Situ LY, Wang LZ, Liu Y *et al.* Automatic detection and repair recommendation for missing checks. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 34(5): 972–992 Sept. 2019. DOI 10.1007/s11390-019-1955-3

Automatic Detection and Repair Recommendation for Missing Checks

Ling-Yun Situ[1,2], *Student Member*, *CCF*, Lin-Zhang Wang[1,2,*], *Distinguished Member*, *CCF*
Yang Liu[3], *Member*, *ACM*, *IEEE*, Bing Mao[1,2], and Xuan-Dong Li[1,2], *Fellow*, *CCF*

[1] *State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China*
[2] *Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China*
[3] *School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798, Singapore*

E-mail: situlingyun@seg.nju.edu.cn; lzwang@nju.edu.cn; yangliu@ntu.edu.sg; {maobing, lxd}@nju.edu.cn

Received February 26, 2019; revised July 22, 2019.

Abstract Missing checks for untrusted inputs used in security-sensitive operations is one of the major causes of various vulnerabilities. Efficiently detecting and repairing missing checks are essential for prognosticating potential vulnerabilities and improving code reliability. We propose a systematic static analysis approach to detect missing checks for manipulable data used in security-sensitive operations of C/C++ programs and recommend repair references. First, customized security-sensitive operations are located by lightweight static analysis. Then, the assailability of sensitive data used in security-sensitive operations is determined via taint analysis. And, the existence and the risk degree of missing checks are assessed. Finally, the repair references for high-risk missing checks are recommended. We implemented the approach into an automated and cross-platform tool named Vanguard based on Clang/LLVM 3.6.0. Large-scale experimental evaluation on open-source projects has shown its effectiveness and efficiency. Furthermore, Vanguard has helped us uncover five known vulnerabilities and 12 new bugs.

Keywords static analysis, missing check, vulnerability detection, repair recommendation

1 Introduction

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

To improve the correctness of program, vulnerability detection plays one of the most important roles. Unfortunately, an automatic approach to precisely detecting arbitrary types of vulnerabilities does not exist according to Rice's theorem[2]. Thus, the state-of-the-art security research has focused on digging specific vulnerabilities buried in code such as buffer overflow[3,4], integer overflow[5,6], use-after-free[7,8], memory leakage[9,10], null pointer dereference[11,12], and out-of-bound errors[13,14] by static and dynamic approaches including static analysis[15−17], taint analysis[18], symbolic execution[19,20], concolic execution[21,22], model checking[23,24], and fuzzing[25,26], etc.

However, it is a fact that missing checks for manipulable data used in security-sensitive operations is one of the major causes of various severe vulnerabilities. Therefore, efficiently identifying and repairing missing checks in realistic software are essential for the prog-

Regular Paper
Special Section on Software Systems 2019
A preliminary version of the paper was published in the Proceedings of Internetware 2018.
This work was supported by the National Key Research and Development Program of China under Grant No. 2017YFA0700604, and the National Natural Science Foundation of China under Grant Nos. 61632015 and 61690204, and partially supported by the Collaborative Innovation Center of Novel Software Technology and Industrialization, and Nanjing University Innovation and Creative Program for Ph.D. Candidate under Grant No. 2016014.
*Corresponding Author

noses of potential vulnerabilities and the improvement of code security, especially in the early development stage.

Several approaches have been proposed to detect missing checks. Chucky[27] detects missing checks by machine learning (i.e., 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. But the feedback from industry developers indicates that missing checks occur a lot during a development stage. Therefore, it is more suitable for analyzing stable code since the assumption is not valid in the early development stage. RoleCast[28] 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 prognosticate potential vulnerabilities and improve code correctness, we propose a systematic static approach to detect missing checks for manipulable data used in security-sensitive operations in C/C++ programs. It could be used for stable code as well as programs under development. First, customized security-sensitive operations (e.g., sensitive API usage, array-index access, division and modular arithmetic) are located with lightweight static analysis based on the abstract syntax tree (AST)[29], call graph (CG)[30] and control flow graph (CFG)[31] of the target program. Second, sensitive data used in these security-sensitive operations is checked to see whether it is manipulable by outside inputs via inter-procedural static taint analysis. Third, a data flow based backward analysis algorithm is applied to explore protection checks from the locations of security-sensitive operations: if no proper protection checks exist, then a missing check is identified. Further, the risk degree of detected missing checks is assessed based on its context metrics. At the same time, the corresponding repair references for the detected high-risk missing checks are recommended. At last, details of the high-risk missing check and recommended repair references are reported.

We have developed an automated and cross-platform tool named Vanguard on the top of Clang/LLVM 3.6.0, and conducted large-scale experiments on open-source projects to demonstrate its effectiveness and efficiency. The results indicate that Vanguard is able to detect missing checks in open-source projects with low false positive (i.e., 13.23% on average) and low time overhead (e.g., 619 s for 500k lines of code in Php-5.6.16).

We summarize the main contributions of this paper as follows.

• A purely static detection and repair recommendation approach for missing checks was proposed to prognosticate potential vulnerabilities and improve the code correctness of C/C++ programs.

• A cross-platform tool named Vanguard was implemented on top of Clang/LLVM 3.6.0. It is capable of identifying high-risk missing checks in realistic projects and recommending useful repair references.

• Large-scale experimental evaluation on open-source projects was performed to demonstrate the effectiveness and efficiency of Vanguard. Furthermore, it ultimately leads us to uncover five known vulnerabilities and 12 new bugs.

The rest of the paper is organized as follows. Section 2 introduces the motivation example and formal definition of the missing check. Section 3 presents a detailed description of the proposed approach. Section 4 introduces the details of implementation and optimization mechanism. Section 5 gives the experimental evaluation results. Section 6 presents and discusses related work. Finally, we conclude the work in Section 7.

2 Missing Checks

The section illustrates an example of missing checks for four kinds of security-sensitive operations and gives the formal definition of the missing check.

2.1 Motivation Examples

Missing checks for security-sensitive operations using manipulable data may result in many severe types of vulnerabilities and a lot of disastrous attacks. For example, CVE-2013-0422 is a vulnerability caused by the missing check for a sensitive access-control function in Java 7, which has been utilized to install malware on millions of hosts by attackers[27]. Recently, "A7-Insufficient Attack Protection" has been proposed as a new type of top 10 security risks by OWASP in 2017①. Missing protection checks for manipulable data used in

①OWASP2017. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, July 2019.

security-sensitive operations (i.e., missing checks) is an indicator of insufficient attack protection.

Intuitively, a code snippet is illustrated in Fig.1 for better understanding of missing checks for different security-sensitive operations. The *dividend*, *operand*, *index* and *len* are untrusted data. They could be manipulated by outside attack inputs i and $upMsg$ when used in four types of security-sensitive operations, e.g., division arithmetic, modular operation, array-index access, and security-sensitive function call without proper protection checks.

```
1  #define MAX_LEN 100;
2  char  array[MAX_LEN];
3
4  void DIV_msg(int i, MSG* msg){
5    int quot;
6    int dividend=msg->msg_len;
7    // if(dividend == 0 ) return;
8    quot =  (i / dividend);
9    /* dividend may be equal to zero*/
10   printf("quot is: %d\n", quot);
11 }
12
13 void MOD_msg(int i, MSG* msg){
14   int quot;
15   int operand=msg->msg_len;
16   // if (operand == 0) return;
17   quot =  (i % operand);
18   /* operand may be equal to zero */
19   printf("quot is: %d\n", quot);
20 }
21
22 void ARRAY_msg(int i, MSG* msg){
23   int index = i + msg->msg_len;
24   // if(index>=MAX_LEN || index<0) return;
25   array[index] = msg->msg_value;
26   /* index may be out of array bound */
27 }
28
29 void FUNC_msg(MSG* msg){
30   char* buf=(char*)malloc(MAX_LEN);
31   if(buf == NULL) return;
32   int len = msg->msg_len;
33   // if (len > MAX_LEN) return;
34   memcpy(buf, upMsg->msg_value, len);
35   /* len may be larger than MAX_LEN */
36 }
37
38 void FUNC2_msg(MSG* msg){
39   char* buf=(char*)malloc(MAX_LEN);
40   if(buf == NULL) return;
41   int len = msg->msg_len;
42   if (len <= MAX_LEN)
43   memcpy(buf, upMsg->msg_value, len);
44 }
45
46 void EntryFun(int i){
47   //get msg from outside
48   MSG* upMsg =recvmsg();
49   DIV_msg(i, upMsg);
50   MOD_msg(i, upMsg);
51   ARRAY_msg(i, upMsg);
52   FUNC_msg(upMsg);
53   FUNC2_msg(MSG* msg);
54 }
```

Fig.1. Code sample of missing checks.

• *Missing Check for Division Arithmetic.* The manipulable data *dividend* is used as a dividend in division arithmetic at line 8 without confirming that *dividend* is not equal to zero as annotated at line 7, which will result in a divide-by-zero error.

• *Missing Check for Modular Operation.* The manipulable data *operand* is used as the second operand in modular operation at line 17 without guaranteeing that *operand* is not equal to zero as annotated at line 16, which may lead to a modulus-by-zero error.

• *Missing Check for Array-Index Access.* The manipulable data *index* is used as the subscript of an array at line 25 without checking that *index* is in the range of array's capacity as annotated at line 24, which will cause an out-of-bounds error.

• *Missing Check for Sensitive API Usage.* The manipulable data *len* is used as an argument of a security-sensitive function call (i.e., *memcpy*) at line 34 without comparing *len* as annotated at line 33, which could give rise to a buffer-overflow vulnerability.

