Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2018-IC-003

2018-IC-003

Vanguard: Detecting Missing Checks for Prognosing Potential Vulnerabilities

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

Internetware 2018

Vanguard: Detecting Missing Checks for Prognosing Potential Vulnerabilities

Lingyun Situ
Nanjing University
Nanjing, China
situlingyun@seg.nju.edu.cn

Linzhang Wang
Nanjing University
Nanjing, China
lzwang@nju.edu.cn

Yang Liu
Nanyang Technological University
Singapore
yangliu@ntu.edu.sg

Bing Mao
Nanjing University
Nanjing, China
maobing@nju.edu.cn

Xuandong Li
Nanjing University
Nanjing, China
lxd@nju.edu.cn

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.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; *Vulnerability scanners*;

KEYWORDS

Missing Checks, Static Analysis, Vulnerability Prognosis

ACM Reference format:
Lingyun Situ, Linzhang Wang, Yang Liu, Bing Mao, and Xuandong Li. 2018. Vanguard: Detecting Missing Checks for Prognosing Potential Vulnerabilities. In *Proceedings of The Tenth Asia-Pacific Symposium on Internetware, Beijing, China, September 16, 2018 (Internetware '18),* 10 pages.
https://doi.org/10.1145/3275219.3275225

Internetware '18, September 16, 2018, Beijing, China

ACM ISBN 978-1-4503-6590-1/18/09…$15.00
https://doi.org/10.1145/3275219.3275225

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 overflow [15, 42], use-after-free [22, 41], memory leakage [24, 37], s [14, 39], 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 [7, 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 arithmetics) 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.

```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\n", 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\n", 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);
}
```

Listing 1: Code samples of missing checks

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.

- **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 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 *operand* is used as the second operand in modular operation at line 17 without guaranteeing that *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 *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 commented at line 24, which will cause an out-of-bounds error. It is defined as a "missing array-index-bound protection check".
- **Missing Check for Sensitive Function Call**: The manipulable data *len* is used as an argument of a security-sensitive function call (i.e., *memcpy*) at line 34 without comparing *len* and size of *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".

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

Listing 2: Missing Check Warnings

2.2 Formal Definition

$$
\begin{array}{rcl}
\textbf{\textit{stmt s}} & ::= & \textbf{\textit{id}} \leftarrow \textbf{\textit{expr}} \\
 & & \mid \textbf{\textit{call\_func}} \\
 & & \mid \textbf{\textit{s}};\ \textbf{\textit{s}}' \\
 & & \mid \textit{if}\ \textbf{\textit{expr}}\ \textit{then}\ \textbf{\textit{s}}\ \text{else}\ \textbf{\textit{s}}' \\
 & & \mid \textit{while}\ \textbf{\textit{expr}}\ \textit{do}\ \textbf{\textit{s}} \\
\textbf{\textit{expr e}} & ::= & \textbf{\textit{id}} \\
 & & \mid \textbf{\textit{constant}} \\
 & & \mid \textbf{\textit{e}}_1 \diamond_b \textbf{\textit{e}}_2 \\
 & & \mid \diamond_u\ \textbf{\textit{e}} \\
 & & \mid \textbf{\textit{e}}_1 \diamond_m \textbf{\textit{e}}_2 \\
 & & \mid \textbf{\textit{e}}_1\ [\textbf{\textit{e}}_2] \\
\textbf{\textit{call\_func c}} & ::= & \textbf{\textit{e}} \leftarrow \textit{call}\ \textbf{\textit{func}}\ (\textit{id}' = \textit{e})^* \\
\textbf{\textit{func f}} & ::= & \textbf{\textit{signature func\_body}} \\
\textbf{\textit{signature}} & ::= & \textbf{\textit{fname id}}^* \\
\textbf{\textit{func\_body}} & ::= & \textbf{\textit{stmt}}^*
\end{array}
$$

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 local variables and formal parameter of functions, and *constant* represent constant variables. We use $\diamond_b$ and $\diamond_u$ to represent typical binary and unary operations, $\diamond_m$ to represent member operator "." or "→", and [] 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:

- $S = Taint(Var) \times Check(Var)$ 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:

$$\frac{t_{i-1} \overset{\alpha_i}{\hookrightarrow}_t t_i, c_{i-1} \overset{\alpha_i}{\hookrightarrow}_c c_i}{< t_{i-1}, c_{i-1} > \xrightarrow{\alpha_i} < t_i, c_i >}$$

  where α_i is the act, $\hookrightarrow_t \subseteq Taint(Var) \times Act \times Taint(Var)$, $\hookrightarrow_c \subseteq Check(Var) \times Act \times Check(Var)$.
- $I \subseteq S$ is a set of initial states.
- $AP = Taint(Var) \cup Check(Var)$ is a set of atomic propositions.
- $L = S \rightarrow 2^{AP}$ is a labeling function.

