Vanguard: Detecting Missing Checks for Prognosing Potential Vulnerabilities

Lingyun Situ, Linzhang Wang, Yang Liu, Bing Mao, Xuandong Li

Internetware 2018
ABSTRACT
It is challenging to have a general solution to precisely detect arbitrary vulnerabilities. Thus security research has focused on detecting specific types of vulnerabilities. Missing checks for untrusted inputs used in security-sensitive operations are one of the major causes of various serious vulnerabilities. Efficiently detecting missing checks is essential for identifying insufficient attack protections and prognosing potential vulnerabilities. This paper proposes a systematic static approach to detect missing checks for manipulable data used in security-sensitive operations in C/C++ programs. We first locate customized security-sensitive operations with lightweight static analysis; then judge assailability of sensitive data used in security-sensitive operations via static taint analysis; finally, assess the existence and risk degree of missing checks using static analysis. We have implemented the approach into an automated and cross-platform tool, named Vanguard, on top of Clang/LLVM 3.6.0. Experimental results on open-source projects have shown its effectiveness and efficiency. Furthermore, Vanguard has led us to uncover five known vulnerabilities and two unknown bugs.

CSCS CONCEPTS
• Security and privacy → Software security engineering; Vulnerability scanners;

KEYWORDS
Missing Checks, Static Analysis, Vulnerability Prognosis

1 INTRODUCTION
The whole field of software engineering is premised on writing correct code without vulnerabilities as well as defending attacks [28]. It is difficult to achieve in practice, especially for C/C++ programmers, because both languages force programmers to make fundamental decisions on handling security-sensitive operations like memory management. Besides, even experienced industrial developers will make mistakes during programming due to the lack of attention on attack protection details for security-sensitive operations.

To improve the correctness of software code, vulnerability detection plays one of the most important roles. Unfortunately, an automatic approach to precisely detect arbitrary types of vulnerabilities does not exist according to Rice’s theorem [33]. Thus, the state of art security research has focused on digging specific kinds of vulnerabilities buried in code, such as buffer overflow [12, 19], integer overflows [15, 42], use-after-free [22, 41], memory leakage [24, 37], out-of-bound errors [6, 20], by all kinds of static and dynamic approaches including static analysis [25, 40], taint analysis [9, 27], symbolic execution [8, 26], concolic execution [31, 32], model checking [4, 16], fuzzing [36, 38] etc.

However, missing checks for manipulable data used in security-sensitive operations are one of the major causes of various specific severe vulnerabilities including all the ones mentioned above. Furthermore, missing checks belong to “A7-Insufficient Attack Protection”, which has been proposed as a new type of Top 10 security risks by OWASP [5] in 2017. Therefore, efficiently identifying missing checks in realistic software is essential for identifying insufficient attack protections and prognosing potential vulnerabilities, especially in the early development stage.

Several approaches have been proposed to detect missing checks. Chucky [43] detects missing checks by a lightweight intra-procedural static taint analysis and anomaly detection algorithm. It identifies missing checks for security APIs usage based on the assumption that missing checks are rare events compared with the correct conditions imposed on security-critical objects in software. Therefore, it is more suitable for analyzing mature code since the assumption is usually not valid in early development stage. RoleCast [34] statically explores missing security authorization checks without explicit policy specifications in the source code of web applications. It exploits common software engineering patterns and a
role specific variable consistency analysis algorithm to detect missing authorization checks. However, RoleCast is tightly bounded to web applications coded in PHP and ASP.

To identify insufficient attack protections and prognosis potential vulnerabilities, we propose a systematic static approach to detect missing checks for manipulable data used in security-sensitive operations in C/C++ programs in this paper. It can be used for mature code as well as programs under development stage. First, customized security-sensitive operations (e.g., security-sensitive functions call, array-index access, and division and modular arithmetic) are located with lightweight static analysis on the abstract syntax tree [21], call graph (CG) [29] and control flow graph (CFG) [35] of the target program. Second, sensitive data used in the located security-sensitive operations are judged to see whether they are manipulable by outside attack inputs via static taint analysis including inter-procedural and intra-procedural taint analysis. Third, a data flow based backward analysis algorithm is applied to explore attack protection checks started from the locations of security-sensitive operations: if no protection checks exist, then a missing check is identified. Further, the risk degree of detected missing checks is assessed based on the context features. At last, details of the detected missing checks are reported as warnings.

We have developed an automated and cross-platform tool, named Vanguard, on top of Clang/LLVM 3.6.0. We have also conducted experiments on several open-source projects to demonstrate its effectiveness and efficiency. The results indicate that Vanguard is able to detect missing checks in open-source projects such as PHP, OpenSSL, Pidgin, Libpng, and Libtiff with low false positive (i.e., 19% in average) with low time overhead (e.g., 619s for 500KLOC in PHP-5.6.16). Also, Vanguard has been adopted by industry users and integrated into their testing platform for improving the correctness of products under development. Vanguard has also led us to uncover five known vulnerabilities and two unknown bugs.

The main contributions of this paper are as follows:

- A systematic static approach to detect missing checks was proposed to identify insufficient attack protections and defend potential vulnerabilities in C/C++ code. It is suitable for mature code and programs under development.
- A cross-platform tool, named Vanguard, was implemented on top of Clang/LLVM 3.6.0, which is capable of identifying missing checks in realistic projects automatically. It has been adopted by industry users to help them defend potential vulnerabilities in industry-level projects.
- Experimental evaluation on open-source projects was conducted to demonstrate Vanguard’s effectiveness and efficiency. Furthermore, it ultimately leads us to uncover five known vulnerabilities and two unknown bugs.

