Scalable Security Verification of Software at Compile Time

Syrine Tlili∗, José M. Fernandez †, Abdefettah Belghith∗, Bilel Dridi‡ and Soufien Hidouri‡

∗Hana Research Laboratory
University of Manouba, Tunis, Tunisia
syrine.tlili@hanalab.org
† Ecole Polytechnique de Montréal
Montréal, Canada
jose.fernandez@polymtl.ca
‡ Université de Carthage
Tunis, Tunisia
bilel.dridi@netc.fr, hidourisoufien@gmail.com

Abstract—Automated verification tools are required to detect coding errors that may lead to severe software vulnerabilities. However, the usage of these tools is still not well integrated into software development life cycle. In this paper, we present our approach that brings the software compilation process and security verification to a meeting point where both can be applied simultaneously in a user-friendly manner. Our security verification engine is implemented as a new GCC pass that can be enabled via flag -fsecurity-check=checks.xml where the input XML file contains a set of user-defined security checks. The verification operates on the GIMPLE intermediate representation of source code that is language and platform independent. The conducted experiments demonstrate the scalability, efficiency and performance of our engine used to verify large scale software, especially the entire Linux kernel source code.

Keywords—Static Analysis; GCC; Security Verification; Finite State Automata.

I. INTRODUCTION

Despite the availability of a large range of software checking tools [1]-[9], their usage is still below the increasing requirements of software security and quality. The dissociation of these tools from software development environments is considered as a major limitation for their large scale adoption. They often require additional effort and time that dissuade programmers from applying them to verify the security of their source code. Besides, some of these tools are distributed under commercial licenses not at hand of a large community of programmers [10], [11].

In this context, we present our open source verification engine implemented as an extension to GCC compiler. The integration of our engine within the compilation process facilitates its large deployment during software development with reduced time and effort. The security analysis is simply activated by enabling the -fsecurity-check flag that takes as input an XML file containing user-defined security checks. The analysis operates on the GIMPLE intermediate representation of source code provided by the TREE-SSA.
with an XML input file of user-defined security properties.

3) We detail the XML specification of security properties modeled as extended finite state automata that track security relevant GIMPLE statements and operands.

4) We demonstrate the scalability, efficiency, and performance of our extended GCC compiler by using it to check the entire Linux Kernel source code.

The rest of this paper is organized as follows: Section II presents the main constructs of GIMPLE Intermediate representation for GCC. Section III is dedicated to the formal specification of security properties. In Section IV, we detail our annotation technique that handles GIMPLE temporary variables, aliasing, and parameter passing. A detailed presentation of our analysis algorithms is provided in Section V. We demonstrate the efficiency and scalability of our approach through experiments detailed in Section VI. Section VII discusses related work and Section VIII concludes.

II. USING GIMPLE IR

This section gives an overview of GIMPLE tree representation of source code on which we based our security analysis.

A. GIMPLE SSA Form

Starting from version 4, GCC is enriched with Tree SSA an optimization framework based on the Static Single Assignment (SSA) form [12]. More specifically, Tree SSA operates on GIMPLE intermediate representation of source code that is language and target independent. In the GIMPLE form, complex control flow structures are linearized into conditional gotos and statements contain no more than three operands except for function calls. Besides, the SSA form of GIMPLE entails that each variable is defined exactly once before being used. Hence, different versions are created for the same variable involved in multiple assignment operations. Temporary variables are also created to hold intermediate values returned by GIMPLE statements.

Listing 1 and Listing 2 illustrate a sample C program with its corresponding GIMPLE representation in SSA form, respectively. The GIMPLE form introduces a set of temporary variables either by defining new variables to hold intermediate values such as variables _5 and _7, or by appending a version number to declared variables at each assignment operation such as variable q_3 and t_6. Besides, GIMPLE IR introduces PHI<...> nodes to handle branching statements where a given variable may have different values depending on the followed branches. In the GIMPLE IR, the return value s_1 of function foo may either be variable s_4(D) (3) if basic block <bb 3> is visited or variable p.0_11 (4) if basic block <bb 4> is visited. The suffix (D) appended to variable s_4(D) means that the variable is defined outside the function scope which is always the case for function parameters. We take advantage of PHI<...> nodes in order to implement a flow-sensitive analysis where security annotations assigned to tracked variables may change from one program point to another. The GIMPLE code of Listing 2 will be used later in Section V to illustrate our approach.

```
int * p;
int * foo(int *s)
{ printf("%d",*s);
  if (*s>5)
    free(s);
  else {
    p = (*int *)malloc(sizeof(int));
    s=p;
  }
  return s;
}
int main (int a){
  int * q;
  int *t;
  q=(int*)malloc(sizeof(int));
p=q;
t=foo(p);
  *q=6;
  free(t);
  return 1;
}
```

