Defining Code-injection Attacks

Donald Ray   Jay Ligatti

Department of Computer Science and Engineering
University of South Florida
`{dray3,ligatti}@cse.usf.edu`

Abstract

This paper shows that existing definitions of code-injection attacks (e.g., SQL-injection attacks) are flawed. The flaws make it possible for attackers to circumvent existing mechanisms, by supplying code-injecting inputs that are not recognized as such. The flaws also make it possible for benign inputs to be treated as attacks. After describing these flaws in conventional definitions of code-injection attacks, this paper proposes a new definition, which is based on whether the symbols input to an application get used as (normal-form) values in the application's output. Because values are already fully evaluated, they cannot be considered "code" when injected. This simple new definition of code-injection attacks avoids the problems of existing definitions, improves our understanding of how and when such attacks occur, and enables us to evaluate the effectiveness of mechanisms for mitigating such attacks.

Categories and Subject Descriptors C.2.0 [*Computer Communication Networks*]: General—Security and protection; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Syntax

General Terms Security, languages, algorithms

1. Introduction

As the popularity of web applications has increased, so have reports of attacks against them [20–22]. The most commonly reported type of attack involves injecting code into a program output by an application, as in SQL-injection attacks [22].

Standard examples of code-injection attacks include an attacker entering the following strings as input to an application:

- `' OR 1=1 --`, to make the application output the program `SELECT balance FROM acct WHERE password='' OR 1=1 --'` (underlined symbols are those the attacker has injected into the output program). This SQL program always returns the balance(s) from the `acct` table, even though an empty-string password is supplied, because: (1) the `1=1` subexpression is true, making the entire `WHERE` clause true, and (2) the `--` command comments out the final apostrophe to make the program syntactically valid. In this case, the attacker has injected the code symbols `OR`, `=`, and `--` into the output program.
- `exit()`, to make the application output the program `SELECT balance FROM acct WHERE pin=exit()`. In this case, the attacker has injected the code symbols `exit()` into the output program to mount a denial-of-service attack against the remote database.

These attacks are commonly referred to as "code-injection attacks" or "command-injection attacks" (CIAs, or just IAs), but here we use the more specific term "code-injection attacks on outputs" (CIAOs) to distinguish these attacks, which require code to be injected into an application's *output*, from more general notions of CIAs, which require code to be injected only into *memory used by* an application (e.g., [39]).

Much research has focused on mechanisms for mitigating CIAOs, and a few efforts have been made to define CIAOs rigorously [3, 34, 37]. However, as Section 2 discusses, existing definitions are based on the flawed belief that CIAOs occur whenever an application's input alters the syntactic structure of its output program. Incorrectly conflating CIAOs with changes to syntactic structures causes existing definitions to be neither sound nor complete: some CIAOs are not recognized as CIAOs (false negatives), and some non-CIAOs are recognized as CIAOs (false positives). The false negatives allow attackers to circumvent existing mechanisms for preventing CIAOs.

Without a satisfactory definition of CIAOs, we can't effectively develop or analyze mechanisms for mitigating them; preventing CIAOs in general requires understanding exactly how and when they succeed. As Sun Tzu's *The Art of War* famously expresses it, "If you know yourself but not the enemy, for every victory gained you will also suffer a defeat" [36].

This is a paper about "knowing the enemy"; it defines precise circumstances under which CIAOs can be said to occur. Defining CIAOs requires two subdefinitions: first we must define which symbols in applications' output programs constitute *code*; second we must define when symbols have been *injected* into output programs. The primary contribution and novelty of this paper lies in the first of these subdefinitions; the second subdefinition can be approached with well-known (but difficult-to-implement) techniques of taint tracking. For the first subdefinition, this paper defines "code" significantly differently from previous work—instead of code being parse-tree-changing symbols, code here refers to symbols that do not form *values* (i.e., fully evaluated terms). This paper argues that CIAOs occur when at least one symbol injected into an output program is used outside of a value.