2.2 Formal Definition

As illustrated in Fig.2, a program consists of a sequence of numbered statements, i.e., assignments, function calls, sequence executions, conditionals, and loops as defined by *stmt*. *id* represents a local variable and formal parameter of functions, and *constant* represents a constant variable. We use $\diamondsuit_b$ and $\diamondsuit_u$ to represent typical binary and unary operations respectively. $\diamondsuit_m$ represents member operator "." or "$\rightarrow$", and [] represents array accesses. It is an abstract program containing all important features of C/C++. A transition system for the program could be defined as Definition 1.

$$
\begin{array}{rcl}
\textbf{\textit{stmt}}\ s & ::= & \textbf{\textit{id}} \leftarrow \textbf{\textit{expr}} \\
 & & \mid \textbf{\textit{call\_func}} \\
 & & \mid s;\ s' \\
 & & \mid \textit{if}\ \textbf{\textit{expr}}\ \textit{then}\ s\ \text{else}\ s' \\
 & & \mid \textit{while}\ \textbf{\textit{expr}}\ \textit{do}\ s \\
\textbf{\textit{expr}}\ e & ::= & \textbf{\textit{id}} \\
 & & \mid \textbf{\textit{constant}} \\
 & & \mid e_1 \diamondsuit_b e_2 \\
 & & \mid \diamondsuit_u e \\
 & & \mid e_1 \diamondsuit_m e_2 \\
 & & \mid e_1\ [e_2] \\
\textbf{\textit{call\_func}}\ c & ::= & e \leftarrow \textit{call}\ \textbf{\textit{func}}\ (id' = e)^* \\
\textbf{\textit{func}}\ f & ::= & \textbf{\textit{signature func\_body}} \\
\textbf{\textit{signature}} & ::= & \textbf{\textit{fname id}}^* \\
\textbf{\textit{func\_body}} & ::= & \textbf{\textit{stmt}}^*
\end{array}
$$

Fig.2. Program composition.

Definition 1 (Transition System). *Let* $TS = (S, Act, \rightarrow, I, AP, L)$ *be a transition system for a program, where*:

• $S = Taint(Var) \times Check(Var)$ *is a set of states,* $Taint(Var)$ *represents whether the variable* Var *is tainted or not, and* $Check(Var)$ *represents whether* Var *is checked or not,*

• Act *is a set of statements,*

• $\rightarrow \subseteq S \times Act \times S$ *is defined by the following rule*:

$$\frac{t_{i-1} \overset{\alpha_i}{\hookrightarrow_t} t_i, c_{i-1} \overset{\alpha_i}{\hookrightarrow_c} c_i}{< t_{i-1}, c_{i-1} > \overset{\alpha_i}{\longrightarrow} < t_i, c_i >},$$

where α_i *is the act,* $\hookrightarrow_t \subseteq Tnt(Var) \times Act \times Taint(Var)$, $\hookrightarrow_c \subseteq Check(Var) \times Act \times Check(Var)$,

• $I \subseteq S$ *is a set of initial states,*

• $AP = Taint(Var) \cup Check(Var)$ *is a set of atomic propositions, and*

• $L = S \rightarrow 2^{AP}$ *is a labeling function.*

Furthermore, we give the formal definition of missing checks based on the definition of transition system as follows.

Definition 2 (Missing Check). *Let* $\rho = < s_0 \overset{\alpha_1}{\longrightarrow} s_1 \overset{\alpha_2}{\longrightarrow} s_2 ... s_{i-1} \overset{\alpha_i}{\longrightarrow} s_i ... >$ *be an execution path whose action sequence is "*$\alpha_1\alpha_2...\alpha_i...$*". There is a missing check on* ρ *iff* ρ *satisfies the following conditions.*

1) $\exists\ \alpha_i \in SSO \subseteq Act$, *where* SSO *is a set of security-sensitive operations. It represents that* α_i *is a security-sensitive operation.*

2) *For* α_i *in condition* 1, $\exists\ d \in SD(\alpha_i) \wedge t_{i-1}(d) = T$, *where* SD *is a function to obtain the data used in* α_i. *It represents that the sensitive data* d *used in* α_i *is tainted.*

3) *For* d *in condition* 2, $c_1(d) \vee c_2(d) \vee ... \vee c_i(d) = F$. *It represents that there are no attack protection checks for* d *and related variables.*

3 Approach

The overview of Vanguard is illustrated in Fig.3. The inputs are the source code of a C/C++ program and the configuration file. The outputs are identified missing check warnings and recommended repair references. Vanguard detects missing checks by the four steps: 1) security-sensitive operations location, 2) arguments assailability judgment, 3) insufficient protection assessment, and 4) repair protection recommendation. Note that because the detection techniques for four types of missing checks are similar, the description of the approach is based on the example of detecting missing checks for sensitive APIs usage in the following.

Vanguard first 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, the modulus in modular operation, index of array access, and arguments of security-sensitive function calls) is obtained to determine whether it is assailable by outside attack input. It is achieved by checking whether it is tainted or not by static taint analysis. Third, if sensitive data is tainted, then a backward data-flow analysis is applied to explore whether there are protection checks for the tainted data or related variables. If not, then a missing check is identified. Vanguard will extract its context metrics, and estimate its risk degree based on the sum of these metrics values. Finally, Vanguard generates warnings and repair references for the detected high-risk missing checks.

To illustrate the processes of Vanguard to detect missing checks, we apply Vanguard on the code snippet in Fig.1. The function *EntryFun* at line 46 is an entry function that calls *recvmsg*, *DIV_msg*, *MOD_msg*,

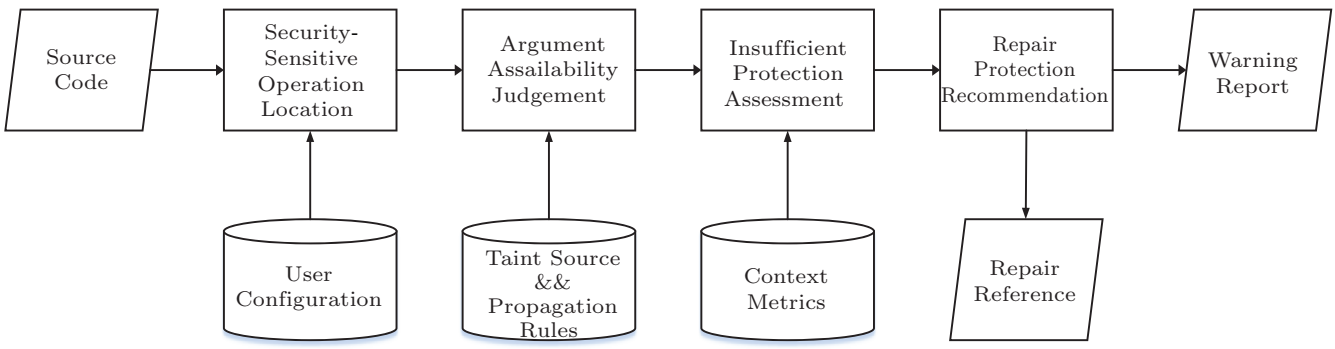


Fig.3. Overview of Vanguard.

ARRAY_msg, and *FUNC_msg*. *recvmsg* is a library function in charge of receiving messages from the outside.

First, security-sensitive operations, i.e., division operator at line 8, modulus arithmetic at line 17, array index $array[index]$ at line 25, sensitive function call *memcpy* at lines 34 and 43, are located as well as their arguments (i.e., *dividend*, *operand*, *index*, buf, and *len*). Notice that sensitive data (i.e., the buf and *len* used as arguments of *memcpy*) is obtained according to our configuration item "*memcpy* : 0 + 2", which represents that the first and the third arguments of "*memcpy*" need to be checked.

Second, the sensitive data is judged to see whether it is assailable by outside attack input using static taint analysis. Our static taint analysis regards the argument i of *EntryFun* as tainted. $upMsg$ is the return value of the library function *recvmsg* configured in our black-list. It is tainted since the return value of a library function in the black-list is tainted by default. Thus, $upMsg$ is marked as tainted too. Local variables *dividend*, *operand*, *index*, and *len* are all tainted, because they are influenced by the taint source i and $upMsg$ through statements at lines 6, 15, 23, and 32 based on our taint analysis rules listed in Table 1. Therefore, these tainted variables can be manipulable by outside attack input.

Third, Vanguard explores whether there are protection checks for the tainted data or related variables. Taking the argument *len* as an example, Vanguard explores proper protection checks for *len* and its related expression (i.e., $msg \rightarrow msg\_len$) before the call site of the sensitive API *memcpy*. There are no protection checks for tainted *len* and related variables in functions *FUNC_msg* and *EntryFun*. Thus, it is marked as a missing check. Furthermore, the context metrics listed in Table 2 are extracted to calculate its risk degree. Note that the configuration item *CheckLevel* is set to 1 here.

At last, Vanguard will generate detailed information about the missing check as a warning as shown in Fig.4. Similarity, Vanguard is able to detect other types of missing checks.

Meanwhile, repair references for the missing check are generated based on existing protection checks (e.g., line 42 in $FUNC2\_msg$) for the same function in the project. They are exported into an XML as Fig.5.

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 the configuration file. The configuration of SSOs is formally represented as Fig.6. *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 the type is $FUNCTION$, then *OpList* is a list of function names. If the type is $OTHERS$, then *OpList* is a list of expressions including division and modulus operand and the array index. *ArgList* is the location of sensitive data that needs to be checked in the security-sensitive operation.

For example, if the security-sensitive operation is sensitive API usage, then its configuration is a list of

Table 1. Taint Propagation Rules

Type	Rule
stmt s	$\Gamma \xrightarrow{s} \Gamma'$
expre	$\Gamma(e) \rightarrow \tau \wedge \Gamma(constant) = U$
$e_1 \diamondsuit_b e_2$	$\Gamma(e_1) = \tau_1, \Gamma(e_2) = \tau_2 \Rightarrow \Gamma(e_1 \diamondsuit_b e_2) = \tau_1 \oplus \tau_2$
$\diamondsuit_u e$	$\Gamma(e) = \tau \Rightarrow \Gamma(\diamondsuit_u e) = \tau$
$e_1 \diamondsuit_m e_2$	$\Gamma(e_1) = \tau \Rightarrow \Gamma(e_1 \diamondsuit_m e_2) = \tau$
$e_1[e_2]$	$\Gamma(e_1) = \tau \Rightarrow \Gamma(e_1[e_2]) = \tau$
$e_1 \leftarrow e_2$	$\Gamma(e_2) = \tau, e_1 \leftarrow e_2 \Rightarrow \Gamma(e_1) = \tau$
$\&e_1 \leftarrow e_2$	$\Gamma(e_2) = \tau, \&e_1 \leftarrow e_2 \Rightarrow \Gamma(e_1) = \tau$
$s; s'$	$\Gamma \xrightarrow{s} \Gamma_1, \Gamma_1 \xrightarrow{s'} \Gamma_2 \Rightarrow \Gamma \xrightarrow{s;s'} \Gamma_2$
if	$\forall e' \in assigned(s) \cup assigned(s'), \quad \Gamma_3(e') = \Gamma(e) \oplus \Gamma_1(e') \oplus \Gamma_2(e')$
while	$i = 0, Do \quad \forall e' \in assigned(s), \Gamma_i(e') = \Gamma(expr) \cup \Gamma_i(e'); \quad i++; Until \quad \Gamma_i == \Gamma_{i-1}$
call_func	$\Gamma(e_1) = \tau_1, ..., \Gamma(e_n) = \tau_n, \Gamma_{g(id_1 \leftarrow e_1, ..., id_n \leftarrow e_n)} = \tau, \quad \Gamma \xrightarrow{expr \leftarrow call\ g} \Rightarrow \Gamma'_{[expr:\tau \mid G(id_1 \leftarrow \tau_1, ..., id_n \leftarrow \tau_n)]}$