The rest of this paper is organized as follows. Section 2 introduces the concept and formal definition of missing check. Section 3 presents an overview and detailed description of our approach. Section 4 introduces the details of implementation and optimization. Section 5 gives the experimental evaluation results. Related works are discussed in Section 6 before we conclude the current work in Section 7.

2 MISSING CHECKS

This section introduces examples of missing checks firstly, and then provides the formal definition.

2.1 Motivation Examples

Missing checks for security-sensitive operations using manipulable data may result in many severe types of vulnerabilities and various disastrous attacks. For example, CVE-2013-0422 is a vulnerability caused by missing check for a sensitive access control function in Java 7, which has been utilized to install malware on millions of hosts by attackers [43]. Recently, “A7-Insufficient Attack Protection” has been proposed as a new type of Top 10 security risks by OWASP [5] in 2017. Thus, missing check, i.e., missing attack protection checks for manipulable data used in security-sensitive operations, is an indicator of insufficient attack protection.

Intuitively, code samples are illustrated in Listing 1 for a better understanding of missing checks. dividend, operand, index and len are untrusted data, which are manipulable by outside attack inputs i and upMsg. They are used in four types of security-sensitive operations (SSO), i.e. division arithmetic, modular operation, array-index access, and security-sensitive function call without protection checks.

### Listing 1: Code samples of missing checks

```c
#define MAX_LEN 100;
char array[MAX_LEN];

void DIV_msg(int i, MSG* msg){
    int quot;
    int dividend=msg->msg_len;
    // if(dividend == 0 ) return;
    quot = (i / dividend);
    /* dividend may be equal to zero */
    printf("quot is: %d
", quot);
}

void MOD_msg(int i, MSG* msg){
    int quot;
    int operand=msg->msg_len;
    // if (operand == 0) return;
    quot = (i % operand);
    /* operand may be equal to zero */
    printf("quot is: %d
", quot);
}

void ARRAY_msg(int i, MSG* msg){
    int index = i + msg->msg_len;
    // if(index >= MAX_LEN || index < 0) return;
    array[index] = msg->msg_value;
    /* index may be out of array bound */
}

void FUNC_msg(MSG* msg){
    char* buf=(char*)malloc(MAX_LEN);
    if(buf == NULL) return;
    int len = msg->msg_len;
    // if (len > MAX_LEN) return;
    memcpy(buf, upMsg->msg_value, len);
    /* len may be larger than MAX_LEN */
}

void EntryFun(int i){
    MSG* upMsg = recvmsg(); // get msg from outside
    DIV_msg(i, upMsg);
    MOD_msg(i, upMsg);
    ARRAY_msg(i, upMsg);
    FUNC_msg(upMsg);
}
```

• Missing Check for Division Arithmetic: The manipulable data \(\text{dividend}\) is used as a dividend in division arithmetic at line 8 without confirming that \(\text{dividend}\) is not equal to zero as commented at line 7, which will result in a divide-by-zero error. It is defined as a “missing divide-zero protection check”.

• Missing Check for Modular Operation: The manipulable data \(\text{operand}\) is used as the second operand in modular operation at line 17 without guaranteeing that \(\text{operand}\) is not equal to zero as commented at line 16, which may lead to a modulus-by-zero error. It is defined as a “missing mod-zero protection check”.

• Missing Check for Array-Index Access: The manipulable data \(\text{index}\) is used as the subscript of an array at line 25 without checking that \(\text{index}\) is in the range of array’s capacity as commented at line 24, which will cause an over-bounds error. It is defined as a “missing array-index-bound protection check”.

• Missing Check for Sensitive Function Call: The manipulable data \(\text{len}\) is used as an argument of a security-sensitive function call (i.e., \(\text{memcpy}\)) at line 34 without comparing \(\text{len}\) and size of \(\text{buf}\) as commented at line 33, which could give rise to a buffer-overflow vulnerability. It is defined as a “missing argument-constraint protection check”.

### 2.2 Formal Definition

#### stmt s ::=
- \(\text{id} \leftarrow \text{expr}\)
- \(\text{call_func}\)
- \(\text{s; s}\)
- \(\text{if} \; \text{expr} \; \text{then} \; \text{s} \; \text{else} \; \text{s}'\)
- \(\text{while} \; \text{expr} \; \text{do} \; \text{s}\)

#### expr e ::=\n- \(\text{id}\)
- \(\text{constant}\)
- \(\text{e}_1 \; \text{\&\&} \; \text{e}_2\)
- \(\text{\&\&} \; \text{e}\)
- \(\text{\|} \; \text{e}_1 \; \text{\&\&} \; \text{e}_2\)
- \(\text{\|} \; \text{e}_1 \; \text{\&\&} \; \text{e}_2\)

#### call_func c ::= \(\text{e} \leftarrow \text{call_func}(\text{id} = e)\)

#### func f ::= \(\text{signature} \leftarrow \text{fname id}\)

#### signature ::= \(\text{fname id}\)

A program consists of a sequence of numbered statements, i.e., assignments, function calls, sequence executions, conditionals, and loops, as defined by \(\text{stmt}\). \(\text{id}\) represents local variables and formal parameter of functions, and \(\text{constant}\) represent constant variables. We use \(\text{\&\&}\) and \(\text{\|}\) to represent typical binary and unary operations, \(\text{\&}\) to represent member operator "." or "\(\rightarrow\)". and \(\text{[]}\) to represent array accesses. This language contains all important features of C/C++. Based on this language grammar, we give the definition of a missing check in Definition 2.1.