To keep the definitions general, this paper tries to abstract as much as possible from underlying languages, programs, and system architectures. Although the definitions will require a few technical assumptions about the languages of applications' output programs (such as that they have a well-defined set of normal-form values), this paper's definitions are not limited to SQL or other popular programming languages; the new definition of CIAOs applies equally well to other code-injection attacks (e.g., LDAP-injection, HTML/script-injection (XSS), and shell-injection attacks). Similarly, the only assumption we make of applications is that tainted inputs can be correctly tracked through them, so we know which symbols in their outputs have been injected.

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.

1.1 Summary of Contributions and Roadmap

This paper demonstrates problems in the conventional definition of CIAOs (in Section 2). The problems make existing CIAO-mitigating mechanisms neither sound nor complete—some CIAOs are considered benign, while some non-CIAOs are considered attacks. After discussing previous work, the paper presents (in Sections 3–4) a new definition of CIAOs that avoids these problems.

Ultimately, a definition of CIAOs has two important high-level uses. First, a definition of CIAOs enables us to precisely determine whether applications exhibit CIAOs. We put the new definition to this first use by illustrating the new definition's improved ability to determine whether applications exhibit CIAOs (primarily in Section 3.3). Second, a definition of CIAOs enables us to analyze the effectiveness of mechanisms at mitigating CIAOs. We put the new definition to this second use by analyzing the effectiveness of several classes of mechanisms for detecting CIAOs (in Section 5).

More specifically, several properties of CIAOs become apparent by considering the new definitions:

- Defining CIAOs as occurring when nonvalue symbols get injected into output programs improves our ability to recognize attacks. We illustrate the improvements in an idealized version of SQL called SQL$^\circ$, "SQL Diminished" (Sections 3–4.1).
- CIAOs can be classified as copy-based or data-dependency-based, depending on how applications propagate untrusted (tainted) inputs into output programs (Section 4.2.1).
- A class of attacks related to CIAOs exists, which we call code-*interference* attacks. The definition of these attacks takes into account control dependencies ignored by taint-tracking mechanisms (Section 4.2.2).
- Surprisingly, every application that always copies some untrusted input verbatim into an (SQL$^\circ$) output program is vulnerable to CIAOs (Section 5.1). This result implies that sound static mechanisms for detecting CIAOs must disallow all such applications, conservatively ruling out a large class of applications in practice. The proof of this result (in Section 5.1) is constructive; the proof defines inputs that will successfully attack any application that verbatim copies some untrusted input into the output program. Although the proof is tailored to SQL$^\circ$, the proof techniques are general and applicable to other languages.
- Similarly, applications that always copy some untrusted input verbatim into an (SQL$^\circ$) output program are vulnerable to code-interference attacks (Section 5.1).
- Neither static nor black-box analysis of applications can precisely detect CIAOs. (Section 5.2)
- Precisely detecting CIAOs requires white-box, runtime-monitoring mechanisms. Under reasonable assumptions, such mechanisms can detect CIAOs in output programs of size n in $O(n)$ time and space. However, there are obstacles that make it difficult to implement such mechanisms in practice. (Sections 5.3–5.4).

After presenting these results in Sections 2–5, Section 6 concludes.

2. Related Work

Conventionally, CIAOs are considered to occur whenever an application's input alters the intended syntactic structure of its output program. Bisht, Madhusudan, and Venkatakrishnan call this "a well-agreed principle in other works on detecting SQL injection" [3]. Indeed, this definition has appeared in many documents: [2–5, 9, 10, 13, 17, 27, 31, 34, 37]. Although a few papers define CIAOs in other ways (e.g., CIAOs occur exactly when keywords or operators get injected, including apostrophes used to form string values in SQL [8, 25], or when injected strings span multiple tokens [38]), the conventional definition dominates the literature.

However, the conventional definition of CIAOs has inherent problems: some CIAOs do not alter the syntactic structures of output programs, while some non-CIAOs do. To illustrate these problems, Sections 2.1 and 2.2 discuss the conventional definitions of CIAOs used by SqlCheck [34, 37] and Candid [2, 3]. As far as we're aware, these are the only existing formal definitions of CIAOs.

