Condate: A Proto-language at the Confluence Between Checking and Compiling

Nic Volanschi

*my***gcc**

nic.volanschi@free.fr

Abstract

Recent years have seen the advent of many different tools for program checking against user-defined properties. Despite this encouraging trend, checking technology is used still marginally today, and only on an occasional basis. Existing checkers are standalone tools, associated — correctly or not — with low efficiency, and duplicating much work already done in the compiler. We believe that, as a complement to more precise verifiers, the next generation of compilers should integrate some amount of user-defined checks that can be performed efficiently.

Combining checking and compiling enables a pervasive propagation of checking technology and continuous use of checking throughout development. It also enables cross-fertilization between the two passes, resulting in increased expressiveness, precision, and even in improved complexity of the checking algorithm.

We illustrate this integrated approach with a full-fledged checking compiler for C, extensible through Condate. Condate is a declarative language for expressing simple user-defined program properties to be checked in addition to normal compilation. Condate mixes in a very concise form syntactic, semantic, control flow, and data flow properties. These properties are defined as a new class of regular path expressions over the control-flow graph, checkable in linear time and covering many useful checks.

We demonstrate the viability of the integrated approach based on Condate by applying it to successfully check some parts of the Linux kernel.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Specialized application languages

General Terms Algorithms, Languages, Verification.

Keywords Declarative languages, Compilers, Program checking, Customization.

1. Introduction

There have been recently important advances in software checking materialized in the advent of many different tools performing various levels of checks. These tools range from purely syntax checkers [5, 29, 9, 19, 12, 11], going through lightweight model checkers [18, 28, 8, 14, 2, 6, 24, 10, 22] up to sound software model checkers [13, 3, 20].

Existing tools use various approaches, but share a common, apparently minor design choice: they are specialized tools, doing *only* program checking. There are several important drawbacks 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 program 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, many programmers completely ignore their existence.

As a consequence of these and maybe other reasons, for instance related to usability or to limited distribution policies (some proprietary tools being kept as a competitive advantage), checking tools are not used on a large scale nowadays.

To solve the above design-related issues, we propose to integrate some amount of user-defined checking within the core of every development process — the compiler.

In a previous work, we showed that user-defined checks can be easily integrated in any existing compiler, by using a language-independent pattern matching technique based on unparsed patterns [33]. Thereby, unparsed patterns enable a pervasive use of checking technology by every programmer. We have implemented this approach in a full-fledged checking compiler prototype called mygcc [34], representing a customizable version of the popular gcc compiler. This prototype checks and compiles full C while adding only about 1000 lines of C source code to the gcc compiler. Based on this prototype, we showed that integrating checking and compiling enables to continuously perform checking throughout software development. This method of permanent checking brings important advantages such as catching bugs in earlier cycles when they are cheaper to fix, and avoiding to re-introduce known bugs.

This paper presents the declarative language used to express user-level checks for mygcc, and formalizes its definition. Our language, called Condate, covers a limited class of sequencing properties on program events, lying at the confluence between control flow and data flow. Similar to previous approaches, program events are identified by syntactic patterns. However, unlike existing approaches, a minimal amount of data flow information is blended into the language without sacrificing its declarative style. This blending is achieved by defining Condate expressions as a restricted class of parametric regular expressions over a labeled version of the

PPDP'06 July 10–12, 2006, Venice, Italy.

control-flow graph (CFG) of a program. The class of properties is designed to be checkable most efficiently during compilation, but to include nevertheless many useful checks.

We show that implementing Condate within a compiler (as opposed to in a standalone checking tool) not only enables pervasive and permanent checking, but also allows for interesting cross-fertilization between checking and compiling, including an improvement in the running time complexity of the checking pass.

Using the implementation within mygcc, we evaluate the Condate language along several axes on a concrete checking benchmark consisting of some part of the Linux kernel. The evaluation axes include the language's expressiveness (the class of expressible checks), its precision (the rate of missed bugs and false errors), and the overhead of the checking pass. This experiment shows that the language can be checked efficiently enough to let programmers add new checks at compilation. This experiment also shows that the expressible checks may avoid hundreds of common bugs that are detected today only by more complex, standalone, software model checkers.