**Definition 2.1. (Missing Check):** Let \(TS = (S, Act, \rightarrow, I, AP, L)\) be a transition system for a program, where:

- \(S = \text{Taint}(Var) \times \text{Check}(Var)\) is a set of states, \(\text{Taint}(Var)\) represents whether the variable \(Var\) is tainted or not, and \(\text{Check}(Var)\) represents whether \(Var\) is checked or not.

#### Listing 2: Missing Check Warnings

- \(\text{Act}\) is a set of statements.
- \(\rightarrow \subseteq S \times Act \times S\) is defined by the following rule:

\[
\begin{align*}
\text{if } & \text{expr } \text{then } \text{s} \text{else } \text{s}' \text{, } &\rightarrow t_1 \leftrightarrow t_1, c_i, j_i \quad \text{if } \text{expr } \text{then } \text{s} \text{else } \text{s}' \text{, } &\rightarrow t_1 \leftrightarrow t_1, c_i, j_i \\
\text{while } & \text{expr } \text{do } \text{s} \text{, } &\rightarrow t_1 \leftrightarrow t_1, c_i, j_i \quad \text{while } & \text{expr } \text{do } \text{s} \\
\end{align*}
\]

where \(t_i\) is the act, \(t_1 \leftrightarrow t_i \subseteq \text{Taint}(Var) \times \text{Act} \times \text{Taint}(Var)\), \(\rightarrow \subseteq S \times Act \times S\) is a set of initial states.
- \(AP = \text{Taint}(Var) \cup \text{Check}(Var)\) is a set of atomic propositions.
- \(L = S \rightarrow 2^AP\) is a labeling function.

Let \(\rho = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_i \rightarrow s_{i+1} \rightarrow \ldots\) be an execution path whose action sequence is \(\alpha_1 \alpha_2 \ldots \alpha_{i+1}\). There is a missing check on \(\alpha_i\) if \(\rho\) satisfies the following conditions:

1. For \(\alpha_i \in \text{SSO} \subseteq \text{Act}\), where \(\text{SSO}\) is a set of security-sensitive operations. It represents that \(\alpha_i\) is a security-sensitive operation.
2. For \(\alpha_i\) in condition (1), \(\exists d \in SD(\alpha_i) \land t_{i-1}(d) = T\), where \(SD\) is a function to obtain the data used in \(\alpha_i\). It represents that the sensitive data \(d\) used in \(\alpha_i\) is tainted.
3. For \(d\) in condition (2), \(\bigvee \alpha_i \in \text{Taint}(Var)\) \(\bigvee \text{Check}(Var)\) \(\bigvee \text{Taint}(Var)\) \(\bigvee \text{Check}(Var)\). It represents that there are no attack protection checks for \(d\) and related variables.

### 3 APPROACH

The overview of Vanguard is illustrated in Fig. 1. The inputs include the source code of a target C/C++ program and the configuration file. The output is a warning report of identified missing checks. Vanguard detects missing checks by three steps: (1) security-sensitive operations location, (2) arguments assailability judgment, and (3) insufficient protection assessment.

First, Vanguard locates customized security-sensitive operations (SSO) with lightweight static analysis on the abstract syntax tree, call graph and control flow graph of the target program. Second, sensitive data used in SSO (i.e., dividend in division arithmetic, modulus in modular operation, index of array access, and arguments of security-sensitive function calls) are obtained to judge whether they are assailable by outside attack input, i.e., to decide whether they are tainted using static taint analysis. Third, if a sensitive data is tainted, then a backward data-flow analysis is applied to explore whether there are attack protection checks for tainted data or related variables. If not, then a missing check is identified, and Vanguard extracts its context features, and adds these features’ value to estimate its risk degree. At last, Vanguard generates a warning report for the missing checks in high-risk context.
Example. To illustrate the processes of Vanguard to detect missing checks, we apply Vanguard on the sample code in Listing 1. The function EntryFun at line 38 is an entry function that calls recumsg, DIV_msg, MOD_msg, ARRAY_msg, and FUNC_msg. The recumsg is a library function in charge of receiving messages from outside.

First, security-sensitive operations, i.e., division operator "/" at line 8, modulus arithmetic "/" at line 17, array index array[index] at line 25, and sensitive function call memcp at line 34, are located as well as their arguments dividend, operand, index, buf, and len. Notice that the sensitive data buf and len used as arguments of memcp are obtained according to our configuration item "memcp : 0 + 2", which represents that the first and third arguments of "memcp" need to be checked.

Next, these sensitive data are judged to see whether they are assailable by outside attack input (i.e., tainted or not) using static taint analysis. Our static taint analysis marks the argument i of EntryFun as tainted. upMsg is the return value of the library function recumsg configured in our black-list. Notice that default taints the return value of a library function in black-list. Thus, upMsg is marked as tainted too. Then local variables dividend, operand, index, and len are all tainted, because they are influenced by the taint source i and upMsg through statements at line 6, 15, 23, and 32 based on our taint analysis rules listed in Table 1. Thus, these tainted variables can be manipulable by outside attack input.

After that, Vanguard explores whether there are related protection checks for these tainted data or related variables. Taking the argument len as an example, Vanguard explores proper attack protection checks for len and its related expressions like msg → msg_len before the call site of the sensitive function call memcp. There are no precondition checks for tainted len and related variables in function FUNC_msg and EntryFun. Thus, it is marked as a missing check. Furthermore, context features listed in Table 2 are extracted to compute its risk degree. Notice that the configuration item CheckLevel is set to 1 here. At last, Vanguard will generate detailed information about the missing argument-constraint protection check and report it as a warning in an XML file as Listing 2. Similarity, Vanguard is able to detect other missing checks.