Let $\rho =< s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 ... s_{i-1} \xrightarrow{\alpha_i} s_i ... >$ 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. 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.

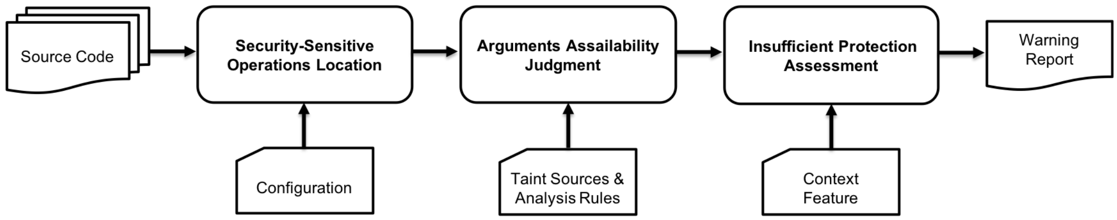

Figure 1: Overview of Vanguard

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 *recvmsg*, *DIV_msg*, *MOD_msg*, *ARRAY_msg*, and *FUNC_msg*. The *recvmsg* 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 *memcpy* 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 *memcpy* are obtained according to our configuration item "*memcpy* : 0 + 2", which represents that the first and third arguments of "*memcpy*" 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 *recvmsg* 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 $msg \rightarrow msg\_len$ before the call site of the sensitive function call *memcpy*. 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.

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

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

$$FUNCTION : fName : Args$$

FUNCTION represents that the type of security-sensitive operation is function, $fName$ is a list of sensitive functions' names related to memory operations (e.g., *malloc*, *memset*, and *memcpy*) 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 $fName$, 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

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

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

$$\varsigma = \{x \mid x \in ArgsEntry \cup ApiRet\}$$

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 white-list and a black-list by users. Let $\tau = \{T, U\}$ 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 $Vars = LocalVars \cup FormalParams$, $LocalVars$ 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 *Vars* by defining a mapping Γ from *Vars* to taint types in the following way:

$$\Gamma : Vars \rightarrow \tau.$$

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

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

where x and y are expressions of the left and right side of some operations *op*. 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 $t1$, $t2$, 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:

$$\Gamma = \Gamma_1 \oplus \Gamma_2 \;\; \textit{iff} \;\; \forall x \in Vars \Rightarrow \Gamma(x) = \Gamma_1(x) \oplus \Gamma_2(x)$$

Algorithm 1: BFSTaintSpread (*CG*, *fEnvs*)

Input: *CG*: *non-recursive Call Graph*; *fEnvs*: *tait environment of vars and formal parameter*

Output: $fEnvs^{'}$: *updated taint environment*

```
foreach v in CG do
    color[v] = WHITE ;
s= CG.start() ;
color[s]= GRAY;
ENQUEUE(Q,s);
while Q != EMPTY do
    u=DEQUEUE(Q);
    foreach v in callee(u) do
        TaintPropagationThroughCall(u,v)
    if color[v]==WHITE then
        ENQUEUE(Q,v)
    color[u]= BLACK;
```

Let *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:

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

Initially, Γ_{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 $assigned(stmt)$ represents the set of left expressions of assignment statements in $stmt$, and $G(id_i) \leftarrow \tau_i$ 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 is 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 *fEnvs* storing taint relations between formal parameters and local variables. The outputs are taint environments $fEnvs^{'}$ 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.

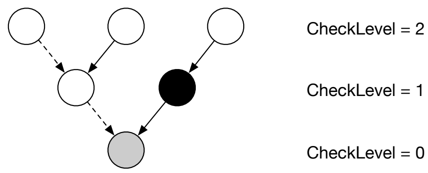

Figure 2: N Level Check

Furthermore, we established 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 continent 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 *data*. These information include function declaration, block, statement and argument expression, they 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 detected, then it is possible to be exploited and attacked by outside input. We perform insufficient protection assessment by exploring the existence of related attack protection checks and (2) further estimating the risk degree of detected missing check in its context based on context features. The basic idea is that if there is a missing check and the function context that the missing check occurred is complex, then the missing check is more likely to be dangerous.