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

- we present the Condate declarative language for expressing user-defined checks to be performed in addition to normal compilation, and integrating in a very concise form syntactic, semantic, control flow, and data flow information
- we precisely define Condate expressions as a new class of parametric regular expressions over the CFG of a program
- we show several benefits of integrating compilation and checking, including an improvement of the running time complexity of checking using global value numbering on three-address code
- we validate the expressiveness, the precision, and the efficiency of Condate by applying it to successfully check some part of the Linux kernel

The rest of this paper is organized as follows. Section 2 recalls the notion of unparsed patterns and the approach of compiler-integrated checking, in order for this paper to be self-contained. Section 3 informally describes the Condate language and its underlying design principles, before we give an exact definition in Section 4. Section 5 describes a standalone algorithm for checking Condate properties. The benefits of compiler-integrated checking are described in Section 6. Section 7 presents the Linux experiment validating and evaluating the Condate language and our prototype. Section 8 discusses related work and Section 9 concludes.

2. User checks in any compiler

Our integrated approach to program checking consists in adding user-defined checks into the tool which constitutes the core of every development process: the compiler. This design decision enables a really widespread use of program checking by every programmer, and also a continuous use of checking throughout development.

Our goal was not just to build one experimental checking compiler, but to define a method to integrate user checking into any existing compiler. Therefore, the implementation had to be easily ported to any language without the need to develop complex language-specific front-ends. These language-independence was 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), 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). 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 that usually introduce new conflicts in the grammar, which may be tedious to solve. As a result, pattern parsers usually implement a limited pattern grammar, allowing meta-variables to occur only in certain positions.

Thus, abstract syntax patterns are easy to implement but difficult to use, while concrete syntax patterns are easy to use but difficult to implement. 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 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 [33]. 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 statement or expression 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 the c matches the pattern.) This implies that a same variable occurring several times in a pattern must stand for the same 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 $\{l \rightarrow list\}$, but it does not match the statement "p = buf[0]->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 implemented almost for free in any compiler.

2.2 The integrated approach

The portable, language-independent technique of unparsed patterns, opens the way to integrate user-defined syntax checks in compilers. Building on this base, one can integrate more powerful checks. However, the design decision of integrating checking and compiling imposes a number of severe constraints on the implementation:

- checking has to be fast, which means not only scalable, but comparable to compilation time; ideally, checking should be of linear complexity in time and space
- the interface has to be smoothly integrated into the compiler interface, and trivial to use
- the implementation cannot contain complex tools such as theorem provers, expression simplifiers, or complex language interpreters, so as compiler implementors can practically accept it

In order to fulfill the above constraints, we chose 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 class of user-defined properties, checkable in linear time, but covering nevertheless many useful checks.

Before giving a formal definition of the Condate language, the next section informally describes its main design principles.

3. Condate design

Condate is a declarative language expressing checks by mixing several levels of code properties.

3.1 Syntax

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.

3.2 Control flow

It is well known that many dataflow analyses and program checks can be expressed as reachability queries over an "exploded program graph", 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)"?'. The exit node may be either any return statement when checking intra-procedurally, or return statements from the main function, when checking inter-procedurally.

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()"

3.3 Data flow

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.

To give a concrete feeling of this limitation, let us 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. However, only paths going through the "else" branch could 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 interface.

3.4 Semantics

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 or expression: "%X=malloc(y)" `| is_cst(y) && val(y)<size(typeof(x))`

3.5 Mixing all together

A constrained reachability query (CRQ) is a query of the form: "Is there a path from a statement f to a statement t avoiding: statements v, successful tests v_t and unsuccessful tests v_e?". A CRQ can thus be expressed as a series of patterns. We sometimes refer to the patterns according to their role in the CRQ as: the "from" pattern, the "to" pattern, the "avoid" pattern, etc.

Of course, some patterns can be omitted in a CRQ, 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.

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 user interface 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 expressed as CRQs can be checked in linear time and space, as showed below.

4. Condate definition

CRQs 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 or the semantics 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 [30], but this is beyond the scope of this paper.

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. Any number of edges may flow out of an action node, and they are all unlabeled.
- 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 "-".

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:

- (generic) an unparsed pattern p is an edge pattern matching any edge leaving any node matched by p
- (success) if p is an unparsed pattern, $+p$ is an edge pattern matching the edges labeled "+" leaving any node matched by p
- (failure) 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 > threshold" 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 an arbitrary concatenation of 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 concatenation of 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 the same expression everywhere.