3.1 Security-Sensitive Operations Location
Locating SSOs is the first step to detect missing checks. A lightweight static analysis is performed on the abstract syntax tree of the target program to locate SSOs based on configuration file.

The configuration of SSOs is formally represented as follows, where CheckItem is a configurable item for a security-sensitive operation. It consists of the type, expression list, and argument list of the security-sensitive operation. Type represents types of security-sensitive operations. If type is FUNCTION, then OpList is a list of function names. If the type is OTHERS, then OpList is a list of expressions containing division and modulus operators and array accesses. ArgList is the location of sensitive data that need to be checked in the security-sensitive operation.

\[
\begin{align*}
\text{CheckItem} & :\text{ Type: OpList : ArgList} \\
\text{Type} & : \text{FUNCTION: OTHERS} \\
\text{OpList} & : \text{ExprType}^+ \\
\text{ArgList} & : \text{NUMBER}^+
\end{align*}
\]

For example, sensitive API usage are security-sensitive operations, which are configured as a list of CheckItems with the format as follows:

\[
\begin{align*}
\text{FUNCTION} : \text{f Name : Args}
\end{align*}
\]

FUNCTION represents the type of security-sensitive operation is function. f Name is a list of sensitive functions’ names related to memory operations (e.g., malloc, memset, and memcp) and sensitive API usage (e.g., FTP_StrCpy and FTP_Strncpy), and Args represents location of arguments we need to examine whether they are assailable by outside attack input or not. Note that “0” represents the first argument, “-1” represents all arguments, and we could specify multiple arguments with “+” (e.g, “0+1+2”) if we want to check multiple arguments in the function.

For each function in the target program, a corresponding CFG is constructed based on its AST. Then each statement of every basic block is analyzed by traversing the CFG. If a statement belongs to the type of “call_func”, then we will check whether the callee’s name “calleeName” of the “call_func” is matched with a sensitive function’s name. If calleeName is matched with one f Name, then a security-sensitive function call is located. Furthermore, the sensitive data data used as the actual arguments in the function call is obtained according to Args specified in configuration file.

The location of other SSOs like division arithmetic, modulus operation, and array-index access is similar to the handling of security-sensitive function calls, and we omit the details here.

3.2 Arguments Assailability Judgment
Once a security-sensitive operation and its arguments (i.e., the sensitive data data) are identified, the next step is to analyze the taint status of sensitive data to judge whether they are assailable by outside attack input using static taint analysis.

Our static taint analysis consists of intra-procedural and inter-procedural analysis. First, intra-procedural taint analysis is used to obtain taint relations between local variables and formal parameters of every function. Then, inter-procedural taint analysis is performed to traverse the call graph of the program in an inverse topological order and spread taint status of entry function to related functions’ formal parameters.
Table 1: Taint Analysis Rules

<table>
<thead>
<tr>
<th>Types</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>Γ(e) → τ ∧ Γ(constant) = U</td>
</tr>
<tr>
<td>e1 &amp; e2</td>
<td>Γ(e1) = τ → Γ(e2) = τ2 = Γ(e1 ⊕ e2) = τ1 ⊕ τ2</td>
</tr>
<tr>
<td>e1 ⊕ e2</td>
<td>Γ(e1) = τ ⇒ Γ(e2) = τ</td>
</tr>
<tr>
<td>e1 &amp;m e2</td>
<td>Γ(e1) = τ ⇒ Γ(e1 &amp;m e2) = τ</td>
</tr>
<tr>
<td>e1 → e2</td>
<td>Γ(e1) = τ ⇒ Γ(e2) = τ1 ← e1 → e2 ⇒ Γ(e1) = τ</td>
</tr>
<tr>
<td>e1 ← e2</td>
<td>Γ(e2) = τ, e1 ← e2 → Γ(e1) = τ</td>
</tr>
</tbody>
</table>

Specifically, all the inputs from outside are regarded as taint sources ζ, which is defined formally as below:

ζ = {x | x ∈ ArgsEntry ∪ ApiRet}

where ArgsEntry represents the set of arguments of entry functions and ApiRet represents the return value of external APIs. The default taint status of an API’s return value is configured using a white-list and a black-list by users. Let τ = {T, U} be the taint type domain for our static taint analysis. T and U indicate the tainted and untainted labels respectively.

3.2.1 Intra-procedural Analysis. For each function in target program, we define \( \text{Vars} = \text{LocalV}ars \cup \text{FormalParams} \), LocalVars is a set of local variable expressions in the function, FormalParams is a set of formal parameters of the function. We associate an environment to \( \text{Vars} \) by defining a mapping \( \Gamma \) from \( \text{Vars} \) to taint types in the following way:

\[ \Gamma : \text{Vars} → τ. \]

In order to handle programs that involve presence of expressions, a binary operator \( ⊕ \) : \( τ \times τ \rightarrow τ \) was defined as follows:

\[ x ⊕ y \begin{cases} U & x = U \land y = U \\ T & x = T \lor y = T \end{cases} \]

where \( x \) and \( y \) are expressions of the left and right side of some operations \( op \). The binary operator \( ⊕ \) will be used to compute the taint state of expressions that depend on other variable expressions. For instance, if the taint states of \( \text{expr}1, \text{expr}2 \) are \( τ1, τ2 \), and \( \text{expr}3 = \text{expr}1 + \text{expr}2 \), then the taint state \( τ3 \) for \( \text{expr}3 \) will be computed as \( τ1 \oplus τ2 \).