Listing 1. Sample C Code

```
main (int a){
  int * t;
  int * q;
  <bb 2>:
    q_3 = malloc (4);
    p = q_3;
    t_6 = foo (q_3);
    *q_3 = 6;
    free (t_6);
    return 1;
  }
  foo (int * s){
    void * p.0;
    int _5;
    int _7;
    <bb 2>:
      _5 = *s_4(D);
      printf("%d", _5);
      _7 = *s_4(D);
      if (_7 > 5)
        goto <bb 3>;
      else
        goto <bb 4>;
    <bb 3>:
    free (s_4(D));
    goto <bb 5>;
    <bb 4>:
      p.0_10 = malloc (4);
      p = p.0_10;
      <bb 5>:
        * s_1 = PHI <s_4(D) (3), p.0_10 (4)>
        return s_1;
    }
```

Listing 2. Sample GIMPLE IR
B. GIMPLE Statements and Expressions

GIMPLE statements and expressions have a tree-structured representation defined, respectively in files gimple.def and tree.def of the GCC distribution. Listing 3 illustrates an excerpt of file gimple.def where each GIMPLE statement is identified by a GSCODE: GIMPLE_CALL for function calls, GIMPLE_ASSIGN for assignment operations, GIMPLE_COND for conditional statements, and GIMPLE_RETURN for return statements. Each statement structure contains an ordered set of operators and operands identified by a GIMPLE tree code DEFTREECODE as illustrated in the excerpt of tree.def file in Listing 4.

Listing 3. Subset of GIMPLE Statements

A GIMPLE operand can either be a simple expression or a compound expression involving more than one operand. For instance, a VAR_DECL stands for a single variable declaration, an INDIRECT_REF compound expression has one operand that stands for the dereferenced pointer. POINTER_PLUS_EXPR expressions represent pointer arithmetic where the first operand is a pointer and the second is of integer type. The COMPONENT_REF aggregate expression encompasses three operands: the structure operand, the accessed field, and the field offset.

Listing 4. Subset of GIMPLE Operands

Security properties are defined as guarded finite state automata with labeled transitions that monitor the sequencing of security relevant program statements and decorate involved program variables with security annotations. Security automata are executed while performing a DFS traversal of the analyzed program control flow graph in GIMPLE form. Hence, each automaton transition is labeled with: (i) the GIMPLE code of the security relevant statement, (ii) a transition enabling boolean guard over monitored GIMPLE variables, (iii) an annotation update action of monitored variables, and (iv) an error reporting action executed when a security property is violated.

Formally, an extended security property automaton is a tuple $PA = (Q, \Sigma, \Delta, E, q_0, q_1)$ where:

- $Q$ is a finite set of automaton states, of which one state $q_0$ is the initial state and one state $q_1$ is the final state.
- $\Sigma \subseteq G_{stmt} \times G_{opd}$ is the input alphabet where $G_{stmt}$ is the set of security relevant GIMPLE statements and $G_{opd}$ the set of monitored GIMPLE operands.
- $Annot$ is the set of security annotations assigned to monitored GIMPLE operands.
- $\Delta \subseteq Act \times (G_{opd} \cup Annot)^*$ is the output alphabet where $Act = (Set \cup Err)$ encompasses: (i) annotating actions $Set$ that assign security annotations to monitored variables and (ii) error reporting actions $Err$ triggered when reaching security property violations.
- $G$ is the set of boolean guards that determine if the transition is enabled. Each guard is a predicate over attributes of monitored GIMPLE variables such as annotations, GIMPLE codes, and declared types.
- $E \subseteq Q \times \Sigma^* \times G \times Q \times \Delta^*$ is a finite set of edges, written $q \xrightarrow{s,g/a} q'$ where $s \in \Sigma^*$, $g \in G$, $a \in \Delta^*$.

The current configuration of an extended security automaton is described by a triple:

$$(q, s, a) \in Q \times \Sigma^* \times \Delta^*$$

where $q$ is the current state, $s$ is the remaining sequence of GIMPLE statements, and $a$ is the output produced so far.