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

$$CheckLevel : N$$

which determines the levels of caller's ancestors we will explore along one path of the call graph. When the *CheckLevel* is equal to "0", *Vanguard* will explore proper attack protection checks in the body of caller invoking the security-sensitive operations. When the *CheckLevel* is equal to "1" or more, *Vanguard* will explore the bodies of the caller, caller's parents, and even ancestors.

Intuitively, the strategy of exploring proper attack protection checks is illustrated in Fig. 2. Starting from the location of security-sensitive operation using tainted argument *data* in the grey node. *Vanguard* will explore the body of caller and caller's ancestors along every path of call graph according to the check level.

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 caller or caller's ancestors. Then, if we find a *IfStmt* and there are variables occurred in the condition of these conditional statements affected by tainted argument data. Then we regard it is as a proper protection check (represented as the block 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)

Table 2: Context features

No.	Feature Names	Feature Meaning
1	NumOfArg	Num of arguments
2	NumOfPlus	Num of "+" in arguments
3	NumOfMinus	Num of "-" in arguments
4	NumOfMultiply	Num of "*" in arguments
5	NumOfDivide	Num of "/" in arguments
6	NumOfDelively	Num of "%" in arguments
7	NumOfSimpleVar	Num of simple vars in arguments
8	NumOfCompositeVar	Num of composite vars in arguments
9	NumOfSizeof	Num of sizeof Ops in arguments
10	NumOfCallerVar	Num of variables in caller
11	NumOfCallerCallExpr	Num of CallExpr in caller
12	CalleeHasBody	Whether the callee has body
13	NumOfInBinary	Num of arguments in binary ops
14	NumOfBinaryOP	Num of binary Ops in caller
15	NumOfTaintArg	Num of tainted arguments

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.

Note that the way to define proper check is not accurate enough but useful in reality based on the assumption that if developers are aware of adding protection check for security-sensitive operation, then the developer will write right protection check conditions. Furthermore, context features listed in Table 2 of function with detected missing check will be extracted to represent its risk degree.

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' read 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

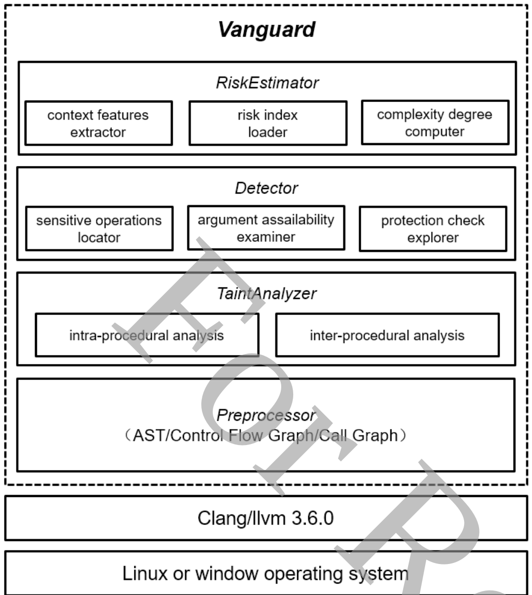

Figure 3: Architecture of *Vanguard*

of their products under development. The core source code of *Vanguard* is available for download from https://github.com/stuartly/MissingCheck.

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 results of TaintAnalyzer as comments, which are obtained by query interface mentioned above in practice. Then we manual audit the 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 taint 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

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

Listing 3: Testing for taint analysis of pointer and reference

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 c is 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).

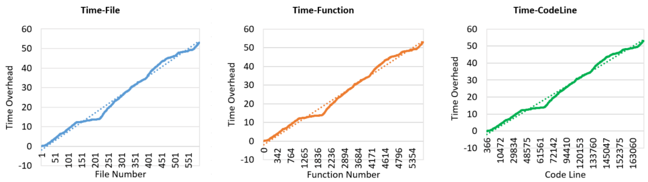

Figure 4: Time overhead with growth of scale

5.1.2 False Positive of Missing Check Detection. Furhtermore, we choose PHP, Openssl, Pidgin, Libtiff, Libpng as testing targets to count the false positive of missing check detection. They are chosen because (1) related work like *Chucky* [43] have analyzed them; (2) they vary from different scale and have vulnerabilities caused by missing checks reported in CVE/NVD [4]. We used *Vanguard* to analysis above projects and asked third party to count the false positive of our report about various missing attack protection checks, the result is illustrated in Table 3, where AST queue is set to 100, security-sensitive function is a set of memory related functions like "memcpy" .etc.

