer a solid basis to inspect syntax, but little or no semantic information is pre-computed. None of these tools pre-compute the control-flow graph, therefore no dataflow information is available. For these reasons, API-based code inspectors make it easy to define syntax checks such as adherence to a coding standard, or computations based on syntax traversal such as function call graphs or class hierarchy extraction. In turn, writing any kind of non-local semantic checks such as verifying sequences of operations or performing model checking requires an important amount of code.

Tools defining a DSL to write code checkers include CodeCheck [1] and tawk [22], defining an imperative languages close to C, Genoa [13] defining a functional language close to Lisp, and ASTLOG [11], defining a variant of the Prolog language. DSL-based tools can very compactly encode sophisticated tree patterns or tree traversals, but none of the cited DSLs integrate control or dataflow information in the language, neither in explicit nor in implicit form.

Writing checkers for both API and DSL code inspectors requires the user to be aware of the details of the AST representation for the subject language, in addition to the API or DSL to traverse it.

Another set of program checkers such as Splint [15, 34] and CQUAL [16] are based on extensible type checking. In this approach users must annotate the types of the checked program with qualifiers expressing program properties that can be checked automatically. Type-based checkers are very efficient (e.g., linear-time) and precise (e.g., sound) in verifying "global" properties in a program, i.e., that do not depend on control flow. Some of these checks could definitely be integrated in a compiler-integrated approach, but for now are implemented as standalone tools. Some extensions were added to check flow-sensitive types [20], but in this case the performance is no more suitable for permanent checking.

Yet another class of extensible checkers transpose model checking techniques, used since a long time in hardware verification, to programs, viewed simply as CFGs, in which the semantics of individual program statements is usually ignored. In this approach of lightweight model checking, user-defined properties represent legal sequences of operations, and are represented by finite automata. Transitions are triggered by syntactic patterns matching program statements. Checking is done by conceptually executing the automata along the CFG. Lightweight model checkers include Cesar [30] for checking Fortran and his evolution called FLAVERS [10] for Ada and Java, MC [14] for C and its variant MJ [3] for Java, MOPS [8] and CodeSurfer Path Inspector [21] for C, PQL [26] and SAFE [17] for Java. Engler et al. clearly demonstrated the practical usefulness of the lightweight model checking approach, by applying MC to detect hundreds of system programming bugs [14] and security bugs [2] in C code. The running time complexity of these tools has been precisely analyzed in the framework of parametric regular path queries [25]. Essentially, the checking time depends on the size of the user automaton. For simple automata, checking may be done in linear time. Some tools such as SAFE allow to parameterize the precision of complementary analyses such as alias analysis so as to obtain more responsive checks for interactive use or more precise checks for use in batch. This feature of SAFE allowed it to prototype a method of continuous quality assurance, not integrated with the compiler, but within the IDE. In comparison to automata-based tools, the checks allowed by our tool are a particular case of lightweight model checks where the automaton has a fixed form with only three states: the initial state, the state after a “from” node, and the error state. One original feature of our approach is allowing to define transitions that depend on variable values, using “successful” and “unsuccessful” patterns. More importantly, all cited tools are distinct from the compiler, and therefore duplicate a great amount of analysis work. Note also that our unrestricted, language-independent pattern matching could be useful in many of these tools.

More precise program checkers take into account variable values in order to distinguish between feasible and unfeasible path. Among them, SLAM [4], Blast [23], and ESP [12] also express user properties as automata. The BLAST checking algorithm has been integrated within an existing IDE as an Eclipse plug-in and optimized to work incrementally, in order to support “Extreme model checking” [24], which consists of performing user checks on each release of a program, during software development. These tools integrate complex subsystems such as symbolic executors, theorem provers and/or expression simplifiers, that cannot reasonably be integrated within compilers. This means that they are designed to remain standalone tools, used out of the critical path in development. Our approach for permanent checking is complementary to extreme model checking, as it chooses to perform simpler checks but that can be integrated easily in every compiler and re-done at every compilation.

Finally, full software model checkers, such as, for example Java PathFinder [35], verify user-defined properties (possibly defined as automata) on non-deterministic programs (e.g. concurrent programs) by trying to execute all possible sequences of non-deterministic choices. The applicability of this approach is usually limited to medium-sized systems because of the state space explosion problem. This remains true even if existing tools achieve great savings by using complex search heuristics, by reducing the number of states explored, and by reducing the state storage cost.