Properties expressed as regular path expressions can be checked using a simple worklist algorithm [23] by translating the regular path expression to an automaton A, and executing transitions in A that match edges in the CFG. The algorithm starts from the entry node and the initial state of A with the empty substitution, and computes all the possible states in A and all corresponding substitutions that can be obtained through this execution. Thus, the algorithm computes the reachable triples $\langle n, a, s \rangle$ consisting of a node n in the CFG, a state a in A and a substitution s. Note that several substitutions may correspond to a node n and a state a for two reasons: (1) n and a can be reached through different paths in the CFG, leading to different substitutions, and (2) different substitutions may match a same regular path expression to a same path in the CFG if some pattern variables are instantiated by negative patterns. At each step, the algorithm computes a match (linear in the size of a CFG label) and a substitution merge (linear in the number of pattern variables) to recompute the current substitution. Therefore, the running time of this worklist algorithm is $O(CFG \times A \times substs \times (pvars + labelsize))$, where $pvars$ is the number of pattern variables, $labelsize$ is the size of CFG node labels, and $substs$ is the number of different substitutions that may correspond to a same node and a same state, which is of the order $vars^{pvars}$ if there are $vars$ variables in the program that may become instances of pattern variables.

As our goal is to integrate user property checking into normal compilation, we consider that this algorithm is not efficient enough to lead to an acceptable performance overhead. This is why we decided to restrict regular path queries to a class that is checkable in linear time and space.

We can now precisely define the grammar of our minimalist language in BNF form:

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

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

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

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

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 C code fragment, and $expr$ a C expression. 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 3.4.

Let us now define the precise meaning of a CRQ as a regular path expression. A CRQ of the form 'Is there a path from a statement of type f to a statement of type t avoiding statements of type v, successful tests of type v_t and unsuccessful tests of type v_e?' is written in Condate as 'from f to t avoid v or $+v_t$ or $-v_e$', and corresponds to the regular path expression $f[\hat{}v + v_t - v_e]^* t$.

The automaton equivalent to a CRQ is shown in Figure 1, and has a fixed size. This reduces the factor A in the time complexity to a constant. We already saw that in spite of this restriction, many useful properties can be expressed.

Furthermore, we impose the restriction that all pattern variables occurring in a CRQ 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 there is at most a single substitution instantiating a CRQ to a path in the CFG (because all pattern vari-

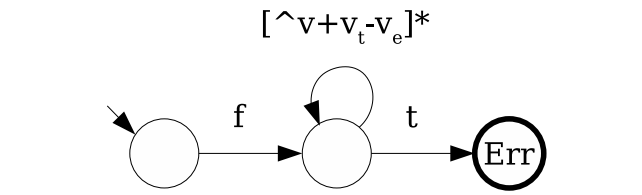

Figure 1. Automaton equivalent to a CRQ.

ables are bound in the first edge of the path). This reduces the factor $substs$ in the time complexity to the number of CRQ *instances*, that is, the number of different substitutions that instantiate the f pattern to "from" statements in the program.

This second 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 CRQ 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.

5. Standalone Implementation

Due to these two restrictions, a CRQ *instance* is checkable in time $O(CFG \times (pvars + labelsize))$. Considering that the number of meta-variables is a constant in practice (frequently 2 or 3), the only super-linear factor comes from the size of the statements in the program $labelsize$. Section 6 will show how this factor can be improved.

The worklist algorithm cited above uses a non-linear amount of space, because it associates each node in the CFG with different automaton states and variable values. In the particular case of CRQs, linear space usage can be achieved with no penalty on the time complexity if CRQ instances are checked independently, as implemented by the algorithm below. The algorithm takes a CFG and a $CRQ = \langle from, to, avoid \rangle$ consisting of a triple of (possibly disjunctive) patterns:

```
proc check(CFG, from, to, avoid)
  substs ← ∅                                   // collect CRQ instances
  foreach node t ∈ CFG do
    global_store ← ∅                           // empty substitution
    match(t, from)                             // instantiating global store
    substs ← substs ∪ {global_store}
  end
  for subst ∈ substs do                        // check one CRQ instance
      global_store ← subst                     // instantiate variables
      list ← []
      foreach node t ∈ CFG do
        // put "from" nodes of the instance on the list
        if match(t, from) then list ← [t | list] fi
      end
      // traverse instantiated CFG
      while list = [t | rest] do
            list ← rest
            if ¬visited(t)
              visited(t) ← true
              if match(t, to)
                print "reached t"
              elsif ¬match(t, avoid)
                    foreach edge e = t → t' do
                      if ¬match(e, avoid)
                        then list ← [t' | list]
                      fi
                    end
              fi
      end
  end
end
```