In order to support inter-procedural taint analysis, an environment for each function is built. It can be reused in different calling contexts. Type variable \( G \) is defined with respect to a function environment \( \Gamma \) as the tuple of variables \( x_1, x_2, ..., x_n \) on which the type variable depends. It denotes \( G(x_1, x_2, ..., x_n) = Γ(x_1) ⊕ Γ(x_2) ⊕ ... ⊕ Γ(x_n) \). Furthermore, we extend the \( \oplus \) operator to \( Γ \) environments:

\[ Γ = Γ_1 \oplus Γ_2 \iff ∀ x ∈ \text{Vars} ⇒ Γ(x) = Γ_1(x) ⊕ Γ_2(x) \]

Algorithm 1: BFSTaintSpread \((CG, f\text{Envs})\)

**Input:** \( CG \): non-recursive Call Graph; \( f\text{Envs} \): taint environment of vars and formal parameters.

**Output:** \( f\text{Envs}'\): updated taint environment

1. foreach \( v \) in \( CG \) do
   2. if color[\( v \)] = WHITE then
      3. \( sCG = \text{CG}.\text{start}(\( v \)) ; \)
      4. color[s\( v \)] = \text{GRAY};
      5. ENQUEUE(Q\( s \));
   while \( Q \neq \text{EMPTY} \) do
      7. \( u = \text{DEQUEUE}(Q) ; \)
      8. foreach \( v \) in callees(\( u \)) do
         9. TaintPropagationThroughCall(\( u, v \));
      10. if color[\( v \)] = \text{WHITE} then
          11. ENQUEUE(Q\( v \));
      12. color[\( u \)] = \text{BLACK};

Let \( \text{Funcs} \) be a set of functions in program. We associate an environment \( Γ \) for each function as follows. We associate type variable \( G(x) \) for each formal parameter \( x \). \( ret \) is created to hold the type of function’s return value. The taint type for return value of the function is a combination of type variables corresponding to the formal parameters and values from \( τ \). A mapping between functions and their associated environment is represented below:

\[ Γ \text{func} : \text{Funcs} → (\text{Vars} → τ) \]

Initially, \( Γ \text{func} \) contains the mappings for library functions. The mappings for user-defined functions will be added when the taint analysis rules list in Table 1 are applied.

Note that \( assigned(\text{stmt}) \) represents the set of left expressions of assignment statements in \( \text{stmt} \), and \( G(id_{i}) ← τ_i \) represents the instantiation of type variable \( G(id_{i}) \) with \( τ_i \).

3.2.2 Inter-procedural Analysis. The original call graph of the program is traversed with a depth-first search algorithm for the sake of obtaining a non-recursive call graph \( CG \) in topological order. Then, \( BFSTaintSpread \) algorithm can be applied on the \( CG \) to perform inter-procedural taint analysis, spreading taint status of entry function to related formal parameters of functions.

As illustrated in Algorithm 1, the inputs are call graph \( CG \) and taint environments \( f\text{Envs} \) storing taint relations between formal parameters and local variables. The outputs are taint environments \( f\text{Envs}' \) storing taint information of formal parameters as well as relations between formal parameters and local variables. Our inter-procedural taint analysis starts at entry function \( s \) of call graph \( CG \) and analyzes the program from top to bottom in breadth-first-search order. The parameters of the entry function are tainted. The call graph is traversed for spreading taint statuses from top entry function’s parameters to their related functions’ formal parameters. For each function, we spread the caller’s actual arguments’ taint statuses to callee’s formal parameters. If multiple functions are calling the same function, then the callee function’s formal parameters’ taint statuses are the combination of its callers’ actual arguments’ taint statuses. In this way, we obtain taint relations between taint sources and formal parameters of each function.
What we need is to collect and provide related information which determines the levels of caller’s ancestors we will explore when locating a security-sensitive operation and its sensitive data which makes it continent to judge taint status of sensitive data with context environment variables in the body of caller and caller’s ancestors. Note that there are proper attack protection checks for taint data or related complex, then the missing check is more likely to be dangerous. CheckLevel will explore proper attack protection checks in the graph of the target program; (2) TaintAnalyzer, which is in charge of establishing taint data pool using static intra-procedural and inter-procedural taint analysis; (3) Detector, which will identify missing checks via lightweight static analysis; and (4) RiskEstimator, which estimates the risk degree of detected missing checks in their contexts by computing context complexity. 

### 4 IMPLEMENTATION

An automated and cross-platform tool called Vanguard was developed based on Clang/LLVM 3.6.0, the architecture is illustrated in Fig. 3. Vanguard consists of four modules: (1) Preprocessor, which is used to obtain abstract syntax tree, control flow graph, and call graph of the target program; (2) TaintAnalyzer, which is in charge of establishing taint data pool using static intra-procedural and inter-procedural taint analysis; (3) Detector, which will identify missing checks via lightweight static analysis; and (4) RiskEstimator, which estimates the risk degree of detected missing checks in their contexts by computing context complexity.

#### Memory Optimization.

In order to avoid the crash while analyzing large-scale projects with Vanguard in a limited memory environment, a cache mechanism for ASTs is used and write is proposed to optimize memory usage. The key idea is to preserve latest used ASTs in memory with an AST queue and users configure the maximal length of AST queue according to practical memory limit.

#### Taint Analysis Optimization.

In order to accelerate the speed of static taint analysis to judge assailability of sensitive data used in security-sensitive operations, a tainted data pool consisting of each variable expression’s taint types is established and stored with the format of 32bit unsigned int type array. It turns taint propagation analysis into bit computation of two-bit arrays of related variable expressions. Meanwhile, a query interface for assessing taint state of a variable is provided. It can be used for identifying a variable taint state conveniently and quickly.