Table 2. Context Metrics

No.	Metric	Semantics
01	Num_Arithmetic_Op	Total number of basic arithmetic operations such as "+, –, *, %, ++, – –, etc." in the function
02	Num_Shift_Op	Total number of shift operations such as $\gg$ and $\ll$ in the function
03	Num_Bit_Op	Total number of bit operations such as "&" and "|" in the function
04	Num_Pointer_Var	Total number of pointer type variables in the function
05	Num_Array_Var	Total number of array type variables in the function
06	Num_UserDefine_Var	Total number of user-defined type variables in the function
07	Num_BasicType_Var	Total number of basic type variables such as int, char, string, float and double in the function
08	Num_Local_Var	Total number of local variables in the function
09	Num_Global_Var	Total number of global variables in the function
10	Num_Para_Var	Total number of variables acting as function parameters
11	Num_Para_Expr	Total number of expressions acting as function parameters
12	Num_Taint_Para	Total number of tainted variables and expressions in in function parameters
13	Num_BasicArith_ParaExpr	Total number of basic arithmetic operations acting as function parameter expressions
14	Num_ShiftOp_ParaExpr	Total number of shift operations appeared in function parameter expressions
15	Num_BitOp_ParaExpr	Total number of bit operations appeared in function parameter expressions
16	Num_Sizeof_ParaExpr	Total number of sizeof functions appeared in parameter expresssions
17	Num_Loop	Total number of loops in the function
18	Num_NestedLoop	Total number of nested loops in the function
19	Num_BasicVar_InLoop	Total number of basic variables appeared in the loop
20	Num_ArrayVar_InLoop	Total number of array type variables appeared in the loop
21	Num_PointerVar_InLoop	Total number of pointer type variables appeared in the loop
22	Num_UserDefineVar_InLoop	Total number of user-defined type variables appeared in the loop
23	Num_PointerArith_InLoop	Total number of pointer arithmetic operations in the loop
24	Num_PointerArith_Function	Total number of pointer arithmetic operations in the function
25	Num_Return_Var	Total number of variables involved in the return statement of the function
26	Num_FunctionCallExpr	Total number of function call expressions in the function
27	Num_LibraryCallExpr	Total number of library API call expressions in the funciton
28	Num_Sensitive_CallExpr	Total number of security sensitive function and library call expressions in the funciton
29	Cyclomatic complexity	Cyclomatic complexity of the function
30	Num_Instrction	Total number of instructions in the function

CheckItems with the format as follows.

$$FUNCTION : fName : Args$$

```
<error>
</Event>
<file>C:/src/tainted_mem.c</file>
<Callerfunction>FUNC_msg</Callerfunction>
<SensitiveOp>memcpy</SensitiveOp>
<Description>
[memcpy] is a sensitive operation using
    tainted data:[len]
Location : [C:/src/tainted_mem.c:34:15.]
Call Stack: EntryFun; FUNC_msg; memcpy;
</Description>
<riskDegree>75<riskDegree>
<line>35</line>
</Event>
</error>
```

Fig.4. Missing check warning.

```
<!-- REPAIR.REFERENCE.XML -->
<ReferableRepair>
<SensitiveOp>memcpy</SensitiveOp>
<ifCheck> len <= MAX_LEN</ifCheck>
</ReferableRepair>
```

Fig.5. Repair references for sensitive API usage.

$$\begin{array}{rcl} CheckItem & ::= & Type: OpList : ArgList \\ Type & ::= & FUNCTION: OTHERS \\ OpList & ::= & ExprType^{+} \\ ArgList & ::= & NUMBER^{+} \end{array}$$

Fig.6. Configuration files.

FUNCTION represents that the type of security-sensitive operation is a function call. $fName$ is a list of sensitive functions' names including memory-related functions (e.g., *malloc*, *memset*, and *memcpy*), string-related functions (e.g., strcpy, and strncpy) and self-defined sensitive functions in the projects. *Args* represents the location of arguments. These arguments need to be examined whether they are assailable by outside attack input. Note that "0" represents the first argument, "–1" represents all arguments. We could specify multiple arguments with "+" such as "0 + 1" when we want to check the first and the second 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 the type of a statement is "$call\_func$", then we will check whether its callee's name "$CalleeName$" is matched with a sensitive function's name. If the $CalleeName$ is matched with one $fName$, then a security-sensitive API usage is located. Furthermore, the sensitive data used as the actual arguments in the function call is obtained according to the $Args$ specified in the configuration file.

The process of other SSOs like division arithmetic, modulus operation, and array-index access is similar to the handling of security-sensitive API usage.

3.2 Arguments Assailability Judgment

Once a security-sensitive operation and its arguments (i.e., sensitive data) are identified, the next step is to judge the assailability of the sensitive data. How to effectively identify if the sensitive data used in the security-sensitive operation is exploitable by outside input is one of main technical challenges. We overcome the challenge by 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 inverse topological order and spread the taint status of entry function to related functions' formal parameters.

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

$$\varsigma = \{x | x \in ArgsEntry \cup ApiRet\},$$

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 APIs' return value is configured using a user-defined white-list and a black-list.

3.2.1 Intra-Procedural Analysis

Let $\tau = \{T, U\}$ be the taint type domain for our static taint analysis. T and U indicate the *tainted* and *untainted* labels respectively. We define $Vars = LocalVars \cup FormalParams$ for each function of the target program. $LocalVars$ is a set of local variable expressions in the function and $FormalParams$ is a set of formal parameters of the function. We associate an environment to $Vars$ by defining a mapping Γ from $Vars$ to taint types in the following way.

$$\Gamma : Vars \to \tau.$$

In order to handle programs that involve the presence of expressions, a binary operator $\oplus : \tau \times \tau \to \tau$ was defined as follows.

$$x \oplus y \begin{cases} U, & \text{if } x = U \wedge y = U, \\ T, & \text{if } x = T \vee y = T, \end{cases}$$

where x and y are expressions of the left and the right side of some operations. U indicates the taint status is false and T represents the taint status is true. The binary operator $\oplus$ will be used to compute the taint state of expressions that depend on other variable expressions. For instance, if the taint states of $expr1$, $expr2$ are $t1$, $t2$ respectively, and $epxr3 = expr1 + expr2$, then the taint state $t3$ for $expr3$ will be computed as $t1 \oplus t2$.

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 Γ as the tuple of variables $(x_1, x_2, ..., x_n)$ on which the type variable depends. It denotes $G(x_1, x_2, ..., x_n) = \Gamma(x_1) \oplus \Gamma(x_2) \oplus ... \oplus \Gamma(x_n)$. Furthermore, we extend the $\oplus$ operator to Γ environments.

$$\Gamma = \Gamma_1 \oplus \Gamma_2 \Longleftrightarrow \forall x \in Vars, \Gamma(x) = \Gamma_1(x) \oplus \Gamma_2(x).$$

Let $Funcs$ be a set of functions in the 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 the 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 environments is represented below:

$$\Gamma_{func} : Funcs \to (Vars \to \tau).$$

Initially, Γ_{func} contains the mappings for library functions. The mappings for user-defined functions will be added when the taint analysis rules listed in Table 1 are applied. Note that $assigned(stmt)$ represents a set of the left expressions of assignment statement and $G(id_i) \leftarrow \tau_i$ represents the instantiating 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, Algorithm 1 is applied on the call graph

to perform inter-procedural taint analysis, and spread taint status of entry function to related formal parameters of functions.

Algorithm 1. BFSTaintSpread (CG, $fEnvs$)

```
   Input: CG: non-recursive call graph;
   fEnvs: function taint environment
   Output: fEnvs': updated function taint environment
 1 foreach v ∈ CG do
 2 |  color[v] = WHITE;
 3 end
 4 s= CG.start();
 5 color[s]= GRAY;
 6 ENQUEUE(Q, s);
 7 while Q != EMPTY do
 8 |  u=DEQUEUE(Q);
 9 |  Callees = u.callees;
10 |  foreach v ∈ Callees do
11 |  |  TaintPropagationThroughCall(u, v);
12 |  end
13 |  if color[v] == WHITE then
14 |  |  ENQUEUE(Q,v);
15 |  end
16 |  color[u]= BLACK;
17 end
```

As illustrated in Algorithm 1, the inputs are the call graph CG and taint environments $fEnvs$ storing taint relations between formal parameters and local variables. The outputs are taint environments $fEnvs'$ storing the taint information of formal parameters and 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 the top entry function's parameters to their related functions' formal parameters. For each function, we spread the caller's actual arguments' taint statuses to the callee's formal parameters. If multiple functions are calling the same function, then the callee'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.

3.2.3 Establishing Tainted Data Pool

Furthermore, we establish a taint data pool Θ based on the results from intra-procedural and inter-procedural analysis. It can be represented as a mapping from expression $e \in Exprs$ associated with context environment $\xi(e) = (FunctionDecls, Blocks, Stmts)$ to its taint status.

$$\Theta : (FunctionDecls, Blocks, Stmts, Expres) \rightarrow \tau,$$

which makes it convenient to judge taint status of sensitive *data*. What we need is to collect and provide related information $\xi(data)$ when locating a security-sensitive operation and its sensitive arguments. The information includes function declaration, block, statement and argument expression, which are represented as $FunctionDecl, CFGBlock, Stmt$ and $Expr$ in Θ respectively.

3.3 Insufficient Protection Assessment

If a security-sensitive operation using one or more taint data is identified, then it is possible to be exploited by the outside attack input. We perform the insufficient protection assessment by 1) exploring the existence of related protection checks and 2) further estimating the risk degree of detected missing checks based on its context metrics.

3.3.1 Exploring Protection Checks

A backward data-flow analysis is performed to explore whether there are proper protection checks for taint data or related variables in the body of the caller and the caller's ancestors. Note that there is a configuration item with the format as follows.

$$CheckLevel : N,$$

which determines the levels of caller's ancestors we will explore along one path of the call graph. When $CheckLevel$ is equal to zero, Vanguard will explore proper protection checks in the body of the caller invoking the security-sensitive operations. When $CheckLevel$ is equal to one or more, Vanguard will explore the bodies of the caller, the caller's parents, and even the ancestors.