Condates are a variant of parametric regular path queries [25], with the following modifications:

– we only allow existential queries, which can be solved more efficiently than universally quantified queries

– edges on the CFG are directly labeled with program instructions, instead of some abstractions thereof; this eliminates the need of a specific front-end for each checker

– we allow to encode a minimal amount of dataflow information in regular expressions, through the "+" and "−" labels; this significantly augments their expressiveness.

Typestate verification [18] is similar to this paper in the sense that it defines classes of regular expressions over program events that are checkable in polynomial time. However, the checks described here do not belong to any of those classes. On the other hand, typestate verification problems take into account (a single level of) aliases among variables, while we currently ignore alias information.

A quite different approach to program checking is to express used-defined properties as so-called contracts associated to interfaces, and consisting in predicates to be checked before and after function calls. Thus, JML [6] extends Java with contracts expressed as stylized comments, and Spec# [5] extends C# with contracts integrated in the base language. In both of these systems, user properties cover a restricted set of first-order logic, and may be checked either statically or dynamically. Similarly to our approach, user properties are handled by an extended compiler. Moreover, some of the properties mentioned in this paper can be re-phrased as function contracts. One major distinction is that contracts are tied in these systems to pre- and post-conditions around function (or method) calls, while mygcc can check properties on every program construct, using pattern matching — not only on function calls. Besides, contract properties cannot directly refer to the control flow, which is a very natural way to express some properties. On the other hand, Spec# defines non-null types that are statically checked, which allows to protect against null pointer de-references. These types are similar to CQUAL's type qualifiers, discussed above.

8. Conclusion

We presented a pragmatic approach for easily extending existing compilers with user-defined checks, very simple to express and very efficiently checked. The practical applicability of the approach was demonstrated by its very concise implementation in the gcc compiler.

The fusion between checking and compiling enables a software development method in which checking permanently accompanies evolution, from the early coding phase to the maintenance phase. It also considerably increases efficiency by eliminating a lot of duplicated analyses.

Beyond these immediate advantages, the fusion between the checker and the compiler opens up new perspectives for:

– Integrating specifications within library interfaces, with no language extension: each interface file could be complemented by a separate check file describing sequencing constraints.

– Integrating specifications within the program itself: since the compiler is also the checker, one can imagine to define, enable, and disable user checks using compiler pragmas.

– Integrating checking with program analyses and optimizations: by implementing all in the same tool, cross-fertilizations are possible, such as optimizations dependent on sequencing constraints, or checks facilitated by program optimizations; we have shown in section 5.6 just one example where GVN improves the complexity of checking.

– Integrating checking within generated code: checks that cannot be performed statically by the compiler could be easily integrated into the generated code to be executed at runtime.

Mygcc is a particular point in a wide spectrum of possible trade-offs. It proves that it is possible to integrate compilation and checking, with little implementation effort, and with an acceptable overhead. But this is just a starting point. Important future extensions may include extending to inter-procedural checks, and to other classes of regular path queries also checkable efficiently. The main goal of this paper was to motivate compiler developers to experiment with offering more powerful user-defined checks. One open question that is raised is: how much checking power can one put in a compiler to maintain a reasonable runtime and implementation overhead?