The results show that our *Vanguard* is able to identify missing checks accurately with low false positive, i.e. 19% in average. The causes of false positives are mainly due to two situations: (1) "if(a) sensitive-op((size_t)a)" or "if(a) sensitive-op(a+b*c)", the taint data is a expression with multiple variables, and one variable is checked. (2) Some checks occur in the WHILE, FOR, SWITCH and ASSERT statements, while we only analyze the situation of IF statement. It can be improved in future work.

5.1.3 Discovery of Vulnerabilities. By add vulnerable functions of above projects as security-sensitive operations into configuration, *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. Furthermore, Vanguard has helps us to find two unknown crash bugs in open source projects jabberd2 [2] [3], which is a widely used XMPP protocol server.

```
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 }
```

Listing 4: Missing check of Jabberd2.

One example is illustrated in 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]->lname. The size of buf is 1024, a buffer overflow may happy if the total size of path[i]->lname 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. $Total = \sum f.NumOfBB * f.NumOfVar$. *TVar* is occurrence number of taint variables. $TPerc = TVar\ Total$, which represents the dependence degree between program variables and outside input. *T*(s) is the time of taint analysis, and *Sp*(M) 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.

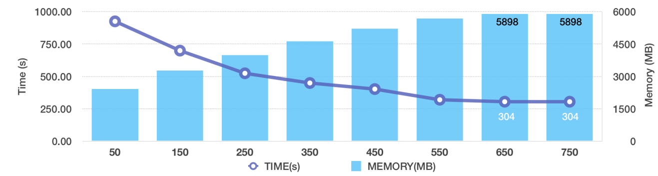

Figure 5: Result of memory optimizing

Table 3: Effectiveness and Efficiency of Vanguard

Project	AST	Func	Loc	Time	Sp(M)	missing checks warnings(false positive warnings)			Average False Positive
						divide/mod-zero	array-index-bound	argument-constraint	
Php-5.6.16	634	8499	497602	619.93	2793.2	43(14)	34(2)	196(83)	36%
Openssl-1.1.0	589	5692	284518	448.23	858.4	7(1)	32(6)	28(6)	19%
Pidgin-2.11.0	38	966	328153	37.57	471.7	27(3)	16(4)	63(11)	16%
Libtiff-4.0.6	83	629	66855	15.52	152.8	57(5)	5(1)	4(1)	10%
Libpng-1.5.21	60	337	24621	17.69	176.9	3(0)	4(0)	13(3)	15%

Table 4: Discovery of Vulnerabilities

Project	File	Function	Vulnerability
Openssl-1.1.0	statem_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

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 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

	C	C++	PHP	JSP
missing divide-zero check	✓	✓		
missing mod-zero check	✓	✓		
missing array-index-bound check	✓	✓		
missing sensitive-APIs usage check	✓ †	✓ †	•	•
missing security logic check	†	†	•	•
missing sql-injection check			•	•
Tool: Vanguard(✓)	Chucky(†)		RoleCast(•)	
False Positive: 19%	<20%		23%	

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.

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
Queue	227	2	244	79	32.38	0.33	0
ABR	408	3	626	300	47.92	0.64	0
Huffman	499	5	809	426	52.66	0.74	20.6
mailx	14609	29	47643	15449	32.43	2.58	76.7

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. Pixy [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/LLVM 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