Besides, Vanguard has been adopted by industry users and integrated into their testing platform for improving the correctness of the call graph, then we identify a missing check. More precisely, we will check whether the tainted data is zero for the division and modular operations. For array-index access, we will further check if the tainted data is within the bound of the array.

#### Table 2: Context features

<table>
<thead>
<tr>
<th>No.</th>
<th>Feature Names</th>
<th>Feature Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NumOfArg</td>
<td>Num of arguments</td>
</tr>
<tr>
<td>2</td>
<td>NumOfPlus</td>
<td>Num of “+” in arguments</td>
</tr>
<tr>
<td>3</td>
<td>NumOfMinus</td>
<td>Num of “-” in arguments</td>
</tr>
<tr>
<td>4</td>
<td>NumOfMultiply</td>
<td>Num of “*” in arguments</td>
</tr>
<tr>
<td>5</td>
<td>NumOfDivide</td>
<td>Num of “/” in arguments</td>
</tr>
<tr>
<td>6</td>
<td>NumOfDelively</td>
<td>Num of “%” in arguments</td>
</tr>
<tr>
<td>7</td>
<td>NumOfSimpleVar</td>
<td>Num of simple vars in arguments</td>
</tr>
<tr>
<td>8</td>
<td>NumOfCompositeVar</td>
<td>Num of composite vars in arguments</td>
</tr>
<tr>
<td>9</td>
<td>NumOfSizeof</td>
<td>Num of sizeof Ops in arguments</td>
</tr>
<tr>
<td>10</td>
<td>NumOfCallerVar</td>
<td>Num of variables in caller</td>
</tr>
<tr>
<td>11</td>
<td>NumOfCallerCallExpr</td>
<td>Num of CallExpr in caller</td>
</tr>
<tr>
<td>12</td>
<td>CalleeHasBody</td>
<td>Whether the callee has body</td>
</tr>
<tr>
<td>13</td>
<td>NumOfInBinary</td>
<td>Num of arguments in binary ops</td>
</tr>
<tr>
<td>14</td>
<td>NumOfBinaryOP</td>
<td>Num of binary Ops in caller</td>
</tr>
<tr>
<td>15</td>
<td>NumOfTaintArg</td>
<td>Num of tainted arguments</td>
</tr>
</tbody>
</table>
We collected and specified some typical testing programs\(^1\) to evaluate its effectiveness from three aspects: (1) Accuracy of TaintAnalyzer; (2) False positive of missing check detection; (3) Ability to uncover vulnerabilities caused by missing checks.

5 EVALUATION

Experimental evaluation was conducted on a computer with 64-bit Ubuntu 16.04 LTS system, a processor of Intel(R) Xeon(R) CPU E5-1650 v3 @3.5GHz and 8GB RAM. The evaluation is designed to answer the following three research questions:

- **Q1**: How is the effectiveness of Vanguard?
- **Q2**: How is the efficiency of Vanguard?
- **Q3**: What is the comparison results with other tools?

5.1 Effectiveness of Vanguard (Q1)

We evaluated its effectiveness from three aspects: (1) Accuracy of static taint analysis; (2) False positive of missing check detection; (3) Ability to uncover vulnerabilities caused by missing checks.

5.1.1 Accuracy Analysis of TaintAnalyzer. The effectiveness of missing check detection are relied on the accuracy of TaintAnalyzer. We collected and specified some typical testing programs\(^1\) to validate correctness of TaintAnalyzer. As illustrated in Listing 3, it is a typical example to verify accuracy of taint propagation situation containing pointer, reference and function, which is one of the most difficult situations of taint analysis. We first specified the analyzed code to validate the accuracy of results in comment.

We set tainted() in black list, then return value of tainted() is tainted. Next, we manually analyze testing code from test_pointer. The x is tainted by tainted source at line 23; Member variable m of p1 is assigned by x at line 28. A struct object is regarded as an entirety, if one member is tainted, then whole struct is tainted, so p1 is tainted. Furthermore, a1 and p2 are tainted too since they are pointing the same address; The initial value of variable c is return value of func(x) at line 30, and taint type of func() is Gamma(in), which means the taint type of return value of func() is determined by its actual argument. Its actual argument x is tainted, so is c tainted; Line 31 is a taint propagation situation of function pointer as argument. The taint type of return value of function pointer_param_in is Gamma(pin), similar as func(), is determined by the taint type of its actual argument. The actual argument of pointer_param_in is address of c, c is tainted, so ret1 is tainted; At line 34, b is initialized by number 1, then b is not tainted. b is the actual argument of ref_param_out(). Due to the definition of function ref_param_out(), the reference argument pout will be tainted and return its address, so the actual argument and return value of ref_param_out() are tainted, thus b and ret2 are tainted. By comparing with the results in comments, we can prove our static taint analysis algorithm is correct and accurate.

Based on above analysis, we can know that TaintAnalyzer is able to analysis various C/C++ expressions and taint propagation situations correctly, including propagation of variable definition and assignment (line 8, 13, 18, 23), propagation of return value(line 23, 30, 31, 35), propagation of structure, pointer and reference assignment (line 25-28) and propagation of pointer and reference as function arguments(line 12, 17, 31, 35).