2.1 CIAOs in SqlCheck

SqlCheck considers the intended syntactic structure of an output program to be any parse tree in which each injected input is the complete derivation of one terminal or nonterminal. For example, parsing the output program `SELECT balance FROM acct WHERE password='' OR 1=1 --'` produces a parse tree in which the injected symbols `' OR 1=1 --` are not the complete sequence of leaves for a single terminal or nonterminal ancestor; SqlCheck therefore recognizes this CIAO.

However, some of what SqlCheck considers intended (i.e., non-attack) structures are actually attacks. For example, parsing the output program `SELECT balance FROM acct WHERE pin=exit()` produces a tree in which the input symbols `exit()` are the complete sequence of leaves for a single nonterminal (function-call[1]) ancestor. Hence, SqlCheck does not recognize this CIAO as an attack. Similarly, an output program of the form `...WHERE flag=1000>GLOBAL` wouldn't be recognized as an attack, despite the injection of a greater-than operator (which may allow an attacker to efficiently extract the value of the `GLOBAL` variable, by performing a binary search over its range). Although SqlCheck allows policy engineers to specify a set of terminal and nonterminal ancestors that inputs may derive from—so engineers could disallow inputs derived as function-call and comparison expressions—it's unclear how an engineer would know exactly which ancestors to allow derivations from. Moreover, engineers may wish to sometimes allow, and sometimes disallow, inputs to derive from particular terminals and nonterminals (as illustrated in Section 3.3), which is impossible in SqlCheck.

Conversely, some of what SqlCheck considers unintended (i.e., attack) structures are actually not attacks. For example, an application might input two strings, a file name `f` and a file extension `e`, and concatenate them to generate the program `SELECT * FROM properties WHERE filename='f.e'`. Although the user has injected no code, SqlCheck flags this output as a CIAO because the user's inputs are not complete sequences of leaves for a single terminal or nonterminal ancestor. In this case, the immediate ancestor of the user's inputs would (assuming a typical grammar) be a string literal, but neither of the user's inputs form a complete string literal—they're missing the dot and single-quote symbols.

The Candid papers describe other, lower-level problems with SqlCheck's definitions [2, 3].

2.2 CIAOs in Candid

Candid considers the intended syntactic structure of an output program, generated by running application A on input I, to be whatever syntactic structure is present in the output of A on input *VR*(I). Here *VR* is a (valid representation) function that converts any input I into an input I' known to (1) be valid (i.e., non-CIAO-inducing) and (2) cause A to follow the same control-flow path as it would on input I. Candid begins by assuming this *VR* function exists, while acknowledging that it does not; in this basic case,

[1] All major SQL implementations we are familiar with allow statements to call functions, including administrator-defined functions [19, 28, 29].

CANDID defines a CIAO to occur when A's output on input I has a different syntactic structure from A's output on input *VR*(I).

Besides the nonexistence of function *VR*, there are some problems with this definition of CIAOs. First, the definition is circular; CIAOs are defined in terms of *VR*, which itself is assumed to output non-CIAO-inducing inputs (i.e., the definition of CIAOs relies on the definition of *VR*, which relies on the definition of CIAOs). Second, the definition assumes that multiple valid syntactic structures cannot exist. For example, suppose *VR*(`','`)=`aaa` and application A on input `','` outputs `SELECT * FROM t WHERE name IN ('a','b')`, while A on input `aaa` executes in the same way to output `SELECT * FROM t WHERE name IN ('aaaab')`. Both of these outputs are valid SQL programs, yet the programs have different syntactic structures (a two-element list versus a single-element list), and neither exhibits a CIAO (in no case has *code* been injected; only values, which take no steps dynamically, have been injected). CANDID would classify the non-CIAO input of `','` as an attack in this case.

To deal with the nonexistence of function *VR*, CANDID attempts to approximate *VR* by defining *VR*(I) to be 1 when I is an integer and $a^{|I|}$ when I is a string (where $a^{|I|}$ is a sequence of a's having the same length as I). Supplying a concrete definition of *VR* resolves the circularity problem in CANDID's basic definition of CIAOs, but it doesn't resolve the second problem described in the previous paragraph (that multiple valid syntactic structures may exist).