Intuitively, the strategy of exploring proper protection checks is illustrated in Fig.7. Starting from the location of security-sensitive operation using tainted argument *data* in the grey node, Vanguard will explore the body of the caller and the caller's ancestors along every path of the call graph according to the check level.

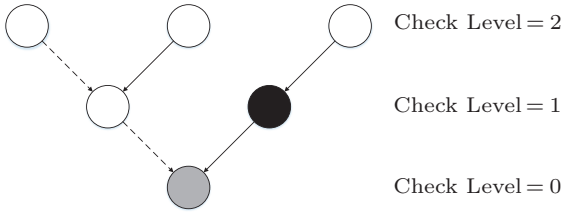


Fig.7. Check levels.

First, we mark argument *data* used by the located sensitive operation as taint source and apply backward

taint analysis to each statement in the body of the caller or caller's ancestors. Second, if we find a conditional statement such as *IfStmt*, *WhileStmt*, *ForStmt* or *SwitchStmt* and their judging conditions contain the tainted expressions affected by tainted argument *data*, we regard it as a proper protection check (represented as the black node). Otherwise, if all the variables occurred in the conditions of detected conditional statements are not affected by tainted argument data along one path (represented as the imaginary arrows) of the call graph, then we identify a missing check.

How to define the proper check condition to detect missing checks is one of the key technical challenges of the approach. For division and modular operations, the condition is simple (i.e., the operand is not equal to zero). For array-index-access, the condition is the boundary of the array. But for sensitive-API usage, the condition is diverse and hard to define precisely. We determine the missing checks for security-sensitive APIs usage by checking if the taint argument occurs in the existing conditions of protection checks. The way to define the proper check is not precise but useful in practice. We implement this strategy based on the assumption that if the developers are aware of adding protection check for the security-sensitive operation and sensitive data, then the developers should write the right protection check condition. This assumption is usually true in practice, especially for experienced developers.

3.3.2 Estimating Risk Degree

The missing check is an indicator of insufficient attack protection. However, it may not necessarily result in a real exploitable vulnerability, which can be triggered by outside attack input. In other words, a missing check may be extremely dangerous in some contexts, but acceptable in the others. Thus, it is necessary to estimate the risk degree of missing checks in their specific contexts for better assessing insufficient attack protection.

We propose a context metrics-based approach to achieve the goal of estimating the risk degree of a missing check in its context. The underline thought is that if the function that has a detected missing check is more complicated, then the missing check is more dangerous and the consequence is more serious if external attackers exploit it. Therefore, we compute the function's context complexity based on its context metrics[32] listed in Table 2 to represent the potential risk degree of detected missing checks.

Once a missing check is identified, context metrics listed in Table 2 of the function with the detected missing check will be extracted via static analysis. Then, the risk degree is calculated based on the values of these metrics using the following equation.

$$RiskDegree = \sum_{i=1}^{N} ContexMetric(i).$$

When the risk degree is larger than the user-defined acceptance limit, a missing check warning is generated, and the detailed information about the missing check will be reported into an XML file as Fig.4. The risk degree of missing checks could be used to filter warnings. We set the acceptance limit to 50 when performing the evaluation in the following. The reason why we set this limit to 50 is that it could filter some functions with low context complexity.

3.4 Repair Protection Recommendation

We first summarize the implementation types of existing proper protection checks, and then propose an approach to generate protection check conditions for various missing checks.

3.4.1 Summary of Implementing Protection Checks

We manually analyze the source code of large-scale open source projects, and study the implementation ways of protection checks for sensitive data used in security-sensitive operations. There are five common ways of implementing protection checks. We take a code sample with the division operation to illustrate the five implementation types of protection check in the following.

Type 1. Implement the protection check by If statement as Fig.8.

```
1 void print_number(int i, int j) {
2     if(j != 0)
3         print("%d\n", i/j);
4 }
```

Fig.8. Deploy constraint protection using $IfStmt$.

Type 2. Implement the protection check by While/For/Switch statement as Fig.9.

```
1 void print_number(int i, int j) {
2     While(j != 0){
3         print("%d\n", i/j);
4         j--;
5     }
6 }
```

Fig.9. Deploy constraint protection using $WhileStmt$.

Type 3. Implement the protection check by assert statement as Fig.10.

```
void print_number(int i, int j) {
    assert (j != 0);
    print("%d\n", i/j);
}
```

Fig.10. Deploy protection check using assert.

Type 4. Implement the protection check by self-defined check function as Fig.11.

```
void print_number(int i, int j) {
    print("%d\n", i/isNotZero(j));
}
```

Fig.11. Deploy protection check using check function.

Type 5. Implement the protection check by ternary expression as Fig.12.

```
void print_number(int i, int j) {
    double result = (j!=0 ? i/j, i);
    print("%d\n", result);
}
```

Fig.12. Deploy protection check using ternary.

Type 1 is mostly used among the above five types of protection check implementation ways. Thus, our repair reference is exported with the style of IfCheck.

3.4.2 Approach of Generating Repair References

Repair References of Missing Checks for Division and Modular Arithmetic. The constraint protection conditions for the missing checks of division and modular arithmetic are clear and determined (i.e., the operand is not equal to 0). When we identify a missing divide-zero or mod-zero check, we could generate the repair protection references for it. The repair references for *dividend* acting as division operand, and *operand* acting as modular operand in Fig.13 are as follows.

```
<ReferableRepair>
<SensitiveOp>i / dividend</SensitiveOp>
<ifCheck> dividend != 0</ifCheck>
</ReferableRepair>
<ReferableRepair>
<SensitiveOp>i % operand</SensitiveOp>
<ifCheck> operand != 0</ifCheck>
</ReferableRepair>
```

Fig.13. Repair references for divide-zero and mod-zero.

Repair References of Missing Checks for Array-Index Access. Because the boundary information for different array-index accesses is different, the protection conditions for different array index accesses are different too. In order to generate useful repair references for missing array-index-bound checks, we propose a static array boundary information construction method based on the extraction of array base, index and declaration size[33]. Once an array is located based on the *ArraySubscriptExpr* in Clang AST, the base and the index of the array are extracted, which could be easily achieved using Clang APIs. Then, we identify the declaration location of the array and obtain its declaration size based on the array base information. Note that we could only extract the array declaration size when the array variable is not an external variable, since AST does not preserve an external variable's size information. At last, we construct the array boundary condition based on its index and size information. Each index should be less than the declaration size of the array. Furthermore, if the type of the index is signed, then the index should be no less than zero. For example, the repair reference for *index* is as shown as Fig.14.

```
<ReferableRepair>
<SensitiveOp> array[index] </SensitiveOp>
<ifCheck> index>=0&&index<MAX_LEN </ifCheck>
</ReferableRepair>
```

Fig.14. Repair references for array-index access.

Repair References of Missing Checks for Sensitive API Usage. For missing checks of sensitive APIs usage, it is difficult to specify their constraint conditions due to the diversity of security-sensitive functions and their arguments. Thus, we generate repair references for missing argument-constraint checks based on the existing protection checks information for the same security-sensitive APIs. We first collect all the proper security check conditions at the same time of detecting missing checks as presented in Subsection 3.3.1. Then we classify the collection of protection checks according to the sensitive APIs and rank them based on the frequency of their occurrence in the project. At last, we integrate the information into a dataset of protection checks C, which could be denoted as below.

$$C : fName, ProperCheck, Frequency,$$

where $fName$ is the function name of the sensitive APIs, $ProperCheck$ is the existing protection check

condition, and *Frequency* represents the occurrence number of the check conditions in the project.

When a missing argument-constraint check is identified, Vanguard searches the dataset of protection checks using the name of the security-sensitive API, and recommends top 5 mostly used protection checks as repair references for the missing check warning as illustrated in Fig.5. Note that our repair recommendations are just references for developers who are in charge of fixing the detected missing checks. We do not insert these repair references automatically and the handling of fixing is handled by developers manually based on developers' professional knowledge.

4 Implementation

An automated and cross-platform tool called Vanguard was developed based on Clang/LLVM 3.6.0. The architecture of Vanguard is illustrated in Fig.15. It consists of five 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 for four kinds of security-sensitive operations with high accuracy and performance; 4) RiskEstimator, which estimates the risk degree of detected missing checks in their contexts by extracting context metrics; 5) RepairRecommender, which recommends repair references for detected missing checks based on the constraint semantic and existing protection checks.

Memory Optimization. In order to improve the ability of Vanguard to analyze large-scale projects in a limited memory environment, a cache mechanism for the read and the write operations of ASTs is proposed to optimize memory usage. The key idea is to preserve the latest used ASTs in memory with an AST queue.

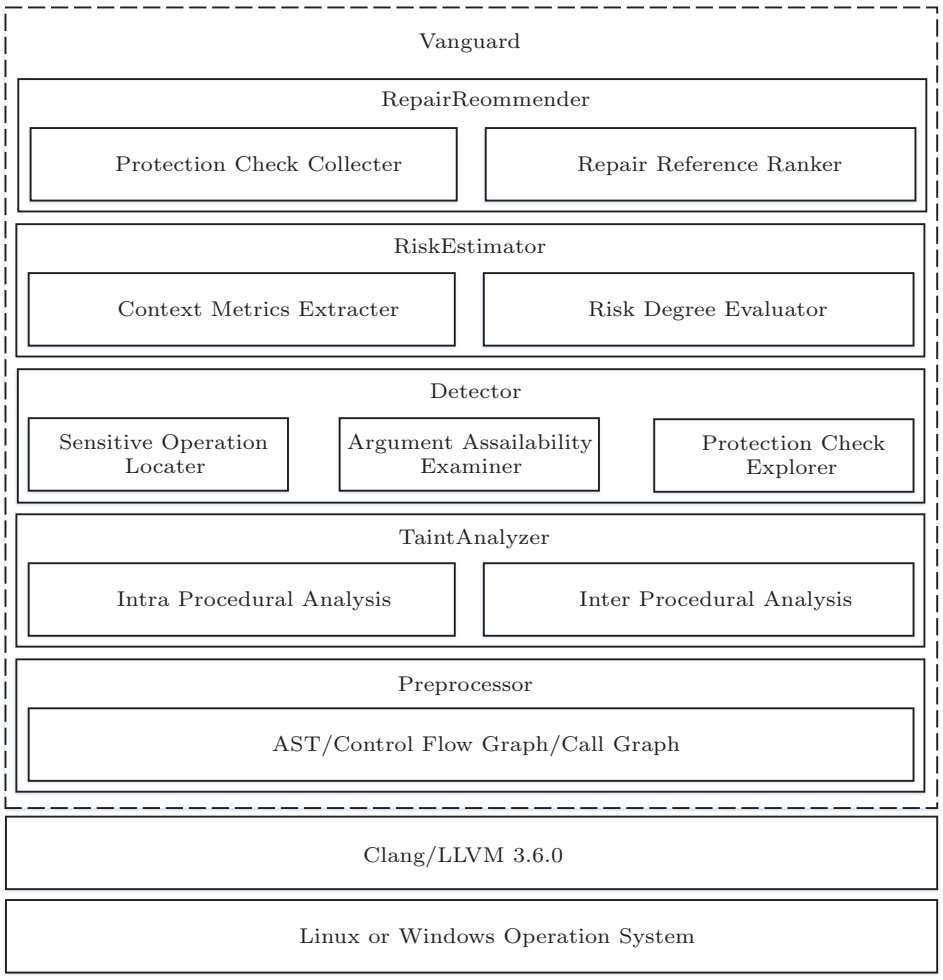


Fig.15. Implementation of Vanguard.