We define the binary relation $\rightarrow_{PA}$ on $Q \times \Sigma^* \times \Delta^*$ as follows: For all $p, q \in Q$, $s, s' \in \Sigma^*$, $a, a' \in \Delta^*$, $g \in G$ if $g$ evaluates to $true$ and $(p, g, s, a', q) \in E$ then

...
(p, ss', a) \vdash_{PA} (q, s', a')

Let $\vdash_{PA}'$ be the transitive and reflexive closure of $\vdash_{PA}$, the function $PA : \Sigma^* \rightarrow 2^{\Delta^*}$ is defined such as for every $w \in \Sigma^*$,

$$PA(w) = \{ a \in \Delta^* | (q_0, w, \epsilon) \vdash_{PA}' (q_1, \epsilon, a) \}$$

During the analysis of a given program, the security automaton $PA$ makes a sequence of moves from configurations to configurations. A violation of the $PA$ property occurs when the automaton reaches the final state $q_1$ with an output $v \in V_{PA}(w)$ where $V_{PA}(w) \subseteq PA(w)$, $w \in \Sigma^*$ and is defined as follows:

$$V_{PA}(w) = \{ a, e \in \Delta^* \cup Err | (q_0, w, \epsilon) \vdash_{PA}' (q_1, \epsilon, a, e) \}$$

Let $T \subseteq \Sigma^*$ denote the set of all possible execution traces of the analyzed program starting from its entry point to the exit, we have:

$$PA(T) = \bigcup_{t \in T} PA(t)$$

$$V_{PA}(T) = \bigcup_{t \in T} V_{PA}(t)$$

A security violation is detected, if there exists some $t \in T$ such that $(q_0, t, \epsilon) \vdash_{PA}' (q_1, \epsilon, a)$ where $a \in V_{PA}(t)$, this implies that:

$$PA(T) \cap V_{PA}(T) \neq \emptyset$$

The main feature of our approach is that it operates on a compiler-generated representation of source code that captures all control and data flow information of the analyzed program. This enables our analysis to exhaustively parse all possible execution paths in order to detect all coding errors that may lead to security vulnerabilities. As for all static analysis techniques, we may generate false positives as we do not discard infeasible paths. Nevertheless, our interprocedural approach takes advantage of valuable information computed by GCC in order to reduce the rate of false positives such as alias relations and data type information. The soundness of our approach is provided under some conditions where function to pointers, threads and jumps are not used.

B. XML Specification of Automata

We formally specify a security automaton in XML so that all of its components are clearly stated: the states, the labeled transitions, the tracked GIMPLE expressions, the security annotations, and the boolean guards on transitions.

We advocate the use of the XML language since the XML tree-structured representation is a well-adapted fit to the tree-structured GIMPLE representation of statements and expressions listed in file gimple.def and tree.def of GCC distribution.

Consider the graphical illustration of the Null Check security automaton of Figure 1 and its XML specification in Listing 5. The automaton transits from node 0 to node 2 when matching GIMPLE_CALL statements to memory allocation functions and decorates its lhs operand with a null annotation. When reaching a GIMPLE_COND statement involving a null annotated pointer, the automaton transits to the final node 1 with no error reporting. On the other hand, the dereferencing of a null pointer at the lhs or the rhs of a GIMPLE_ASSIGN statement triggers a transition to the final node with an ERROR_REPORTING output of the security automaton.

```
<automaton name="Null Check" status="1">
  <startNode> 0 </startNode>
  <transition src="0" dst="2">
    <stmt gimple="GIMPLE_CALL(malloc)"/>
    <FN fn="malloc"/>
  </transition>
  <transition src="2" dst="1">
    <stmt gimple="GIMPLE_ASSIGN(chk)"/>
    <LHS chk_hist="null" set_hist="chk"/>
    <RHS code="indirect_ref" index="0"/>
    <FN fn="malloc"/>
  </transition>
  <transition src="1" dst="0">
    <stmt gimple="GIMPLE_COND("chk",err)" check "err">
      <LHS chk_hist="null" set_hist="chk"/>
      <RHS code="chk" index="0"/>
    </stmt>
  </transition>
</automaton>
```

Listing 5. Null Check Automaton

To perform the security checking, the gcc compiler is feed with a list of security properties defined in an XML file starting with a top-level element named <policy>. Inside this root element, each security property automaton is mapped into XML constructs encompassing the following elements:

1) Element <automaton> contains a security automaton specification identified by a name attribute. When, the status attribute is set to 1, the automaton property is verified during the security analysis, otherwise the property is disregarded.
2) Elements <startNode> and <exitNode> represent the unique initial node and final node, respectively. Element <node> is used to specify additional automaton nodes with unique integer identifiers. The start node identifier is set to 0, whereas the identifier of the exit node is set to 1.
3) Element <transition> defines a link between a src node and a dst node with a gcode attribute that matches the GIMPLE code of the tracked statement.
4) Element <FN> is used within a GIMPLE_CALL transition with a decl_name attribute that matches the name of the tracked function.
5) Element <ARG> denotes an argument operand of a function call. The value of the index attribute stands for its position in the argument chain.
6) Element <LHS> is used when tracking the left-hand-side operand of a GIMPLE assignment operation or the return operand of a function call.
7) Element <RHS1> and <RHS2> respectively stands for the first rhs operand and the second rhs operand, if any.

Elements <ARG>, <LHS>, <RHS1>, and <RHS2> have a gcode attribute that matches GIMPLE codes of tracked GIMPLE expressions as listed in file tree.def of GCC distribution. Moreover, to allow a fine-grained tracking of GIMPLE expressions, we define the index attribute to pinpoint a specific operand of compound expressions. For instance, in line 12 of Listing 5, we track the pointer operand of a dereference operation.

During automaton transitions, tracked operands may be decorated with relevant annotations defined in the XML set_annot attribute. Moreover, the optional chk_annot attribute defines a boolean guard whose value should match the annotation of the tracked operand in order to trigger the related automaton transition, otherwise the transition is not activated. When both attributes are defined, the chk_annot attribute is considered before the set_annot attribute during the analysis. For instance, the transition from node 2 to node 3 is triggered when matching a GIMPLE_COND statement with a null annotated operand as stated in the chk_annot attribute, otherwise the automaton remains in the same state. We also define some other boolean guard on other attributes of GIMPLE variables such as chk_code to target a specific GIMPLE expression code, and chk_type to target a specific data type. Basically, we extend the list of XML guards to cover all attributes of GIMPLE variables provided in GCC.

IV. ANNOTATING GIMPLE OPERANDS

This section details the technique we use to decorate program variables with security-relevant annotation. More specifically, we illustrate our approach to deal with temporary variables, alias relationships and parameter passing at function calls.

A. Dealing with temporary variables

The GIMPLE SSA form introduces temporary and versioning variables that render the usage of syntactic pattern matching unsuitable. Thus, we advocate an annotation technique that decorates GIMPLE variables based on their unique identifiers DECL_UID and their version numbers SSA_NAME_VERSION.

All GIMPLE temporary variables that are derived from the same initial variable have the same DECL_UID but different version numbers each time used as lhs operands. As such, each version of a given variable can have a flow-sensitive security annotation that may change from one program point to another. Besides, GIMPLE introduces PHI<> nodes to represent all possible versions of a given variable at the end of branching statements. This enables our approach assigning more than one annotation to a given variable according to the followed branches.

To efficiently annotate tracked GIMPLE variables, we define a hashtable indexed by a string identifier that combines the DECL_UID and the version number SSA_NAME_VERSION of a given GIMPLE variable. Each entry in the hashtable contains the security annotation of the indexed GIMPLE variable. More specifically, an annotation encompasses: (i) a user-defined string specified in the set_annot attribute of the automaton XML specification and (ii) the index of the security automaton that is tracking the variable. Note that a given GIMPLE variable can be tracked by a single automaton at a time. Nevertheless, a single automaton may monitor a set of variables involved in an aliasing relationship as detailed in the following section.

B. Dealing with Aliasing

Many insidious security violations are hard to detect because of variables aliasing pitfalls. To achieve a better precision, we advocate the usage of the points-to analysis computed by the GCC compiler. Thus, we query the GCC alias oracle refs_may_alias_p () to verify whether two GIMPLE variables are referencing the same memory location. This approach allows us to propagate security annotation through aliasing information that is consistent with compiler generated data-flow information. Nevertheless, the GCC alias oracle detects implicit and explicit aliasing relations based on too conservative assumptions that may lead to a high rate of false positives. For instance, all fields of a structure are considered as aliased memory references.