\[\text{int taintd();}\
\text{struct A{}\
\text{int m;}\
\text{};}\
\text{int func(int in){}\
\text{int a = in;}\
\text{return a;} \quad /\!\!/ \text{TaintValue(func)=Gamma(in)*/}\
\text{)}\
\text{int pointer_param_in(int* pin){}\
\text{int x = *pin;}\
\text{return x;} \quad /\!\!/ \text{TaintValue(func) = Gamma(x)*/}\
\text{)}\
\text{int* ref_param_out(int& pout){}\
\text{pout = taintd();}\
\text{return & pout;}\
\text{)}\
\text{int test_pointer(){}\
\text{int x = taintd();} \quad /\!\!/ x= tainted*/}\
\text{struct A a1;}\
\text{struct A* p1 = &a1;}\
\text{struct A* p2 = p1; /\!\!/ a1, p1, p2 = untainted*/}\
\text{p1->m = x;} \quad /\!\!/ a1, p1, p2 = tainted*/\
\text{int c = func(x); /\!\!/ c = tainted*/}\
\text{int ret1 = pointer_param_in(&c);} \quad /\!\!/ \text{ret1= tainted*/}\
\text{int b = 1;} \quad /\!\!/ b = untainted*/\
\text{int* ret2 = ref_param_out(b);} \quad /\!\!/ b = tainted \quad \text{ret2 = tainted*/}\
\text{return 0;}\]
One example is illustrated in Listing 4. The function config_load_with_id is in charge of turning an xml config file into a config hash. The array of path is a reference of result of passing config file. In the loop at line 7, strncpy is a security sensitive memory operation. The loop is trying to copy data from bd.nad->cdata + path[j]->iname to buf. The path is a tainted data affected by outside input xml config. There is a missing check for the total size of path[i]->iname. The size of buf is 1024, a buffer overflow may happen if the total size of path[i]->iname is larger then 1024. We have used dynamic testing to validate the potential vulnerability and construct a test case to trigger the bug. It will make the XMPP protocol server crash.

Based on above observations, we can know that Vanguard is able to detect various missing attack protection checks effectively with low false positive, and its ability to identify missing checks can help to uncover known vulnerabilities. It also can be helpful for identifying potential vulnerabilities for further validating, which narrows the field of unknown vulnerability detection.

5.2 Efficiency of Vanguard (Q2)

We evaluate the efficiency of Vanguard from two aspects: (1) performance of static taint analysis on typical code samples; (2) scalability of missing check detection on open source projects.

5.2.1 Performance of TaintAnalyzer. We selected taint analysis benchmark [1] mentioned in [10] to evaluate the performance of our static taint analysis algorithm. The reason we choose these programs as benchmark is: (1) they are typical programs used by other taint analysis works, and (2) they are implementations of some complex algorithms with various taint propagation situations involving pointer, array, structure and so on. The result is illustrated in Table 6. Where Loc represents the code line of the project. AST is number of AST files, it also equals to the number of source file. Total is total occurrence number of variables. Because the taint environment of each basic block is diffident, and taint types of variables are context-sensitive, Total count the occurrence numbers of all the variables in all the blocks, i.e. Total = \sum_{f} f.NumOfBB * f.NumOfVar. TVar is occurrence number of taint variables. TPer = TVar Total, which represents the dependence degree between program variables and outside input. T(s) is the time of taint analysis, and SM is memory cost.

Based on above observations, we can known that our static taint analysis has good performance in dealing with different scale projects, the time and memory overhead of TaintAnalyzer is low. For instance, it is able to analysis mailx a program with 10K line code in 2.58s with 76.7MB memory cost. It also indicates that it is able to analysis various complex programs with all kinds of C/C++ expressions and structures such as pointer, array, reference and so on.
Table 3: Effectiveness and Efficiency of Vanguard

<table>
<thead>
<tr>
<th>Project</th>
<th>AST</th>
<th>Func</th>
<th>Loc</th>
<th>Time (s)</th>
<th>Sp(M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Php-5.6.16</td>
<td>634</td>
<td>8499</td>
<td>497062</td>
<td>619.93</td>
<td>2793.2</td>
</tr>
<tr>
<td>Openmem-1.1</td>
<td>860</td>
<td>5692</td>
<td>284518</td>
<td>184.21</td>
<td>858.4</td>
</tr>
<tr>
<td>Pidgin-2.10.11</td>
<td>986</td>
<td>328531</td>
<td>375.51</td>
<td>220.3</td>
<td>2793.2</td>
</tr>
<tr>
<td>Littiff-4.0.6</td>
<td>83</td>
<td>697</td>
<td>66855</td>
<td>35.32</td>
<td>152.8</td>
</tr>
<tr>
<td>Littiff-4.0.6</td>
<td>83</td>
<td>697</td>
<td>66855</td>
<td>35.32</td>
<td>152.8</td>
</tr>
<tr>
<td>Littiff-4.0.6</td>
<td>83</td>
<td>697</td>
<td>66855</td>
<td>35.32</td>
<td>152.8</td>
</tr>
</tbody>
</table>

Table 4: Performance of TaintAnalyzer

<table>
<thead>
<tr>
<th>Project</th>
<th>Loc</th>
<th>AST</th>
<th>Total</th>
<th>TVar</th>
<th>TPerc(%)</th>
<th>T(s)</th>
<th>Sp(M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circles</td>
<td>84</td>
<td>1</td>
<td>197</td>
<td>164</td>
<td>83.25</td>
<td>0.95</td>
<td>0</td>
</tr>
<tr>
<td>Queue</td>
<td>227</td>
<td>2</td>
<td>244</td>
<td>79</td>
<td>32.38</td>
<td>0.33</td>
<td>0</td>
</tr>
<tr>
<td>ARR</td>
<td>408</td>
<td>5</td>
<td>626</td>
<td>300</td>
<td>47.92</td>
<td>0.64</td>
<td>0</td>
</tr>
<tr>
<td>Huffman</td>
<td>495</td>
<td>5</td>
<td>809</td>
<td>426</td>
<td>52.66</td>
<td>0.74</td>
<td>20.6</td>
</tr>
<tr>
<td>mails</td>
<td>14609</td>
<td>29</td>
<td>47643</td>
<td>15449</td>
<td>32.43</td>
<td>2.58</td>
<td>76.7</td>
</tr>
</tbody>
</table>

5.2.2 Scalability of Missing Check Detection. As we can see from Table 3, Vanguard finishes analyzing PHP-5.6.16 in 619.93s, which is a project with more than 490 thousand lines of code. Furthermore, we count the time-overhead of Vanguard on project PHP-5.6.16 with increment of AST files, code lines, and functions. All the plots in Figure 4 have shown Vanguard’s complexity of is nearly linear, which is scalable on large size of projects.