The users could configure the maximal length of AST queue according to practical memory limit.

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

Configurable Optimization. In order to improve the flexibility of static taint analysis, taint sources and taint propagation rules are set to be configurable.

1) Taint sources are configurable to focus on situations that users are interested in. The return values of library functions in the whitelist are all not tainted by default, and the return values of library functions in the blacklist are all tainted by default.

2) Taint propagation rules are configurable to achieve adaptive accuracy and complexity, and avoid over-tainting by handling some complex statements.

Open Source. The core source code of Vanguard is online available[②].

5 Evaluation

A large-scale experimental evaluation was conducted on a computer with 64-bit Ubuntu 16.04 LTS system, a processor of Intel® Xeon® CPU E5-1650 v3 @3.5 GHz and 8 GB RAM. And the evaluation is designed to answer the following research questions.

*RQ*1. How is the effectiveness of Vanguard?

*RQ*2. How is the efficiency of Vanguard?

*RQ*3. How is the performance of Vanguard compared with other tools?

5.1 Effectiveness

We evaluate the effectiveness of Vanguard from four aspects: 1) accuracy of static taint analysis; 2) accuracy of missing check detection; 3) ability to prognosticate vulnerabilities; 4) usability of repair references.

5.1.1 Accuracy of Static Taint Analysis

The effectiveness of missing check detection relies on the accuracy of static taint analysis. A set of typical testing programs[③] is selected to perform the evaluation. The reason why we choose the benchmark is that the benchmark contains all taint propagation situations including the pointer, reference and function calls. In addition, it is usually used to validate the effectiveness of taint analysis techniques. For intuitively, we present a typical code sample in Fig.16 to show the accuracy of our taint analysis. The results from our taint analysis are specified as comments in the code. We will manually audit the code to validate the accuracy of the results in the following.

```
1  int tainted();
2
3  struct A{
4    int m;
5  };
6
7  int func(int in){
8    int a = in;
9    return a;
10   /* TaintValue(func)=Gamma(in) */
11 }
12
13 int pointer_param_in(int* pin){
14   int x = *pin;
15   return x;
16   /* TaintValue(func) = Gamma(in) */
17 }
18
19 int* ref_param_out(int& pout){
20   pout = tainted();
21   return & pout;
22 }
23
24 int test_pointer(){
25   int x = tainted();  /* x= tainted */
26
27   struct A a1;
28   struct A* p1 = &a1;
29   struct A* p2 = p1;
30   /*a1, p1, p2 are untainted*/
31   p1->m = x;  /* a1, p1, p2= tainted */
32
33   int c = func(x); /*c= tainted*/
34   int ret1 = pointer_param_in(&c);
35   /* ret1= tainted */
36
37   int b = 1;        /* b = untainted */
38   int* ret2 = ref_param_out(b);
39   /* b = tainted, ret2= tainted */
40
41   return 0;
42 }
```

Fig.16. Code sample for taint analysis of pointer and reference.

As we can see from Fig.16, the function *tainted*() is in the *blacklist*. The return value of *tainted*() is tainted, thereby x is tainted by taint source at line 25. We regard a STRUCT object as an entirety. If

②https://github.com/stuartly/MissingCheck, July 2019.

③Taint analysis benchmark. https://github.com/dceara/tanalysis/tree/master/tanalysis/tests, July 2019.

any member of the STRUCT is tainted, then the whole STRUCT is tainted. The member variable m of p1 is assigned by x at line 31, and p1 is tainted. Furthermore, a1 and p2 are tainted too since they are pointing the same address. The initial value of variable c is the return value of func(x) at line 33, and the 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; thus c is tainted. Line 34 presents a taint propagation situation of function pointer as an argument. The taint type of return value of function pointer_param_in is Gamma(pin) determined by the taint type of its actual argument. The actual argument of pointer_param_in is the address of c and c is tainted; thus ret1 is tainted. At line 37, b is initialized by 1, and 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; thus the actual argument and the return value of ref_param_out() are tainted, and b and ret2 are tainted. By comparing the results in comments with analyzed results, we can prove our static taint analysis is correct and accurate.

Based on the above analysis, we could conclude that our static taint analysis is able to analyze various C/C++ expressions and taint propagation situations correctly. More specifically, it could handle the propagation of variable definition and assignment (lines 8, 14, 18, and 25), return value (lines 25, 33, 34, and 38), the propagation of structure, pointer and reference assignment (lines 25–28), the propagation of pointer and reference as function arguments (lines 12, 17, 31, 35), etc.

5.1.2 Accuracy of Missing Check Detection

In order to evaluate the accuracy of detecting missing checks, we use Vanguard to analyze the selected large-scale open source projects and count the false positive manually. These open source projects are chosen because 1) related work like Chucky[27] has analyzed some of them; 2) they are widely used and under the active maintenance; 3) they have enough diversity in terms of scale and functionality. We use Vanguard to analyze the selected projects and ask the third party to count the false positive of reported warnings. We determine a missing check warning as a false positive when we manually find a protection check for its sensitive data in the function or its caller.

The result is illustrated in Table 3. Note that *AST* represents the number of abstract syntax trees, *Func* represents the number of functions, *Loc* represents code lines, T(s) represents the analysis time in seconds, and Sp(M) represents the memory overhead in megabytes. The total warnings and the false positive warnings are

Table 3. Effectiveness and Efficiency of Detecting Missing Checks

Project	*AST*	*Func*	*Loc*	T(s)	Sp(M)	Missing Check Warnings			False Positive
						Divide/Mod-Zero	Array-Index	Sensitive-API	
Php-5.6.16	634	8 499	497 602	619.93	2 793.2	43 (14)	34 (2)	96 (7)	13.29%
Openssl-1.1.0	589	5 692	284 518	448.23	858.4	7 (1)	32 (6)	28 (3)	14.93%
Pidgin-2.11.0	38	966	328 153	37.57	471.7	27 (3)	16 (4)	63 (6)	12.26%
Libpng-1.5.21	60	337	24 621	17.69	176.9	3 (0)	4 (0)	13 (3)	15.00%
Libxml2-2.9.9	88	4 618	230 235	47.82	707.1	23 (0)	10 (2)	25 (0)	3.45%
Libtiff-4.0.6	80	790	69 608	18.48	207.3	72 (6)	6 (4)	4 (0)	12.19%
Tengine-2.2.3	127	1 663	173 830	118.87	1 135.5	22 (3)	5 (0)	83 (2)	4.54%
WavPack-5.1.0	23	245	32 923	5.54	78.4	14 (8)	0 (0)	99 (3)	9.73%
Libsass-3.5.5	46	726	29 812	7.11	477.0	4 (8)	4 (0)	29 (7)	18.91%
Jasper-2.0.14	54	674	30 352	12.23	169.1	22 (2)	0 (0)	1 (0)	8.69%
Espruino-2v01	96	1 997	1 141 645	19.97	335.7	8 (3)	8 (1)	6 (3)	31.82%
Libvips-v8.7.4	411	5 333	167 730	1 378.80	1 095.1	236 (4)	13 (2)	27 (4)	5.71%
ImageMagick-7.0.8	255	3 519	564 420	564.33	1 032.2	168 (23)	6 (2)	110 (1)	9.42%
Libgit2-v0.27.7	431	5 973	188 113	668.13	815.2	3 (1)	6 (0)	51 (3)	6.56%
Libharu-2.3.0	58	807	151 996	11.28	225.0	17 (9)	15 (0)	2 (0)	26.47%
Tsar-1.0	30	129	6 138	10.37	149.7	19 (5)	8 (0)	1 (0)	17.85%
Coreutils-8.30	395	1 757	206 751	117.91	10.2	55 (12)	9 (2)	49 (8)	19.48%
Nasm-2.14.02	81	684	93 954	23.04	458.1	0 (0)	4 (3)	18 (0)	13.62%
Libssh2-1.8.0	57	368	31 589	11.34	220.9	1 (0)	7 (1)	20 (1)	7.14%
Libpostal-v1.1.a	43	787	578 235	87.52	1 256.5	9 (2)	0 (0)	64 (8)	13.69%
Average false positive						15.44%	15.43%	7.48%	13.23%

represented with format of $X(Y)$, where X is the number of total warnings and Y is the number of false positive warnings. When we perform the evaluation, the AST queue is set to 100, and the security-sensitive function is a set of memory-related and string-related functions such as memcpy, strcpy and so on.

As we can see from Table 3, Vanguard is able to identify missing checks accurately with low false positive, i.e., 13.23% on average. More precisely, the false positive is 15.44% for detecting missing divide-zero and mod-zero checks, 14.43% for detecting missing array-index-bound checks, and 7.48% for detecting missing argument-constraint checks for sensitive API usage. The cause of false positives is mainly due to the limitation of protection check types we are handling. Some protection checks are implemented using the assert statement, self-defined check function. But we only consider the implementation type 1 and type 2 of implementing protection checks, i.e, $IfStmt$, $WhileStmt$, $ForStmt$, and $SwitchStmt$. The support for type 3–type 5 will be improved in the future work.

5.1.3 *Ability to Prognosticate Vulnerabilities*

By adding known vulnerable functions of above projects as security-sensitive functions into the configuration file, Vanguard's ability to identify missing checks is able to lead us to uncover some known vulnerabilities posted in National Vulnerability Database (NVD)④ in open source projects as illustrated in Table 4. Note that the reason why we choose the five known vulnerabilities is that their root causes are missing checks according to the CVE descriptions. Thus, they are suitable to verify the effectiveness of detecting missing checks.