In a first traversal, the algorithm scans all the program for "from" statements, and collects the number of different substitutions associated to them. Each such substitution represents an instance of the CRQ, which is then checked in two subsequent passes. The first pass over an instance collects "from" nodes. These constitute the initial worklist for the second pass over an instance, which traverses the CFG from "from" nodes to "to" nodes using only edges matching the via pattern. For any "to" node that is reached, there exists a path satisfying the CRQ. The space used by the algorithm is of only one bit per CFG node, to recognize already visited nodes, plus a global store for pattern variables, recording their substitution for the current instance of the CRQ. However, to compute the set of all instances of a CRQ during the first traversal of the algorithm, a set of global stores is needed. New CRQ instances are added only if their stores differently instantiate the "from" pattern.

6. Integrated Implementation

When implementing the checking algorithm inside a concrete compiler, we encountered some interesting challenges, but also some interesting opportunities for cross-fertilization.

We chose gcc as our base because we wanted to demonstrate that the compiler-integrated approach can be easily incorporated into any existing compiler. This is why 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.

6.1 Dealing with temporaries

The main difficulty we encountered was that analyses and optimizations within gcc are performed on a simplified internal representation of the AST called GIMPLE [26]. In GIMPLE, expressions are broken down 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.

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.

6.2 On-demand temporary inlining

Interestingly, some inlining can be avoided if the intermediate representation has been constructed using a form of global value numbering [31]. 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 previous code fragment 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 CRQ '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, $patternsize$.

7. Assessment

We entirely implemented the Condate language in our prototype mygcc. Mygcc is a fully functional gcc version that performs (intra-procedural) checking as an additional compilation step. In terms of user interface, we just added a new gcc flag "–tree-check=file" to specify a file containing CRQ definitions. Mygcc is freely available in binary form [27]. We are currently discussing with the gcc development team to incorporate source changes into the official gcc release.

In order to validate and evaluate assess the Condate language, we applied mygcc to reproduce the detection of some previously reported bugs in the Linux kernel. These bugs were reported by a previous study [7] that used 12 different user checkers written in Metal for the MC tool and detected 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 [25]. Checks in Metal are considerably more powerful than CRQs — they are expressed as arbitrary automata mixed with executable C code.

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

- test the expressiveness of the language by expressing a maximum number of checks
- test the precision of the language 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

7.1 Expressiveness

To asses the expressiveness of Condate, we expressed as CRQs as much as possible of the 12 MC checkers cited above. The results are given in Table 1.

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 CRQ because it encodes complex checks about pointer uses and the assumptions that can be deduced from them.

An interesting comment about the expressiveness of CRQ-based properties is that this framework allows to optimize the checkers by factorizing similar user properties in the same CRQ. 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 CRQ:

> 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)".

Theoretically, 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.

7.2 Precision

To test the precision of mygcc, 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.

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

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 loose 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(X) && !INT_CST_LT(Y, size(typeof(X)))

Table 1. Expressiveness comparison between Metal and Condate.

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 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. In addition to the bugs previously reported, mygcc found four new bugs.

7.2.1 Missed bugs

Two of the bugs missed by Condate 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.

7.2.2 False positives

With respect to MC, mygcc found only two 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.

7.2.3 Different diagnostics

For two bugs that were found both by 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.

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 CRQ as a "to" pattern, like other dereferences.

7.2.4 New bugs

Mygcc also found four new bugs not previously reported by the MC study. The first one is 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, signaled by mygcc. In reality, even the pointer "mymtd" would be null in this case, but mygcc does not notice this detail. We do not know why MC overlooked this bug.

Another 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.