To reduce false positives, we define an option to our analysis where it can be configured to walk SSA immediate-use chains in order to extract explicit aliasing relations induced by assignment operations. To do this, we use the FOR_EACH_IMM_USE_STMT and FOR_EACH_IMM_USE_ON_STMT macros of GCC in function alias_inference defined in Listing 6.
Flow-sensitive annotations are spread from right-hand-side operands to left-hand-side operands. More specifically, we target assignment operations involving unary rhs operands such as: $p = q$, $p = \ast q$, $p = k.q$, $p.l = q$, $p = q.l$, $p \rightarrow l = q$, and $p = q \rightarrow l$. Moreover, PHI nodes are introduced in GIMPLE IR as internal statements that start with a # symbol. These PHI nodes are also considered as immediate-use statements that require special attention when dealing with aliasing.

In fact, we faced multiple challenges related to PHI nodes and type conversion operations at GIMPLE assignment rhs. We illustrate these difficulties through a fragment of the GIMPLE IR of file /crypto/ecdh/ech_osssl.c from openssl-1.0.1d in Listing 7.

```
<bb 8>:
_25 = strien (password_8);
bufsiz_26 = (int) _25;
if (bufsiz_26 > bufsiz_27(D))
goto <bb 9>;
else
  goto <bb 10>;
<bb 9>:
<bb 10>:
#bufsiz_1=PHI<bufsiz_26(8),bufsiz_27(D)(9)>
_28 = (long unsigned int) bufsiz_1;
memcpy (buf_29(D), password_8, _28);
```

Listing 7. GIMPLE IR from Openssl

We checked the openssl-1.0.1d distribution against the Heartbleed automaton of Figure 2 in Section V. At the conditional statement, the rhs operand bufsiz_26 is assigned a chk annotation and the lhs operand bufsiz_27(D) has no annotation. These operands are then used in a PHI node containing possible values of variable bufsiz_1 depending on whether bb 9 or bb 10 is traversed. Following the false branch, we propagate the chk annotation of bufsiz_26(8) to bufsiz_1. On the true branch traversing bb 9, the variable will get no annotation and stands for an unchecked variable. PHI variable bufsiz_1 is used as a rhs operand of an assignment operation to variable _28. However, GCC does not consider assignment operation with type conversion as an immediate-use stmt. Thus, function alias_inference will not propagate annotations from rhs to lhs causing a false positive alert on path <bb 8> to <bb 10> for using an unchecked size argument in function memcpy. To overcome this issue, we define a function assign_stmt_analysis that handles explicit assignment operations that are not immediate_use statement and propagates annotation from rhs to lhs.

The aforementioned analysis option may ignore an aliasing relationship that is computed by the GCC alias-oracle, but, it cannot claim an aliasing relationship that is not confirmed by the oracle. For more flexibility, users of our tool are free to disregard this option and refer to the conservative GCC alias-oracle.

C. Parameter passing

As mentioned earlier, security analysis is performed by traversing control flow graphs of analyzed functions. In the interprocedural mode, the analysis jumps from the caller cfg to the callee cfg where a DFS traversal is performed. Hence, our context-sensitive interprocedural analysis moves from the caller scope to the callee scope and propagates operands security annotations from caller to callee accordingly.

Listing 8 illustrates function sync_arg_call_decl defined in order to synchronize call arguments with declaration arguments at each call site. We use the function gimple_call_arg() to iterate over the call arguments of a GIMPLE_CALL statement. Security annotations of each call argument is transferred to the corresponding declaration arguments and stored in table htab. For each annotated declaration argument, we call function alias_inference to ensure that annotations are spread to all aliased variables declared in the callee function scope.

```
static void sync_arg_call_decl(tree node, 
gimple_call_stmt, int nops,htab_t htab){
  int i = 0;
tree argCall = NULL;
tree argDecl = DECL_ARGUMENTS (node);
while (argDecl && (i < nops)) {
    argCall = gimple_call_arg(call_stmt, i);
    set_annot(argCall,argDecl->annot,htab);
    argDecl = TREE_CHAIN (argDecl);
    i++;
}
}
```

Listing 8. Parameter and Annotation Passing
At function return, security annotations are propagated in the opposite way, i.e., from the caller function to the callee function as illustrated in function sync_arg_decl_call in Listing 8. However, the propagation is performed only for function arguments passed by reference. Otherwise, when arguments are passed by value, security annotations at the caller function scope remain unchanged.

V. ANALYSIS ALGORITHMS

In this section, we detail the intraprocedural and interprocedural algorithms implemented in our security analysis tool.