Table 4. Discovery of Known Vulnerabilities

Project	File	Function	Vulnerability
Openssl-1.1.0	stalen_dtls.c	BUF_MEM_grow_clean	CVE-2016-6308
Pidgin-2.10.11	protocol.c	mxit_send_invite	CVE-2016-2368
Libpng-1.5.21	pngrutil.c	png_read_IDAT_data	CVE-2015-0973
Libtiff-4.0.6	tif_fax3.c	_TIFFFax3Fillruns	CVE-2016-5323
Libtiff-4.0.6	tif_packbits.c	TIFFGetField	CVE-2016-5319

Furthermore, Vanguard has helped us find 12 new bugs which have been confirmed by the maintainers of the open-source projects. Please refer to our website⑤ for more details. Some of them have been validated using dynamic approaches. More specifically, two unknown bugs have been verified by fuzzing that they could be used to crash the jabberd2⑥⑦, which is a widely used XMPP protocol server. Fig.17 presents an instance.

The function config_load_with_id is in charge of turning an XML config file into a config hash. The array of the path is a reference of the 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] → lname to buf. path is a tainted data affected by outside input XML config. There is a missing check for the total size of path[j] → lname. The size of buf is 1 024, a buffer overflow is triggered if the total size of path[j] → lname is larger than 1 024.

```
1  /* turn an xml file into a config hash */
2  int config_load_with_id(config_t c, const
       char *file, const char *id)
3  {
4    ......
5    char buf[1024], *next;
6    ......
7    for(i = 1; i < bd.nad->ecur && rv == 0; i
       ++)
8    {
9      ......
10     next = buf;
11     for(j = 1; j < len; j++)
12     {
13       strncpy(next, bd.nad->cdata + path[j
           ]->iname, path[j]->lname);
14       next = next + path[j]->lname;
15       *next = '.';
16       next++;
17     }
18     next--;
19     *next = '\0';
20     .......
21   }
22   .......
23 }
```

Fig.17. Missing check of Jabberd2.

④National vulnerability database. https://nvd.nist.gov, July 2019.

⑤Website of Vanguard. https://sites.google.com/view/missing-check/home, July 2019.

⑥Bug1. https://github.com/jabberd2/jabberd2/issues/159, July 2019.

⑦Bug2. https://github.com/jabberd2/jabberd2/issues/160, July 2019.

Based on the above analysis, we could conclude that Vanguard is able to prognosticate potential bugs and vulnerabilities.

5.1.4 Usability of Repair References

Our repair references mainly refer to the constraint conditions for sensitive date used in security-sensitive operations. The usability is usually validated by developers who are in charge of fixing detected missing checks. In order to avoid the subjectivity of manual validation to some extent, we randomly choose 100 security sensitive functions with precondition security checks in projects Php-5.6.18, Openssl-1.1.0, Pidgin-2.10.11, Libtiff-4.0.6, Libpng-1.5.21, and Libxml2-2.9.9. We first delete these checks, then use Vanguard to detect these missing checks, and furthermore analyze the recommended repair references. By comparing the repair references with protection checks we deleted before, we determine the usability of repair references. If a repair reference contains the protection checks we deleted, then it is direct usage. If the repair reference could be slightly modified to be the same as the original protection checks, it is indirect usage.

The results are collected and illustrated in Table 5. It shows that more than 66.83% recommended repairs are useful, and nearly 16.5% recommended repairs are used directly. In addition, we observe that the repair reference recommendation performs better in projects with more reused code because these projects will have more similar precondition security checks.

Table 5. Effectiveness of Repair References

Project	Repair Reference	Direct Usage (%)	Indirect Usage (%)
Php-5.6.16	499	18	53
Openssl-1.1.0	497	19	49
Pidgin-2.10.11	258	13	41
Libtiff-4.0.6	487	18	55
Libpng-1.5.21	491	16	48
Libxml2-2.9.9	398	15	56

Summary. Based on the above observations (i.e., Subsection 5.1.1–Subsection 5.1.3), we could positively answer RQ1 that Vanguard is able to detect various missing checks effectively with low false positive and recommend useful repair references.

5.2 Efficiency

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 Static Taint Analysis

We select a taint analysis benchmark mentioned in [34] to evaluate the performance of our static taint analysis algorithm. The reasons why we choose these programs are: 1) they are typical programs used by other taint analysis studies[35], 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. Note that *Loc* represents the code lines of the project. *AST* is the number of AST files, and it also equals the number of source files. $Total$ represents the total occurrence number of variables. Because the tainted environment of each basic block is different, and taint types of variables are context-sensitive, $Total$ counts the occurrence number of all the variables in all the blocks. $TVar$ is the occurrence number of taint variables. $TPerc$ is the result of TVar divided by Total, which represents the dependence degree between program variables and outside input. T(s) is the execution time of taint analysis in seconds, and Sp(M) is memory overhead in megabytes.

As we can see from Table 6, we could know that our static taint analysis performs well in dealing with different scale projects, and the time and memory overhead of taint analysis is low. For instance, it is able to analyze a program with 10k line code (i.e., Mailx) in 2.58 s with 76.7 MB memory cost. It also indicates that

Table 6. Performance of TaintAnalyzer

Project	*Loc*	*AST*	$Total$	$TVar$	$TPerc$(%)	T(s)	Sp(M)
Circles	84	1	197	164	83.25	0.95	0.0
Queue	227	2	244	79	32.38	0.33	0.0
ABR	408	3	626	300	47.92	0.64	0.0
Huffman	499	5	809	426	52.66	0.74	20.6
ArmAssembler	2 071	3	65 024	9 173	14.11	1.69	40.4
Mailx	14 609	29	47 643	15 449	32.43	2.58	76.7

it is able to analyze various complex programs with all kinds of C/C++ expressions and structures such as the pointer, array, reference and so on.

5.2.2 Scalability of Missing Check Detection

We also count the size of analyzed projects as well as their time and memory overhead. As can be seen from Table 3, Vanguard could finish analyzing Php-5.6.16 in 619.93 s, which is a project with more than 500 000 lines code. It proves that Vanguard could handle large-scale projects.

Furthermore, we count the time-overhead of Vanguard on project Php-5.6.16 with the increment of AST files, functions and code lines. All the plots in Fig.18 have shown that Vanguard's complexity is nearly linear, which is scalable on a large size of projects. In addition, the effect of our memory optimizing in Vanguard is evaluated by analyzing Php-5.6.16 with setting the different sizes of the AST queue. The result in Fig.19 indicates that Vanguard is capable of analyzing Php-5.6.16 with lower space-cost when the size of the AST queue is smaller. Obviously, Vanguard will load ASTs more frequently and cost more time at the same time. But when the size of the AST queue is larger than the number of total ASTs of the target project (e.g., 634 for Php-5.6.16), the space-time cost will stay stable (e.g., 5 898 MB and 304 s) since all ASTs will be loaded into memory at the beginning.

Summary. Based on the above observations (i.e., Subsection 5.2.1 and Subsection 5.2.2), we could positively answer RQ2 that Vanguard is capable of dealing with large-scale projects efficiently with low time-space overhead.

5.3 Comparison

As far as we know, the existing studies to detect missing checks are mainly Chucky[27] and RoleCast[28]. We first compare Vanguard with Chucky and RoleCast from three aspects: 1) kinds of programming languages, 2) types of missing checks, and 3) average false positive. The information is specified in Table 7. Note that the average false positive rates of Chucky and RoleCast are collected from their papers[27,28].

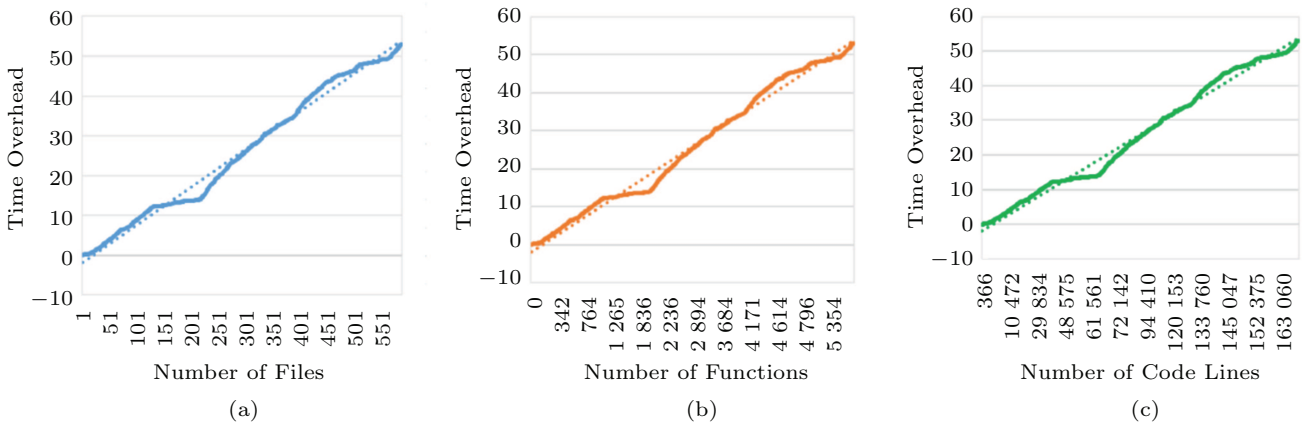


Fig.18. Time overhead with the growth of scale. (a) Number of Files. (b) Number of Functions. (c) Number of code lines.

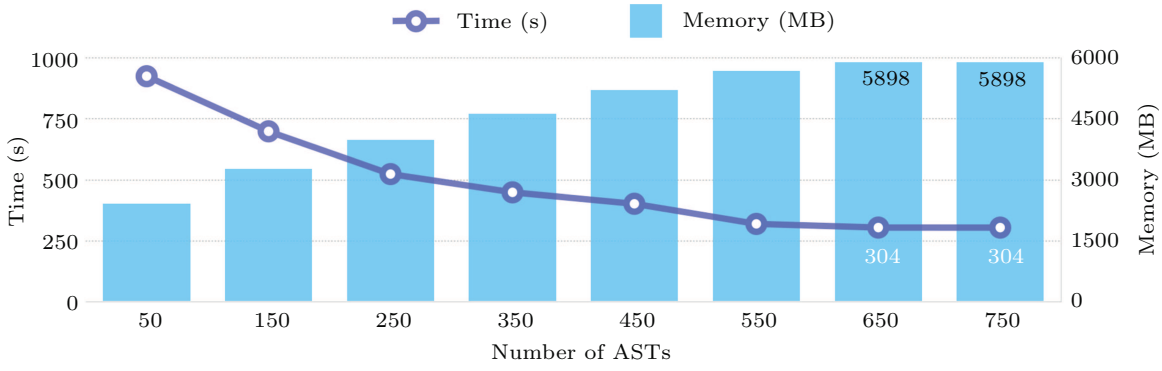


Fig.19. Result of memory optimizing.