Moreover, CANDID's approximation of *VR* creates new problems:

- The approximation incorrectly assumes a string of a's or a 1 could never be attack inputs. An application could inject an input a or 1 into an output program as part of a function call, field selection, or even keyword (e.g., and), all of which could be CIAOs. For example, suppose an application outputs a constant string, echoes its input, and then outputs parentheses; on input `exit` it outputs the program `...pin=exit()`. CANDID would not recognize this CIAO because the application outputs `...pin=aaaa()` on input `aaaa`, which has the same syntactic structure as the `...pin=exit()` output. The problem here is that `aaaa` is actually an attack input for this application.
- The approximation may also cause benign inputs to be detected as attacks. For example, suppose an application outputs `SELECT * FROM t WHERE flag=TRUE` on input `TRUE`, and follows the same control-flow path to output `SELECT * FROM t WHERE flag=aaaa` on input *VR*(`TRUE`)=`aaaa`. Because these two output programs have different syntactic structures (a boolean literal versus a variable identifier), CANDID would flag the input `TRUE` as an attack, even though the user has injected no code.
- The approximation can also break applications, as discussed in [3]. To illustrate this problem, let's consider the application `if(input<2) then restart() else output(1/(input-1))`. CANDID cannot in general operate on this application because it evaluates applications on both actual (I) and candidate (*VR*(I)) inputs, while following the control-flow path required to evaluate the actual input. In this case, whenever the application's actual input is greater than one, CANDID will try to evaluate `1/(input-1)` on the candidate input 1, which causes the application to halt with a divide-by-zero error, despite there being no errors in CANDID's absence.

It could be argued that the example applications in the bullets above would be uncommon in practice. But limiting the definition of CIAOs to common applications obligates us to define what makes an application common, so we can test whether a given application is "common" enough for the definition of CIAOs to apply. Even then, one couldn't say anything about CIAOs in uncommon applications.

Related-work Summary CIAOs cannot be said to occur when an application's output program has an altered syntactic structure.

- CIAOs can occur without altering the syntactic structure of output programs (e.g., by injecting `exit()` or `1000>GLOBAL` in SQLCHECK, or `exit` in CANDID).
- Non-CIAOs can occur while altering the syntactic structure of output programs (e.g., by injecting file name `f` and extension `e` in SQLCHECK, or `TRUE` in CANDID).

3. Partitioning Program Symbols into Code and Noncode

This section begins building a new definition of CIAOs. Because CIAOs occur when *code* symbols get injected into output programs, the question of which output-program symbols constitute code is key to defining CIAOs. This section addresses that question and defines how to separate code from noncode. (The discussion is limited to the context of CIAOs; in other contexts it makes sense to consider entire output programs as "code".)

3.1 Overview of Partitioning Technique

Let's begin by defining what is not code, rather than what is. This paper considers noncode to be the closed *values* in a programming language. Values are valid but operationally irreducible terms (i.e., normal forms) [16, 32]. Values can be thought of as the "fully evaluated" computations in a programming language, typically including standalone string and integer literals, pointers, objects, lists and tuples of other values, etc. Values are closed when they contain no free variables; open values have free variables (e.g., a tuple value like `(4,x)` and standalone variables are open values).

Closed values are fully evaluated, dynamically passive constructs, which by themselves cause no dynamic computation to occur. On the other hand, because nonvalues and open values are not part of these passive terms, they are used to help specify dynamically active computation and therefore constitute code (in the case of open values, the dynamic activity specified by a free variable is a substitution operation, which substitutes a term for the free variable at runtime). Injecting symbols that only form closed values into an output program therefore cannot be considered a CIAO—only irreducible, dynamically passive terms (i.e., "noncode") will have been introduced. CIAOs occur when untrusted inputs get used outside of closed values in output programs.