A. Security context

During our flow-sensitive analysis, program control flow graphs are checked against a set of security automata specified in XML and given as input to the extended GCC. Initially when the analysis begins, there is no annotated GIMPLE variables and no running security automata instances. A new instance of a given security automaton is created when its initial transition matches the current GIMPLE statement. The new instance transits from one state to another while traversing the analyzed program statements. During automaton transition, tracked GIMPLE operands may be assigned flow-sensitive security annotations.

During program traversal, we maintain a global security context sec_policy defined as a data structure that encompasses a vector of all automata instances and the hashtable htab storing security annotations of all tracked operands. The global security context is flow-sensitive and may change from one program point to another through the creation of new automata instances or state transitions of running instances.

B. Intraprocedural Analysis

Listing 9 details the algorithm of function dfs_block_traversal that performs a DFS traversal of the control flow graph of a given function func starting from a given basic_block bb down to the unique exit block, following the flow edges. When reaching the exit basic-block, the analysis recursively backwards to the latest branching point and traverses the rest of branching edges until all possible paths are followed. Notice that our analysis does not exclude paths that are actually infeasible at run-time.

We use the edge iterator ei to move from one basic block to its successors. At branching point, the function recursively follows all conditional edges taking as input a hashtable hbranch which is a copy of table htab containing security annotations of all security-relevant program variables. When reaching the end of a branching path, the analysis restores the latest version of the annotation table htab and the latest global security context sa_state before proceeding to the remaining branches.

In order to optimize the traversal, we define the block summary bb_sum which is a data set that contains a copy of the current global security context sa_state each time the basic block is entered. All basic-blocks summaries are stored in vector summaries_vec. The analysis of a basic block bb is performed only when the current global security context does not match any of the context copies stored in block summary bb_sum. When a match is found, the analysis aborts the traversal of the given basic block and backtracks to the latest branching point.

Within each visited basic block, we use the GIMPLE sequence iterator gsi to go through each statement and give it as input to function stmt_analysis.

```c
DFS_block_traversal(func, bb, summaries_vec, trace, htab, sec_policy) {
    edge_iterator ei;
    basic_block bb_succ;
    bb_summary bb_sum=NULL;
    htab_t hbranch=NULL;
    vec_policy_state sa_state=NULL;
    hbranch = annot_table_copy(htab);
    DFS_block_traversal(func, bb_succ, hbranch, summaries_vec, sec_policy);
    policy_restore(sec_policy, sa_state);
    if (!single_succ_p(bb)) {
        hbranch = annot_table_copy(htab);
        DFS_block_traversal(func, bb_succ, hbranch, summaries_vec, sec_policy);
    } else {
        trace->pop();
    }
}
```

Listing 9. DFS traversal
C. Interprocedural Analysis

In the interprocedural mode, the analysis jumps from the caller cfg to the callee cfg at each call site. The analysis of the caller function is resumed when returning from the callee function. Functions sync_arg_call_decl and sync_arg_call_decl in Section IV are used to ensure that relevant security annotations are consistently propagated from caller scope to callee scope and vice versa.

As for basic block analysis, we use function summaries in order to optimize programs analysis. However, function summaries are different from block summaries. In fact, a function summary contains two vectors: (i) a vector sum_in that stores security annotations of function arguments and global variables at function entry point, and (ii) a vector sum_out that stores security annotations of the function return operand and global variables. Thus, a callee function is analyzed only when the current security annotations of its arguments and global variables do not match previous annotations stored in vector sum_in. When a match occurs, instead of analyzing the callee body, we propagate the security annotations of the return operand and global variables from vector sum_out into the scope of the caller function and resume its analysis.

We illustrate our interprocedural analysis through the GIMPLE code of Listing 2 that is checked against the Null Check Automaton and the Heartbleed Automaton of Figure 2. Despite its severe impact, Heartbleed vulnerability of Openssl is caused by a common programming error described in CERT Coding rule ARR38-C [13] that can be easily avoided. Thus, the Heartbleed automaton tracks GIMPLE_CALL(memcpy) where the last argument that stands for the number of bytes to copy is not decorated with a chk annotation, i.e., it has not been checked before being used. If so, the automaton detects a security violation and terminates with an error reporting.

![Figure 2. Automaton for Heartbleed Vulnerability Detection](image)

We compiled the openssl-1.0.1d package with flag -fsecurity-check=checks.xml enabled in order to check the source code against the aforementioned Null Check and Heartbleed security automata. The compile-time verification of source code was performed in interprocedural mode and took 254 seconds. On the other hand, the compilation process with no enabled security checks took 218 seconds. Hence, our verification produces an execution overhead of 16%.