Table 7. Vanguard, Chucky and RoleCast

Capability	Vanguard	Chucky	RoleCast
Support Language	C/C++	C/C++/PHP/JSP	PHP/JSP
Detect missing div-zero check	✓	✗	✗
Detect missing mod-zero check	✓	✗	✗
Detect missing array-index-bound check	✓	✗	✗
Detect missing sensitive-API-usage check	✓	✓	✓
Detect missing sensitive-logic check	✗	✓	✓
Detect missing sql-injection check	✗	✗	✓
Average false positive	13%	20%	23%

As we can see from Table 7, Vanguard and Chucky are able to handle C/C++ languages while RoleCast focuses on PHP and JSP languages. All of the three tools are capable of detecting missing checks for sensitive APIs usage. In addition, Vanguard can also detect missing checks for divide-zero, mod-zero, and array-index-bound. Chucky and RoleCast can also detect missing checks for security logic. RoleCast can also handle missing checks for sql-injection. In terms of average false positive of detection, Vanguard performs better.

Furthermore, we perform strict comparison experimental evaluation for Chucky and Vanguard on the same project (i.e., libpng-1.2.44) with the same sensitive APIs (i.e., "png_memcpy", "png_malloc", "png_free", and "strcpy"). The reason why we choose the project is summarized as follows. 1) The project is analyzed by Chucky and the corresponding configuration (i.e., source/sink symbols) is specified in [27]; 2) Chucky is difficult to use due to the lack of maintenance in practice. It is hard to analyze other projects without fixing Chucky's bugs such as the issue in GitHub⑧. We use Chucky and Vanguard to analyze the libpng-1.2.44 for specified sensitive APIs. We collect all Vanguard's missing check warnings, and select Chucky's warnings whose anomaly scores are larger than 0.5. The results are presented in Table 8.

As can be seen from Table 8, Vanguard is able to detect more missing checks than Chucky for the same sensitive APIs, and its false positive is lower. For sensitive-APIs "png_memcpy" and "png_free", Vanguard detects more missing checks including all the real missing check warnings detected by Chucky. Chucky cannot identify some missing checks, because its detection is based on anomaly detection algorithm. Chucky detects missing checks by computing the distance between the sensitive operation and its N neighbors. If the sensitive operation and all its neighbors do not have the protection check, Chucky could not detect it. For example, Vanguard found five "strcpy" without checks, while Chucky could not identify them. For sensitive API "png_malloc", Chucky detects more missing checks than Vanguard, but four of them are "png_malloc(PNG_MAX_PALETTE_LENGTH)". The size argument is a constant, not a variable. It means that this missing check will not be exploitable by outside attack input. Vanguard does not report them because it identifies missing checks for each sensitive operation one by one and would report it only if its sensitive data is tainted.

The time overhead of Chucky is much higher than that of Vanguard. On average, Chucky consumes about 14 times time overhead than Vanguard. The reason is that Chucky spends a lot of time on querying the database which stores the parsed program, and querying the database is time-consuming.

Summary. Based on the above observation (i.e., Subsection 5.3), we could positively answer RQ3 that Vanguard performs better than existing tools in the aspects of supported missing check types, accuracy and performance.

6 Related Work and Discussion

6.1 Taint Analysis

Taint analysis[34,36] attempts to identify variables that have been tainted with user-controllable input. Static taint analysis[18,37] can achieve higher code coverage without run-time overhead compared with dynamic taint analysis[38,39]. Meanwhile, the disadvantage is that it will lose a certain degree of accuracy for the lack of dynamic information. Dytan[39] is a general framework for dynamic taint analysis. Pixy[40] applies static taint analysis to detect SQL injection, cross-site

⑧Issues of Chucky. https://github.com/a0x77n/chucky-ng/issues, July 2019.

Table 8. Comparison Between Vanguard and Chucky on Detecting Missing Checks for Sensitive-API Usage

Project	Sensitive API	Chucky			Vanguard		
		Time (s)	Total_Num (FP_Num)	FP (%)	Time (s)	Total_Num (FP_Num)	FP (%)
libpng 1.2.44	png_memcpy	85.78	11 (5.0)	45.45	6.43	17 (2.00)	11.76
libpng 1.2.44	png_malloc	123.86	10 (2.0)	20.00	6.43	4 (0.00)	0.00
libpng 1.2.44	png_free	134.22	23 (7.0)	30.43	6.43	27 (5.00)	18.51
libpng 1.2.44	strcpy	31.92	0 (0.0)	-	6.43	5 (0.00)	0.00
Average		93.94	11 (3.5)	31.81	6.43	13.25 (1.75)	13.20

scripting or command injection bugs in PHP scripts. Safer[41] is a tool combining taint analysis with control dependency analysis to detect control structures that can be triggered by untrusted user input. Inspired by [39], we design and implement an extensible static taint analysis framework including intra-procedural and inter-procedural analysis with features of controllable taint sources and taint propagation rules. It is used to determine whether sensitive data used by security-sensitive operators is assailable by the attack input or not.

6.2 Missing Check Detection

Chucky[27] is a missing check detection tool using intra-procedural static taint analysis and machine learning. It identifies missing checks for security logic and API usage based on the assumption that the missing check is a rare event. Therefore, it is more suitable for analyzing mature code because its assumption is usually not valid in the software under development stage. Different from Chucky's detection for missing checks using machine learning (i.e., anomaly detection algorithm), Vanguard identifies missing checks by pure static analysis including intra-procedural and inter-procedural taint analysis. In addition, Vanguard is able to identify missing checks for more types of security-sensitive operations including division arithmetic, modulus operation and array-index access without Chucky's assumption. Our tool is aimed to improve the code's correctness, which can be used on mature code and software under the development stage.

RoleCast[28] is a static analysis tool to identify security-related events such as database writes in web applications by using a consistent web application pattern without specification. 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 focuses on software written in C/C++ language.

6.3 Warnings Ranking and Validation

Z-Ranking[42] prioritizes warnings based on the frequency of review results. Kim and Ernst proposed a history-based warning prioritization (HWP) algorithm by exploiting the relationship between warnings and bug fixes in the software change history[43]. Li *et al.*[9] proposed an approach to reduce the human validation effort by dynamically classifying statically generated memory leak warnings. Clarify[44] is a tool created to improve error reporting by learning the behaviors of an application based on the summary of its execution history. Our previous work[45] estimated the risk degree of detected missing checks based on their context metrics. We extend previous work[45] by adding more context metrics.

6.4 Program Repair techniques

The mainstream program repair approaches could be classified into syntax-based and semantic-based techniques. Syntax-based repair techniques such as GenProg[46], RSRepair[47] and ACS[48] require subtasks including fault localization, patch generation and execution of regression test cases. Relifix[49] utilizes previous program versions in order to perform the automated repair of regression bugs, and it relies on the syntactic similarity of the previous and the buggy program. Semantics-based techniques like SemFix[50], DirectFix[25], Angelix[51] and JFIX[52] split patch generation into two phases. First, they infer a synthesis specification for the identified program statements. Second, they synthesize a patch for these statements based on the inferred specification. DeepFix[53] is a multi-layered sequence-to-sequence neural network which is trained to predict erroneous program locations along with the required correct statements. WeakAssert[54] is a weakness-oriented assertion recommendation toolkit. It matches the abstract syntax trees with well-designed weakness patterns and inserts assertions into proper locations automatically, in order to identify

potential weakness by verification. Different from weekassert that focuses on generating and inserting assertions automatically for identifying weakness, Vanguard focuses on recommending proper protection conditions as repair references for developers who are in charge of fixing detected missing checks. The fixing of missing checks is done by developers manually. We extend our previous work[45] to recommend constraint conditions for four kinds of missing checks. More specifically, we generate the repair references based on 1) constraint semantic for dive-zero and mod-zero, 2) summary of array boundary information, and 3) existing similar protection checks for the sensitive API usage.

7 Conclusions

An automatic static detection and repair recommendation system (i.e., Vanguard) for missing checks in C/C++ programs was designed and implemented on top of Clang/LLVM 3.6.0, which is aimed at prognosticating potential vulnerabilities and improving the correctness of programs. It is able to identify high-risk missing checks by 1) locating customized security-sensitive operations, 2) judging the 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 context metrics, and 4) recommending repair references for high-risk missing checks. Large-scale experimental evaluation on open source projects showed the effectiveness and efficiency of Vanguard. Furthermore, Vanguard's ability to identify missing checks led us to uncover five known vulnerabilities and 12 new bugs.

Future work includes automatic validation of detected missing checks by dynamic analysis and improvement of the automatic repair recommendation technique.

References

[1] Piromsopa K, Enbody R J. Survey of protections from buffer-overflow attacks. *Engineering Journal*, 2011, 15(2): 31-52.

[2] Sipser M. Introduction to the Theory of Computation (2nd edition). Course Technology, 2006.

[3] Gao F, Wang L, Li X. BovInspector: Automatic inspection and repair of buffer overflow vulnerabilities. In *Proc. the 31st Int. Conference on Automated Software Engineering*, September 2016, pp.786-791.

[4] Chen G, Jin H, Zou D, Zhou B, Liang Z, Zheng W, Shi X. SafeStack: Automatically patching stack-based buffer overflow vulnerabilities. *IEEE Trans. Dependable and Secure Computing*, 2013, 10(6): 368-379.

[5] Wang T, Wei T, Lin Z, Zou W. IntScope: Automatically detecting integer overflow vulnerability in X86 binary using symbolic execution. In *Proc. the 16th Network and Distributed System Security Symposium*, February 2009, Article No. 17.

[6] Dietz W, Li P, Regehr J, Adve V. Understanding integer overflow in C/C++. *ACM Trans. Software Engineering and Methodology*, 2015, 25(1): Article No. 2.

[7] Lee B, Song C, Jang Y *et al*. Preventing use-after-free with Dangling pointers nullification. In *Proc. the 22th Annual Network and Distributed System Security Symposium*, February 2015, Article No. 21.