Appendix

A. Complete Condate listing

The complete checkers used by mygcc to perform all the mentioned tests on the Linux kernel are listed in the following.

```
## FREE checker:
## Do not use freed memory.
from  "kfree_skb(%X)" or "dev_kfree_skb_any(%X)" or
      "kfree(%X)"
to    "%_ = %X->%_" or "%X->%_ = %_"
avoid "%X = %_"

## BLOCK checker (1/2):
## To avoid deadlock, do not call blocking
## functions with interrupts enabled.
from  "__global_cli()"
to    "%_ = request_irq(%_,%_,%_,%_,%_)" or
      "%_ = tty_register_driver(%_)" or
      "%_ = __constant_copy_from_user(%_,%_,%_)" or
      "%_ = __generic_copy_from_user(%_,%_,%_)" or
      "%_ = __constant_copy_to_user(%_,%_,%_)" or
      "%_ = __generic_copy_to_user (%_,%_,%_)"
avoid "__global_restore_flags(%_)"

## BLOCK checker (2/2):
## To avoid deadlock, do not call blocking
## functions with a spinlock held.
from  "spin_lock(%X)" or "spin_lock(%X + %Y)"
to    "ia_tx_poll(%X)" or
      "%_ = ia_pkt_tx(%_,%_)" or "%_ = ia_start(%_)" or
      "%_ = __constant_copy_from_user(%_,%_,%_)" or
      "%_ = __generic_copy_from_user(%_,%_,%_)" or
      "%_ = __constant_copy_to_user(%_,%_,%_)" or
      "%_ = __generic_copy_to_user(%_,%_,%_)"
avoid "spin_unlock(%X)" or "spin_unlock(%X + %Y)"

## NULL checker:
## Check potentially null pointers returned from
## functions.
from  "%X = drm_alloc(%_,%_)" or "%X = vmalloc(%_)" or
      "%X = kmalloc(%_,%_)" or "%X = alloc_pages(%_,%_)" or
      "%X = sb_bread(%_,%_)" or "%X = init_etherdev(%_,%_)" or
      "%X = dev_alloc_skb(%_)" or "%X = skb_clone(%_,%_)" or
      "%X = iget(%_,%_)" or "%X = iget_locked(%_,%_)" or
      "%X = create_proc_entry(%_,%_,%_)" or
      "%X = kmem_cache_alloc(%_,%_)" or
      "%X = scsi_register(%_,%_)" or "%X = udf_tread(%_,%_)" or
      "%X = hpfs_map_anode(%_,%_,%_)" or
      "%X = __idetape_kmalloc_stage(%_,%_,%_)" or
      "%X = alloc_skb(%_,%_)" or "%X = findcontrbydriverid(%_)" or
      "%X = ipc_alloc(%_)" or "%X = fore200e_kmalloc(%_,%_)" or
      "%X = pci_alloc_consistent(%_,%_,%_)" or
      "%X = scsi_malloc(%_)" or
      "%X = hfs_malloc(%_)" or "%X = ckmalloc(%_)" or
      "%X = get_usb_bluetooth(%_,%_)" or
      "%X = kmalloc_node(%_,%_,%_)" or
      "%X = alloc_etherdev(%_)" or "%X = __bread(%_,%_,%_)" or
      "%X = __dev_alloc_skb(%_,%_)" or "%X = ipc_rcu_alloc(%_)"
to    "__constant_c_and_count_memset(%X,%_,%_)" or
      "%_=__constant_c_and_count_memset(%X,%_,%_)"
      or "__constant_c_memset(%X,%_,%_)" or
      "%_ = __constant_c_memset(%X,%_,%_)" or
      "__constant_memcpy(%X,%_,%_)" or
      "__constant_memcpy3d(%X,%_,%_)" or
      "%_ = __constant_memcpy3d(%X,%_,%_)" or
      "__memcpy(%X,%_,%_)" or
      "%_ = __constant_memcpy(%X,%_,%_)" or
      "%_ = __memcpy(%X,%_,%_)" or
      "%_ = %X->%_" or "%X->%_ = %_" or
      "%X->%_[%_] = %_" or "*\(%X + %_\) = %_" or
      "%_ = *\(%X + %_\)" or
      "*\(\(%X + %_\) - %_\) = %_" or
      "\(\(\(%X + %_\)->%_\) = %_\)" or
      "%_ = strncmp(%X,%_,%_)" or
      "%_ = *\(\(%X + %_\) + %_\)"
avoid +"%X != 0" or +"%X != 0B" or "%X = %_" or
      -"%X == 0" or -"%X == 0B" or
      -"\(\(%X == 0B\) || %_\)"
```

References

1. Abraxas Software, Inc. CodeCheck. http://www.abxsoft.com
2. K. Ashcraft, D. Engler. "Using Programmer-Written Compiler Extensions to Catch Security Holes". In Proc. IEEE Symp. on Security and Privacy. May 2002.
3. G. Back, D. Engler. "MJ - a system for constructing bug-finding analyses for Java". Technical report, Stanford University. September 2003.
4. T. Ball, S. Rajamani. "The SLAM Toolkit". In Proceedings of the 13th International Conference on Computer Aided Verification. LNCS Vol. 2102. 2001.