We were able to detect the Heartbleed vulnerability in line 2506 file of openssl-1.0.1d/ssl/t1_lib.c. We illustrate in Listing 10 a code fragment from its GIMPLE representation showing that variable _35 is not checked before being used in memcpy.

```c
#24 = payload_16 + 19;
#25 = (int) #24;
bp_27 = CRYPTO_malloc (#25, "t1_lib.c", \#2500);
*bp_27 = 2;
#29 = payload_16 >> 8;
...
#35 = (long unsigned int) payload_16;
memcpy (bp_34, pl_17, #35);
```

Listing 10. GIMPLE code of the Heartbleed vulnerability

VI. EXPERIMENTAL RESULTS

We fully prototyped our security analysis as an interprocedural pass of GCC version 4.8.2 encompassing two files tree-security-check.c and tree-security-check.h with a total of 3500 lines of C code added to the compiler. Some internal GCC files have been modified in order to integrate our extension within the GCC compilation process: tree-pass.h, passes.c, common.opt, and opts.c. In order to prove the efficiency and the scalability of our approach we used our extended GCC compiler on large scale software on a Core i7 6G RAM Linux machine.

A. Detecting real errors

To assess the capability in detecting real errors of our extended GCC compiler, we conduct an experiment where we checked the widely used cryptographic library Openssl against the Null Check Automaton and the Heartbleed Automaton of Figure 2. Despite its severe impact, Heartbleed vulnerability of Openssl is caused by a common programming error described in CERT Coding rule ARR38-C [13] that can be easily avoided. Thus, the Heartbleed automaton tracks GIMPLE_CALL(memcpy) where the last argument that stands for the number of bytes to copy is not decorated with a chk annotation, i.e., it has not been checked before being used. If so, the automaton detects a security violation and terminates with an error reporting.

![Image](image)

Table I

<table>
<thead>
<tr>
<th>Line</th>
<th>Variable, Annotation</th>
<th>Annotating function</th>
<th>Automata</th>
<th>Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>_29, null</td>
<td>stmt_analysis</td>
<td>NullChk: 0 → 2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>pnull</td>
<td>alias_inference</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>snull</td>
<td>sync_arg_call_decl</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>17</td>
<td>_5null</td>
<td>stmt_analysis</td>
<td>NullChk: 2 → 1</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>_29(D),null</td>
<td>sync_arg_call_decl</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>25</td>
<td>snull</td>
<td>alias_inference</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>28</td>
<td>p.0.11null</td>
<td>stmt_analysis</td>
<td>NullChk: 0 → 2</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>snull</td>
<td>alias_inference</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>31</td>
<td>s_1.f</td>
<td>alias_inference</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>31</td>
<td>t.6_dblf</td>
<td>sync_arg_decl_call</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>t.6_dblf</td>
<td>stmt_analysis</td>
<td>DbfFree: 0 → 2</td>
<td></td>
</tr>
</tbody>
</table>

Table I: Running Analysis on GIMPLE Code of Listing 2
Nevertheless, the verification of the Heartbleed automaton generates a high rate of false positives, around 80 alerts. Most of these false alerts come from the fact that programmers tend to make confident assumptions on the size of the manipulated data and omit to perform the required security checking. Some other alerts come from the usage of GIMPLE temporary variables as illustrated in the excerpt of file ech_ossl.c and its corresponding GIMPLE representation in Listing 11. The analysis decorates variable outlen.0_74 with a chk at line 203. It is then assigned a new value after being checked. As such, a new version outlen.0_76 is created with no chk annotation causing a false alert to be issued at line 205 when the false branch to basic block <bb 28> is followed.

```c
/*C Source Code
openssl-1.0.1d/crypto/ecdh/ech_ossl.c */
...
203 if (outlen > buflen)
204 outlen = buflen;
205 memcpy(out, buf, outlen);
...

/*GIMPLE representation*/
...
if (outlen.0_74 > buflen_50)
goto <bb 27>;
else
goto <bb 28>;
<bb 27>:
outlen = buflen_50;
<bb 28>:
outlen.0_76 = outlen;
memcp(out_68(D), buf_58, outlen.0_76);
...
```

Listing 11. Heartbleed False Positive Alerts

**B. Detecting new errors**

We also found a pointer Null Check error in file openssl-1.0.1d/apps/speed.c where the return pointer of a malloc function at line 2657 is not checked before being used at line 2670 as illustrated in Listing 12.

```
2657 fds = malloc_multi + sizeof *fds);
2658 for(n = 0; n < multi; ++n)
2659 {
2660 ...
2667 if(fork())
2668 {
2669 close(fd[1]);
2670 fds[n] = fd[0];
...
```