[8] Yan H, Sui Y, Chen S, Xue J. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proc. the 40th Int. Software Conference on Engineering*, May 2018, pp.327-337.

[9] Li M, Chen Y, Wang L, Xu G. Dynamically validating static memory leak warnings. In *Proc. the 22nd Int. Symposium on Software Testing and Analysis*, July 2013, pp.112-122.

[10] Sui Y, Ye D, Xue J. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Trans. Software Engineering*, 2014, 40(2): 107-122.

[11] Szekeres L, Payer M, Wei T, Dawn S. SoK: Eternal war in memory. In *Proc. the 34th Int. IEEE Symposium on Security and Privacy*, May 2013, pp.48-62.

[12] Das A, Lal A. Precise null pointer analysis through global value numbering. In *Proc. the 15th Int. Symposium on Automated Technology for Verification and Analysis*, October 2017, pp.25-41.

[13] Akritidis P, Costa M, Castro M, Hand S. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proc. the 18th USENIX Security Symposium*, August 2009, pp.51-66.

[14] Kim D, Nam J, Song J *et al*. Automatic patch generation learned from human-written patches. In *Proc. the 35th Int. Conference on Software Engineering*, May 2013, pp.802-811.

[15] Li P, Cui B. A comparative study on software vulnerability static analysis techniques and tools. In *Proc. the 2010 International Conference on Information Theory and Information Security*, Dec. 2010, pp.521-524.

[16] Wagner D A, Foster J S, Brewer E A, Aiken A. A first step towards automated detection of buffer overrun vulnerabilities. In *Proc. the 7th Network and Distributed System Security Symposium*, February 2000, Article No. 1.

[17] Situ L, Zou L, Wang L, Liu Y, Mao B, Li X. Detecting missing checks for identifying insufficient attack protections. In *Proc. the 40th Int. Conference on Software Engineering*, May 2018, pp.238-239.

[18] Ming J, Wu D, Xiao G, Wang J, Liu P. TaintPipe: Pipelined symbolic taint analysis. In *Proc. the 24th USENIX Security Symposium*, August 2015, pp.65-80.

[19] Cadar C, Sen K. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 2013, 56(2): 82-90.

[20] Li Y, Su Z, Wang L, Li X. Steering symbolic execution to less traveled paths. In *Proc. the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, October 2013, pp.19-32.

[21] Majumdar R, Sen K. Hybrid concolic testing. In *Proc. the 29th Int. Conference on Software Engineering*, May 2007, pp.416-426.

[22] Seo H, Kim S. How we get there: A context-guided search strategy in concolic testing. In *Proc. the 22nd Int. Symposium on Foundations of Software Engineering*, November 2014, pp.413-424.

[23] Bérard B, Bidoit M, Finkel A, Laroussinie F, Petit A, Petrucci L, Schnoebelen P, McKenzie P. Systems and Software Verification: Model-Checking Techniques and Tools. Springer, 2001.

[24] Situ L, Zhao L. CSP bounded model checking of preprocessed CTL extended with events using answer set programming. In *Proc. the 2015 Asia-Pacific Software Engineering Conference*, December 2015, pp.16-23.

[25] Sutton M, Greene A, Amini P. Fuzzing: Brute Force Vulnerability Discovery (1st edition). Addison-Wesley Professional, 2007.

[26] Stephens N, Grosen J, Salls C *et al*. Driller: Augmenting fuzzing through selective symbolic execution. In *Proc. the 23rd Annual Network and Distributed System Security Symposium*, February 2016, Article No. 28.

[27] Yamaguchi F, Wressnegger C, Gascon H *et al*. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proc. the 2013 ACM SIGSAC Conference on Computer & Communications Security*, November 2013, pp.499-510.

[28] Son S, McKinley K S, Shmatikov V. RoleCast: Finding missing security checks when you do not know what checks are. In *Proc. the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems*, October 2011, pp.1069-1084.

[29] Lattner C. LLVM and Clang: Next generation compiler technology. https://clvm.org, July 2019.

[30] Ryder B G. Constructing the call graph of a program. *IEEE Trans. Software Engineering*, 1979, 5(3): 216-226.

[31] Stanier J, Watson D. Intermediate representations in imperative compilers: A survey. *ACM Computing Surveys*, 2013, 45(3): Article No. 26.

[32] Pendleton M, Garcia-Lebron R, Cho J H, Xu S. A survey on systems security metrics. *ACM Computing Surveys*, 2017, 49(4): Article No. 62.

[33] Gao F, Chen T, Wang Y, Situ L, Wang L, Li X. Carraybound: Static array bounds checking in C programs based on taint analysis. In *Proc. the 8th Int. Asia-Pacific Symposium on Internetware*, September 2016, pp.81-90.

[34] Ceara D, Potet M L, Ensimag G I, Mounier, L. Detecting software vulnerabilities-static taint analysis. https://docplayer.net/18105375-Detecting-software-vulnerabilities-static-taint-analysis.html, July 2019.

[35] Lalande J F, Heydemann K, Berthomé P. Software countermeasures for control flow integrity of smart card C codes. In *Proc. the 19th European Symposium on Research in Computer Security*, Sept. 2014, pp.200-2018.

[36] Kang M G, McCamant S, Poosankam P, Dawn S. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proc. the 18th Network and Distributed System Security Symposium*, February 2011, Article No. 15.

[37] Li L, Bartel A, Klein J, Traon Y L, Arzt S, Rasthofer S, Bodden E, Octeau D, Mcdaniel P. I know what leaked in your pocket: Uncovering privacy leaks on Android Apps with static taint analysis. arXiv:1404.7431, 2014. https://arxiv.org/pdf/1404.7431.pdf, June 2019.

[38] Schwartz E J, Avgerinos T, Brumley D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proc. the 31st IEEE Symposium on Security and Privacy*, May 2010, pp.317-331.

[39] Clause J, Li W, Orso A. Dytan: A generic dynamic taint analysis framework. In *Proc. the 16th ACM/SIGSOFT Int. Symposium on Software Testing and Analysis*, July 2007, pp.196-206.

[40] Jovanovic N, Krüegel C, Kirda E. Pixy: A static analysis tool for detecting Web application vulnerabilities (Short Paper). In *Proc. the 27th IEEE Symposium on Security and Privacy*, May 2006, pp.258-263.

[41] Chang R, Jiang G, Ivancic F, Sankaranarayanan S, Shmatikov V. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *Proc. the 22nd IEEE Computer Security Foundations Symposium*, July 2009, pp.186-199.

[42] Kremenek T, Engler D. Z-Ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Proc. the 10th Int. Symposium on Static Analysis*, June 2003, pp.295-315.

[43] Kim S, Ernst M D. Prioritizing warning categories by analyzing software history. In *Proc. the 4th Workshop on Mining Software Repositories*, May 2007, pp.27-31.

[44] Ha J, Rossbach C J, Davis J V, Roy I, Ramadan H E, Porter D E, Chen D L, Witchel E. Improved error reporting for software that uses black-box components. In *Proc. the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2007, pp.101-111.

[45] Situ L, Wang L, Liu Y, Mao B, Li X. Vanguard: Detecting missing checks for prognosing potential vulnerabilities. In *Proc. the 10th Asia-Pacific Symposium on Internetware*, September 2018, Article No. 5.

[46] Goues L C, Nguyen T V, Forrest S *et al*. GenProg: A generic method for automatic software repair. *IEEE Trans. Software Engineering*, May 2011, 38(1): 54-72.

[47] Qi Y, Mao X, Lei Y, Dai Z, Wang C. The strength of random search on automated program repair. In *Proc. the 36th Int. Conference on Software Engineering*, May 2014, pp.254-265.

[48] Xiong Y, Wang J, Yan R, Zhang J, Han S, Huang G, Zhang L. Precise condition synthesis for program repair. In *Proc. the 39th Int. Conference on Software Engineering*, May 2017, pp.416-426.

[49] Tan S H, Roychoudhury A. Relifix: Automated repair of software regressions. In *Proc. the 37th IEEE/ACM International Conference on Software Engineering*, May 2015, pp.471-482.

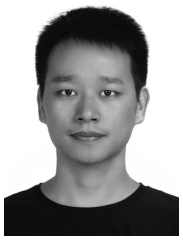
[50] Nguyen H D T, Qi D, Roychoudhury A, Chandra S. SemFix: Program repair via semantic analysis. In *Proc. the 35th Int. Conference on Software Engineering*, May 2013, pp.772-781.

[51] Mechtaev S, Yi J, Roychoudhury A. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proc. the 38th Int. Conference on Software Engineering*, May 2016, pp.691-701.

[52] Le X B D, Chu D H, Lo D, Le G C, Visser W. JFIX: Semantics-based repair of Java programs via symbolic PathFinder. In *Proc. the 26th ACM SIGSOFT Int. Symposium on Software Testing and Analysis*, July 2017, pp.376-379.

[53] Gupta R, Pal S, Kanade A, Shevade S. DeepFix: Fixing common C language errors by deep learning. In *Proc. the 31st AAAI Conference on Artificial Intelligence*, February 2017, pp.1345-1351.

[54] Wang C, Jiang Y, Zhao X, Song X, Gu M. Weak-Assert: A weakness-oriented assertion recommendation toolkit for program analysis. In *Proc. the 40th Int. Conference on Software Engineering*, May 2018, pp.69-72.



Ling-Yun Situ is a Ph.D. candidate in State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing. He received his B.S. degree in computer science from Jiangsu University, Zhenjiang, in 2011, and his M.S. degree in computer science from Guilin University of Electronic and Technology, Guilin, in 2014. His research interests include software and system security, static analysis and fuzzing.

Lin-Zhang Wang is a professor in State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing. He received his Ph.D. degree in computer science from Nanjing University, Nanjing, in 2005. His research interests include software security testing, model based testing and verification.

Yang Liu is an associate professor in School of Computer Science and Engineering, Nanyang Technological University, Singapore. He received his B.S. degree in computer science from National University of Singapore, Singapore, in 2005, and his Ph.D. degree in computer science from National University of Singapore, Singapore, in 2010. His research interests include formal methods, security, software engineering and multi-agent systems.

Bing Mao is a professor in State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing. He received his Ph.D. degree in computer science from Nanjing University, Nanjing, in 1997. His research interests include distributed operating system, software and system security.

Xuan-Dong Li is a professor in State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing. He received his B.S. degree in 1985, M.S. degree in 1991, and Ph.D. degree in 1994, all in computer science from Nanjing University, Nanjing. His research interests include software modelling and analysis, software testing and verification.