3.2 Formal Assumptions and Definitions

An application vulnerable to CIAOs outputs programs in some language L (e.g., SQL) having finite concrete-syntax alphabet Σ_L (e.g., the set of printable ASCII characters). An output program, which we call an L-program, is a finite sequence of Σ_L symbols that form an element of L.

Definition 1. *For all languages L with alphabet Σ_L (i.e., $L \subseteq \Sigma_L^*$), an L-program is an element of L.*

Additional definitions will rely on the following assumptions and notational conventions:

- The length of program p is denoted as $|p|$ (so when $p = \sigma_1\sigma_2..\sigma_n$, where each σ is a program symbol in Σ_L, we have $|p| = n$).
- The i^{th} symbol in program p is denoted as $p[i]$.
- Well-defined functions exist for computing free variables in all output-program languages under consideration. Function

$FV_L(p, l, h)$ takes an L-program $p = \sigma_1\sigma_2..\sigma_n$, a low symbol number $l \in \{1..n\}$, and a high symbol number $h \in \{l..n\}$ and returns the set of variables that are free in the shortest term in p that contains all of the symbols $\sigma_l..\sigma_h$.

- Well-defined functions also exist for testing whether terms are values in all output-program languages under consideration. Predicate $Val_L(p, l, h)$ is true iff the shortest term that contains the l^{th} to h^{th} symbols in L-program p is a value.

When the output language L is clear from context, we'll omit it as a subscript on FV_L and Val_L functions.

We now formalize Section 3.1's intuition of noncode program symbols. We use the predicate NCV to indicate whether symbols in an L-program form a *noncode value*. NCV is true for an L-program p and low and high program-symbol numbers l and h iff the shortest term containing the l^{th} to h^{th} symbols in p is a closed value.

Definition 2. *For all languages L, predicate $NCV(p, l, h)$ over $L \times \{1..|p|\} \times \{l..|p|\}$ is true iff $FV(p, l, h) = \emptyset$ and $Val(p, l, h)$.*

Code symbols are those that cannot possibly be part of any noncode value. When $p[i]$ is code (where p is an output program), we write $Code(p, i)$.

Definition 3. *For all L-programs $p = \sigma_1\sigma_2..\sigma_n$ and position numbers $i \in \{1..|p|\}$, $Code(p, i)$ is true iff for all low and high symbol-position numbers $l \in \{1..i\}$ and $h \in \{i..|p|\}$, $\neg NCV(p, l, h)$.*

3.3 Example Separation of Code from Noncode

The remainder of this section illustrates Definition 3 in the context of SQL° (SQL Diminished), an idealized SQL-style language inspired by the MSDN SQL Minimum Grammar [18]. Figure 1 presents SQL°'s syntax, which makes several assumptions:

- Full SQL° programs are valid `statement`s.
- Operators in SQL° have standard precedence and associativity.
- A set of (administrator-defined and/or standard-library) parameterless functions exists, and SQL° expressions (i.e., `expr`s) can invoke these functions with the `ID()` syntax (where `ID` is an identifier, in this case a function name). Such function calls are possible in typical SQL implementations [14, 19, 28, 29].
- Similarly, a set of (administrator-defined and/or standard) variables exists, and variable identifiers are valid SQL° expressions.
- Comments in SQL° begin with `--` and continue to the first newline.
- String literals in SQL° have the same escape sequence as string literals in full SQL (i.e., a double apostrophe represents a single apostrophe). Also as in full SQL, apostrophe directions are irrelevant in SQL°, though we use directed apostrophes in this paper for clarity.

Values in SQL° are the last six terms listed in Figure 1 for category `expr` (i.e., from `INT_LITERAL` to `NULL`). Intuitively, the values in a typed programming language are normally all the fully evaluated terms of each type in the language. SQL° has types for integers (`INT`), strings having a given size (`CHAR(INT_LITERAL)`), booleans (`BOOL`), and floats having a given precision (`FLOAT(INT_LITERAL)`), so its values are the fully evaluated terms of each of these types—including integer literals, string literals, the true and false keywords, and floating-point literals. Finally, `NULL` is a fully evaluated term of any type, also a value.