Listing 12. Null Check Error in OpenSSL

The best way to demonstrate the scalability of our analysis tool is to use it with a large software such as the entire Linux kernel. We have performed a null check analysis on an entire Linux kernel 3.8-rc6. First, we identified the set of memory allocation functions of the Linux kernel API. We found around 2000 functions memory allocation functions that we specified in our security property automaton. We run the interprocedural security analysis with the Null Check automaton while compiling Linux-3.8-rc6 source code. We were able to detect new coding errors that we reported to the Linux kernel community together with a suggested set of patches available at [14]. A excerpt of the aforementioned patch is illustrated in Listing 13.

```
---linux-3.8-rc6-vanilla/arch/x86/platform/efi/efi.c
@@ -924,6 +924,8 @@
 void __init
 efif_enter_virtual_mode(void)
 new_memmap = krealloc(new_memmap,
+ if (!new_memmap)
+ return -ENOMEM;
memcpy(new_memmap + (count * memmap.desc_size),
 md, memmap.desc_size);
 count++;
```

Listing 13. Patching Linux Kernel

The checking time of the entire Linux kernel is 7785 seconds whereas the compilation time with no security check took 7155. Hence, an overhead of 9% is induced by running the null pointer check during compilation process of the entire Linux kernel 3.8-rc6.

**VII. RELATED WORK**

This section presents different static analysis approaches and tools used to detect security vulnerabilities in source code.

Our approach is inspired from the MetaCompilation (MC) [1] static analysis tool that combines annotation-based and automata-based techniques for detecting vulnerabilities of source code. It provides a high-level automata language metal to define temporal security properties as finite state automata. The latter are executed while performing a flowsensitive DFS traversal of programs control flow graphs. A security violation is detected when automata reach the final state. MC analysis performs intraprocedural and interprocedural analysis in a context-sensitive way that takes into account aliasing relations. However, we perform a more precise alias analysis that increases the efficiency of our approach. The MC approach is implemented in a commercial tool Coverity as opposed to our extended GCC that is made available to the community under project e-munity in Sourceforge.net [15].

MOPS is another pioneer tool that uses pushdown model-checking technique to verify high-level security properties modeled as finite state automata. BLAST [5], SLAM [6] and SAT [7] are model-checking tools based on predicate abstraction that are mainly used to detect vulnerabilities in Windows. The full integration of these tools into the
compilation process is not straightforward since they require additional time and effort to generate program models that are suitable for their model-checking engine.

Some other tools such as CQUAL [2] and Splint [16] use a type-based approach to detect vulnerabilities in source code. These tools extend data types with qualifiers in order to express safety and invariant properties of programs. The verification process of these invariants can easily be integrated within the type-checking process at compile-time, however these tools are still decoupled from the software development life cycle. Besides, most of these tools require user modification of source code that may be cumbersome to programmers.

There are also other tools that extend the GCC compiler to perform security analysis such as mygcc [3] and GMC2 [4]. The compile-time analyzer mygcc defines a declarative language Condate to express user-defined security properties that are checked at runtime. It performs an intraprocedural analysis of source code and has been applied to check only some parts of the Linux Kernel.

VIII. Conclusion

We presented our interprocedural and flow-sensitive technique for detecting security vulnerabilities in source code. We implement this approach as an extension to GCC compiler in order to integrate the security verification process at early stages of the software development process. The GCC security pass is activated through the -fsecurity-check=securitpolicy.xml taking as input an input file containing a set of user-defined security properties specified in XML. In fact, our approach models security properties as extended finite state automata with guarded transitions that track security-relevant program actions and decorate program variables with security annotations. Our security extension operates on GIMPLE representation of source code. It performs a DFS traversal of GIMPLE control flow graphs while executing the security properties automata to detect security violations. We take advantage of alias and data types information generated by GCC to increase the precision of our approach and reduce the number of false positives. We conducted experiments on large scale software to demonstrate the efficiency of our extended compiler in detecting known errors and other new errors in openssl-1.0.1d and the Linux Kernel. We also demonstrate the scalability of our tool as we were able to verify an entire linux kernel source code. Our extended GCC compiler is entitled e-munity and is made available for download at Sourceforge.net [15].

REFERENCES