5. M. Barnett, K. Leino, and Wolfram Schulte. In CASSIS 2004, LNCS vol. 3362, Springer, 2004.
6. L. Burdy,Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, E. Poll. “An overview of JML tools and applications.” In T. Arts, W. Fokkink, eds.: “Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)”. Volume 80 of Electronic Notes in Theoretical Computer Science (ENTCS)., Elsevier, 2003.
7. Checkstyle. Open-source project at SourceForge.net. http://checkstyle.sourceforge.net
8. H. Chen, D. Wagner. “MOPS: an infrastructure for examining security properties of software”. In Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS). Washington, DC. November 2002.
9. A. Chou, J. Yang, B. Chelf, S. Hallem, D. Engler, ”An empirical study of operating system errors”. In 18th Symp. Operating Systems Principles (SOSP). Oct 2001.
10. J. Cobleigh, L. Clarke, L. Osterweil. “FLAVERS: A finite state verification technique for software systems”. IBM Systems Journal, 41(1). 2002.
11. R. Crew. “ASTLOG: A Language for Examining Abstract Syntax Trees”. In USENIX Conference on Domain-Specific Languages. October 1997.
12. M. Das, S. Lerner, M. Seigle. “Esp: Path-sensitive program verification in polynomial time”. In Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), Jan. 2002.
13. P. Devanbu. “GENOA — a customizable, front-end-retargetable source code analysis framework”. ACM Transactions on Software Engineering and Methodology (TOSEM) 8(2). April 1999.
14. D. Engler, B. Chelf, A. Chou, S. Hallem. “Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions”. Proc. of 4th Symposium on Operating System Design and Implementation (OSDI), San Diego. October 2000.
15. D. Evans, D. Larochelle. “Improving Security Using Extensible Lightweight Static Analysis”. IEEE Software 19(1). January 2002.
16. J. Foster, M. Fhndrich, A. Aiken. “A Theory of Type Qualifiers”. In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). Atlanta, Georgia. May 1999.
17. E. Geay, E. Yahav, and S. Fink. 2006. “Continuous code-quality assurance with SAFE”. In Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (Charleston, South Carolina, January, 2006). PEPM ’06. ACM Press, New York, NY, 145-149.
18. J. Field, D. Goyal, G. Ramalingam, and E. Yahav. “Typestate verification: Abstraction techniques and complexity results”. In Proc. of SAS’03, volume 2694 of LNCS, pages 439–462. Springer, June 2003.
19. FindBugs. http://findbugs.sourceforge.net/
20. J. Foster, T. Terauchi, A. Aiken. “Flow-Sensitive Type Qualifiers”. In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). Berlin, Germany. June 2002.
21. Gramma Tech. CodeSurfer Path Inspector. http://www.grammatech.com
22. W. Griswold, D. Atkinson, C. McCurdy. “Fast, Flexible Syntactic Pattern Matching and Processing”. In 4th International Workshop on Program Comprehension. 1996.
23. T. Henzinger, R. Jhala, R. Majumdar, G. Necula, G. Sutre, W. Weimer. “Temporal-Safety Proofs for Systems Code”. Proc. of the 14th International

Conference on Computer-Aided Verification (CAV). LNCS 2404. Springer-Verlag, 2002.

24. T. Henzinger, R. Jhala, R. Majumdar, M. Sanvido. “Extreme model checking”. In Proceedings of the International Symposium on Verification: Theory and Practice. LNCS 2772. Springer-Verlag, 2004.
25. Y. Liu, T. Rothamel, F. Yu, S. Stoller, N. Hu. “Parametric regular path queries”. ACM SIGPLAN Notices, 39(6) (PLDI). May 2004.
26. M. Martin, B. Livshits, M. Lam. “Finding application errors and security flaws using PQL: a program query language”. In Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA). 2005.
27. MC bug viewer. http://metacomp.stanford.edu
28. J. Merill, “GENERIC and GIMPLE: A New Tree Representation for Entire Functions”. Proc. of the GCC 2003 Summit.
29. Mygcc prototype. http://mygcc.free.fr
30. K. Olender , L. Osterweil. “Cesar: a static sequencing constraint analyzer”. ACM SIGSOFT Software Engineering Notes 14(8). December 1989.
31. PMD. Open-source project at SourceForge.net. http://pmd.sourceforge.net/
32. T. Reps. “Program analysis via graph reachability”. Information and Software Technology 40(11-12). November/December 1998.
33. B. K. Rosen, M. N. Wegman, F. K. Zadeck. “Global Value Numbers and Redundant Computations”. In Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1988.
34. Splint. Open-source project. http://www.splint.org
35. W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda. “Model Checking Programs”. Automated Software Engineering Journal, 10(2), April 2003.
36. N. Volanschi. “Condate: A Proto-language at the Confluence Between Checking and Compiling”. Eighth ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP). 2006.
37. N. Volanschi, C. Rinderknecht. “Unparsed patterns: easy user-extensibility of program manipulation tools”. In ACM SIGPLAN 2008 Workshop on Partial Evaluation and Program Manipulation (PEPM ’08), January 2008. To appear.