Finally, mygcc found another new bug found 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 checker as a dereference.

7.2.5 Automatic vs. manual inspection

From the practical point of view, it is important to note that some of the real errors found by both MC and mygcc are not at all easy to find by manual inspection, due to the contrived control path that should be followed. For example, there is a bug in file "partition.c", function "udf_fill_spartable", in the following code fragment:

```
for (i=0; i<rtl; i++)
{
  if (index > sb->s_blocksize)
  {
    udf_release_data(bh);
    bh = udf_tread(sb, ++spartable,
                   sb->s_blocksize);
    if (!bh)
    {
      sdata->s_spar_loc[i] = 0;
      continue;
    }
    index = 0;
  }
  se = (SparingEntry *)&(bh->b_data[index]
  ...
}
```

In this code fragment, the pointer "bh" is assigned to the result of function "udf_tread()" that may return null, but the subsequent test seems to check "bh" properly. However, the problem is that if the pointer is null, the continue statement is executed to restart the loop and in the next iteration the outer-level conditional may be skipped altogether, to hit the dereference "bh->b_data[index]" on the last line. This example is a typical case where manual inspection is probably not sufficient.

7.3 Performance

The examples described in the above 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. Beyond that, mygcc aims not just at being scalable to large programs, but to impose a reasonable overhead on compilation time. We measured the overhead of the three Condate checkers presented above when compiling three programs that are part of the Linux study.

The checking overhead is directly related to the number of checkers used, to the number of CRQ instances found in the program, and to the size of the patterns. However, checking time never exceeded 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%.

When interpreting these numbers, it is important to note that we did not have the time to optimize the implementation of the current prototype.

8. Related work

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, or Checkstyle [5], in which user-defined checks have to be coded in C++, respectively in Java. More recently, some extensible code inspectors such as PMD [29] 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 a significant amount of code.

Tools defining a DSL to write code checkers include CodeCheck [9], tawk [19], defining an imperative language close to C, Genoa [12] 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 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, 32] and CQUAL [17] are based on extensible type checking. In this approach users must annotate the native types of the programing language with qualifiers in order to express new classes of 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 adopted in a compiler-integrated approach, but for now are implemented as standalone tools. Some extensions were added to CQUAL to check flow-sensitive types [18], but in this case the performance is probably no more suitable for compiler-integrated 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 or illegal 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 an early tool called Cesar [28] for checking Fortran and his evolution called FLAVERS [8] for Ada and Java, MC [14] for C and its variant MJ [2] for Java, MOPS [6] and Uno [22] for C, PQL [24] for Java and CodeSurfer Path Inspector [10] for C. 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 [1] in C code. The running time complexity of these tools has been precisely analyzed in the framework of parametric regular path queries [23]. 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. One original feature of Condate is allowing to define transitions that depend on variable values, using the "+" and "-" patterns. On the other hand, 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 paths, by using for example symbolic computation, such as in PREfix [4] and its successor PREfast. Among path-sensitive checkers, SLAM [3], Blast [20], and ESP [13], 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" [21], 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.

CRQs used in Condate are a variant of parametric regular path queries [23], 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 check-specific front-end
- we allow to encode a minimal amount of dataflow information in regular expressions, through the "+" and "-" labels; this significantly augments their expressiveness.

Typestate verification [16] 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.

9. Conclusions

Condate makes it possible for every programmer to define new checks performed internally by the compiler, without a prohibitive overhead of doing so. However, there is still much room for performance improvement.

User-defined checks may increase the confidence of a programmer with respect to his or hers code, especially if used on a continuous basis during development. The checks expressible today are quite limited but non-trivial, as they refer simultaneously to syntax, semantics, control and data flow. At the syntactic level, checks in Condate are very easy to write and to understand, because writing patterns in concrete syntax does not require deep knowledge about the AST internal representation in the compiler, nor of any API to traverse it. The control and data flow level are very intuitively expressed as reachability under constraints using variables shared by the patterns. Of course, adding semantic constraints in the pattern requires knowledge of the functions internal to the compiler, but our experiment showed that semantic patterns less frequently needed.

The language can be extended in different ways. First, more comfort could be added for the programmer by defining macros reusable between CRQs. Then, new kinds of checks could be added, either by defining other classes of regular path expressions that can be checked efficiently, or by integrating for instance checks base on type qualifiers into the language. Finally, the language could be integrated in the subject programming language (C in our example) to define for example checks exported by a library aside its normal interface.