[1] 2009. TaintAnalysisBenchmark. https://github.com/dceara/tanalysis/tree/master/tanalysis/tests. (2009).
[2] 2017. Bug1. https://github.com/jabberd2/jabberd2/issues/160. (2017).
[3] 2017. Bug2. https://github.com/jabberd2/jabberd2/issues/159. (2017).
[4] 2017. National Vulnerability Database. https://nvd.nist.gov. (2017).
[5] 2017. OWASP2017. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project. (2017).
[6] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors.. In *USENIX Security Symposium*. 51–66.
[7] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. 2013. *Systems and software verification: model-checking techniques and tools*. Springer Science & Business Media.
[8] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.
[9] Jun Cai, Peng Zou, Jinxin Ma, and Jun He. 2016. SwordDTA: A dynamic taint analysis tool for software vulnerability detection. *Wuhan University Journal of Natural Sciences* 21, 1 (2016), 10–20.
[10] Dumitru Ceara, Marie-Laure Potet, Grenoble INP ENSIMAG, and Laurent MOUNIER. 2009. Detecting Software Vulnerabilities-Static Taint Analysis. *Vérimag-Distributed and Complex System Group, Polytechnic University of Bucharest* (2009).
[11] Richard Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. 2009. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *Computer Security Foundations Symposium, 2009. CSF'09. 22nd IEEE*. IEEE, 186–199.
[12] Gang Chen, Hai Jin, Deqing Zou, Bing Bing Zhou, Zhenkai Liang, Weide Zheng, and Xuanhua Shi. 2013. Safestack: Automatically patching stack-based buffer overflow vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 10, 6 (2013), 368–379.
[13] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 196–206.
[14] Ankush Das and Akash Lal. 2017. Precise Null Pointer Analysis Through Global Value Numbering. *arXiv preprint arXiv:1702.05807* (2017).
[15] Will Dietz, Peng Li, John Regehr, and Vikram Adve. 2015. Understanding integer overflow in C/C++. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 1 (2015), 2.
[16] François Dupressoir, Andrew D Gordon, Jan Jürjens, and David A Naumann. 2014. Guiding a general-purpose C verifier to prove cryptographic protocols. *Journal of Computer Security* 22, 5 (2014), 823–866.
[17] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*. IEEE, 6–pp.
[18] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation.. In *NDSS*.
[19] Uday P Khedker. 2014. Buffer Overflow Analysis for C. *arXiv preprint arXiv:1412.5400* (2014).
[20] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 802–811.
[21] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*. 1–2.
[22] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-after-free with Dangling Pointers Nullification.. In *NDSS*.
[23] Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. 2014. I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis. *arXiv preprint arXiv:1404.7431* (2014).
[24] Mengchen Li, Yuanjun Chen, Linzhang Wang, and Guoqing Xu. 2013. Dynamically validating static memory leak warnings. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 112–122.
[25] Peng Li and Baojiang Cui. 2010. A comparative study on software vulnerability static analysis techniques and tools. In *Information Theory and Information Security (ICITIS), 2010 IEEE International Conference on*. IEEE, 521–524.
[26] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. 2013. Steering symbolic execution to less traveled paths. In *ACM SigPlan Notices*, Vol. 48. ACM, 19–32.
[27] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. 2015. TaintPipe: Pipelined Symbolic Taint Analysis.. In *USENIX Security*, Vol. 15.
[28] Krerk Piromsopa and Richard J Enbody. 2011. Survey of Protections from Buffer-Overflow Attacks. *Engineering Journal* 15, 2 (2011), 31–52.
[29] Barbara G Ryder. 1979. Constructing the call graph of a program. *IEEE Transactions on Software Engineering* 3 (1979), 216–226.
[30] 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.
[31] Koushik Sen. 2007. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 571–572.
[32] Hyunmin Seo and Sunghun Kim. 2014. How we get there: A context-guided search strategy in concolic testing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 413–424.
[33] Michael Sipser. 2006. *Introduction to the Theory of Computation*. Vol. 2. Thomson Course Technology Boston.
[34] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. 2011. Rolecast: finding missing security checks when you do not know what checks are. *ACM SIGPLAN Notices* 46, 10 (2011), 1069–1084.
[35] James Stanier and Des Watson. 2013. Intermediate representations in imperative compilers: A survey. *ACM Computing Surveys (CSUR)* 45, 3 (2013), 26.
[36] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium*.
[37] Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering* 40, 2 (2014), 107–122.
[38] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: brute force vulnerability discovery*. Pearson Education.
[39] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 48–62.
[40] David Wagner, Jeffrey S Foster, Eric A Brewer, and Alexander Aiken. 2000. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities.. In *NDSS*. 2000–02.
[41] Mingxin Wang and Jingfu Zhao. 2015. A free boundary problem for the predator–prey model with double free boundaries. *Journal of Dynamics and Differential Equations* (2015), 1–23.
[42] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. 2009. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution.. In *NDSS*. Citeseer.
[43] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. 2013. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 499–510.