In addition, the effect of our memory optimizing in Vanguard is evaluated by analyzing PHP-5.6.16 with setting different size of AST queue. The result in Figure 5 indicates that Vanguard is capable of analyzing PHP-5.6.16 with lower space-cost when size of AST queue is smaller.

Obviously, Vanguard will load ASTs more frequently and cost more time at same time. But when size of AST queue is larger than the number of total ASTs of target project (e.g. 634 for PHP-5.6.16), the space-time cost will stay stable (e.g. 5898MB and 304s) since all ASTs will be loaded into memory at the beginning.

Based on above observations, we can know that Vanguard is capable of dealing with different large-scale projects with low space-time cost, and its complexity is nearly linear. Meanwhile, our memory optimizing technique is effective. It allows Vanguard to be used in different environments with limited memory resources adaptively.

5.3 Comparison with Other Tools (Q3)

Existing work to detect missing checks are mainly Chucky [43] and RoleCast [34] as far as we know. We compare Vanguard with Chucky and RoleCast from three aspects: (1) Kinds of programming languages; (2) Types of missing checks; (3) Average false positive.

Table 5: Vanguard, Chucky and RoleCast

<table>
<thead>
<tr>
<th>Missing Check Type</th>
<th>C</th>
<th>C++</th>
<th>PHP</th>
<th>JSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>missing divide-zero check</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>missing mod-zero check</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>missing array-index-bound check</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>missing sensitive-APIs usage check</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>missing security logic check</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>missing sql injection check</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

As we can see from Table 5, Vanguard and Chucky are able to handle C/C++ languages while RoleCast focus on PHP and JSP. All three tools are capable of detecting missing checks for sensitive APIs usage, meanwhile Vanguard can detect missing check for divide-zero, mod-zero and array-index-bound. Chucky and RoleCast can detect missing checks for security logic. Furthermore RoleCast can handle missing checks for sql-injection. In terms of false positive of detection, three tools have approximative accuracy.

6 RELATED WORK

6.1 Taint Analysis

Taint analysis [18] [10] attempts to identify variables that have been tainted with user controllable input. Static taint analysis [27] [23] can achieve higher code coverage without runtime overhead compared with dynamic taint analysis [30] [13]. Meanwhile the disadvantage is that it will loss a certain degree of accuracy for lack of dynamic information. Dytan [13] is a general framework for dynamic taint analysis. Pxy [17] applies static taint analysis to detect SQL injection, cross-site scripting or command injection bugs in PHP scripts. Safer [11] is a tool combining taint analysis with control dependency analysis to detect control structures that can be triggered by untrusted user input. Inspired by [13], we design and implement an extensible static taint analysis including intra-procedural and inter-procedural analysis with features of controllable taint sources and taint propagation rules. It is used to judge whether sensitive data used by security-sensitive operators is assailable by attack input or not.

6.2 Missing Check Detection

Chucky [43] is a missing check detection tool using intra-procedural static taint analysis and machine learning. It identifies missing checks for security logic and APIs usage based on assumption that missing checks are rare events. Therefore, it is more suitable for analyzing mature code due to the assumption are usually not valid in early development stage. Different from Chucky’s detection for missing check using machine learning, Vanguard identifies missing checks by pure static analysis including intra-procedural and inter-procedural taint analysis. Vanguard is able to identify missing checks for more types of security sensitive operations including division arithmetic, modulus operation, array-index access. Our tool is aimed to improve code’s correctness, which can be used on mature code and programs at development stage.

RoleCast [34] is a static analysis tool to identify security-related events such as database writes in web applications, using a consistent web application pattern without specification. Then, it exploits common software engineering patterns and a role specific variable consistency analysis algorithm to detect missing authorization...
checks. This approach is tightly bounded to web applications written in PHP and JSP, while Vanguard can be applied to common software systems written in C/C++ language.

7 CONCLUSIONS

Vanguard, an automatic static detection system for missing checks in C/C++ programs is designed and implemented on top of Clang/L-LVM 3.6.0, which is aimed at improving correctness of software code by identifying insufficient attack protections. It is able to identify missing checks by: (1) locating customized security-sensitive operations with lightweight static analysis; (2) judging assailability of sensitive data used in security-sensitive operations via static taint analysis; (3) assessing existence and risk degree of missing checks using static analysis and complexity computation. Experimental results on open source projects have shown Vanguard’s effectiveness and efficiency. Furthermore, Vanguard has been adopted by industry users. And it’s ability to identify missing checks has led us to uncover five known vulnerabilities and two unknown bugs.

8 ACKNOWLEDGEMENT

The paper was partially supported by the National Key Research and Development Plan (No. 2016YFB1000802), the National Natural Science Foundation of China (No. 61472179, 61561146394, 61572249), the Doctoral Creative Innovation Research project of Nanjing University (2016014).

REFERENCES

[29] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In Security and privacy (SP), 2010 IEEE symposium on. IEEE, 317–331.