Another open research avenue is to explore further the possible interactions and cross-fertilization between checking on one hand and program analyses, optimizations and transformations on the other hand.

Today, most often programmers have to adapt themselves to compiler idiosyncrasies. Tomorrow this will surely have to change. Condate is a particular point in a wide spectrum of possible designs. We can probably foresee a future trend of more semantic-enabled and more user-centric compilers, obtained by fusing together compilation with other powerful analyzers. In this perspective, we can already experiment with different proto-languages for that confluence.

A. Complete Condate listing

The complete checkers used by mygcc to performs 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
      -"\(\(%X == 0B\) || %_\)"
```

Acknowledgments

The author would like to thank Nathalie Deleau for continuous support during this work. Thanks also to Paco Babane for his essential contribution in the implementation, and to Romeo for useful help when typesetting the paper.

References

[1] K. Ashcraft, D. Engler. "Using Programmer-Written Compiler Extensions to Catch Security Holes". In Proc. IEEE Symp. on Security and Privacy. May 2002.

[2] G. Back, D. Engler. "MJ - a system for constructing bug-finding analyses for Java". Technical report, Stanford University. September 2003.

[3] T. Ball, S. Rajamani. "The SLAM Toolkit". In Proceedings of the 13th International Conference on Computer Aided Verification. LNCS Vol. 2102. 2001.

[4] W. R. Bush, J. D. Pincus, and D. J. Sielaff. "A static analyzer for finding dynamic programming errors". Software - Practice and Experience, vol. 30(7). 2000.

[5] Checkstyle. Open-source project at SourceForge.net. http://checkstyle.sourceforge.net

[6] 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.

[7] 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.

[8] J. Cobleigh, L. Clarke, L. Osterweil. "FLAVERS: A finite state verification technique for software systems". IBM Systems Journal, 41(1). 2002.

[9] CodeCheck. Abraxas Software, Inc. http://www.abxsoft.com

[10] CodeSurfer Path Inspector. Gramma Tech. http://www.grammatech.com

[11] R. Crew. "ASTLOG: A Language for Examining Abstract Syntax Trees". In USENIX Conference on Domain-Specific Languages. October 1997.

[12] P. Devanbu. "GENOA — a customizable, front-end-retargetable source code analysis framework". ACM Transactions on Software Engineering and Methodology (TOSEM) vol 8(2). April 1999.

[13] 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.

[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. 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.

[17] 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.

[18] 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.

[19] W. Griswold, D. Atkinson, C. McCurdy. "Fast, Flexible Syntactic Pattern Matching and Processing". In 4th International Workshop on Program Comprehension. 1996.

[20] 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.

[21] 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.

[22] G. Holzmann. "Static source code checking for user-defined proper-

ties." In World Conference on Integrated Design and Process Technology, Pasadena, CA, June 2002. Society for Design and Process Science.

[23] Y. Liu, T. Rothamel, F. Yu, S. Stoller, N. Hu. "Parametric regular path queries". ACM SIGPLAN Notices, 39(6) (PLDI). May 2004.

[24] 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.

[25] MC bug viewer. http://metacomp.stanford.edu

[26] J. Merill, "GENERIC and GIMPLE: A New Tree Representation for Entire Functions". Proc. of the GCC 2003 Summit.

[27] Mygcc prototype and documentation. http://mygcc.free.fr

[28] K. Olender , L. Osterweil. "Cesar: a static sequencing constraint analyzer". ACM SIGSOFT Software Engineering Notes 14(8). December 1989.

[29] PMD. Open-source project at SourceForge.net. http://pmd.sourceforge.net/

[30] T. Reps. "Program analysis via graph reachability". Information and Software Technology 40(11-12). November/December 1998.

[31] 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.

[32] Splint. Open-source project. http://www.splint.org

[33] N. Volanschi. "Unparsed patterns: integrating unrestricted, concrete syntax pattern matching in any compiler". January 2006. Submitted for publication.

[34] N. Volanschi. "A Portable Compiler-Integrated Approach to Permanent Checking". In Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE), Tokyo. September 2006. To appear.