Given that values in SQL° are exactly the last six terms listed as `expr`s in Figure 1, Definitions 2 and 3 imply that $Code(p, i)$ holds iff, after parsing program p, $p[i]$ is not part of a nonterminal categorized as one of these six kinds of `expr`s. Noncode symbols are those in closed values; all others are code. This definition also works when partitioning whitespace and comment symbols: no symbol involved in whitespace or comments can possibly be within a value (all values in SQL° are single tokens), so whitespace and comment symbols are code.[2]

```
   statement  ::=  CREATE TABLE ID ( id_type_list )
                |  DELETE FROM ID w_option
                |  DROP TABLE ID
                |  INSERT INTO ID vals
                |  SELECT s_list FROM ID w_option

id_type_list  ::=  ID type | id_type_list , ID type

        type  ::=  INT | CHAR ( INT_LITERAL ) | BOOL
                |  FLOAT ( INT_LITERAL )

    w_option  ::=  ε | WHERE expr

        expr  ::=  expr op expr | NOT expr | ( expr )
                |  expr IS NULL | ID | ID ( )
                |  INT_LITERAL | STR_LITERAL | TRUE
                |  FALSE | FLOAT_LITERAL | NULL

          op  ::=  OR | AND | < | > | = | + | ∗

        vals  ::=  VALUES ( e_list )
                |  SELECT s_list FROM ID w_option

      e_list  ::=  expr | e_list , expr

      s_list  ::=  ∗ | i_list

      i_list  ::=  ID | i_list , ID
```

Figure 1. Syntax of SQL°.

A few observations about this definition of code in SQL°:

- Parsing is necessary to determine whether a symbol is code. For example, an integer literal is code when used in a `type` specification, but noncode when used as an expression.
- Conventional definitions of CIAOs are incompatible with the definition of code in SQL°. For example, there exists no set of terminals and nonterminals in Figure 1 that exactly derive noncode symbols (`expr`s may contain code, and even `INT_LITERAL`s may be code depending on the context); hence, our definition of code is inexpressible with SQLCHECK [34].
- Code and noncode can't be partitioned by considering noncode to be literals. Some literals are code (e.g., an integer in a `type`) and other are not (e.g., an integer expression). Although all nonliterals (e.g., a `CREATE` keyword) are code in SQL°, languages with more sophisticated values (e.g., lists) may have nonliteral, noncode symbols (e.g., commas between elements of a list value).
- Code and noncode also can't be partitioned by considering code to be keywords and operators. Some keywords are code (e.g., `CREATE`) and others are not (e.g., `TRUE`). Some symbols that are neither keywords nor operators are code (e.g., function-name `ID`s) and others are not (e.g., literals).

Next, let's consider several example output programs, beginning with the examples from Sections 2.1 and 2.2, to see how the new definitions partition injected symbols.

[2] It may also be reasonable to partition lexer-removed symbols (typically whitespace and comments) into code and noncode in other ways. For example, one might consider lexer-removed symbols code iff their existence affects the sequence of tokens in the output program.

	1	2	3	4	5	6	7	8	9	10	11
This paper	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes	Yes	No
SQLCHECK [34]	Yes	No	No	Yes	No	No	No	No	No	No	No
CANDID [3]	Yes	Yes	Yes	No	No	No	Yes	No	No	No	Yes
WASP [8] and Nguyen-Tuong et. al. [25]	Yes	Yes	Yes	No	No	No	No	No	No	No	Yes
Xu et. al. [38]	Yes	Yes	Yes	No	No	No	No	No	No	No	No

Figure 2. A comparison of definitions for partitioning code and noncode. Column numbers refer to the example output programs enumerated in Section 3.3, row names indicate partitioning techniques, and cells specify whether any of the underlined symbols are considered code.

1. `SELECT balance FROM acct WHERE password='' OR 1=1 --'` The injected `OR`, `=`, and `--` (and spaces) are code, so a CIAO has occurred.
2. `SELECT balance FROM acct WHERE pin= exit()` All the injected symbols are code, so a CIAO has occurred.
3. `...WHERE flag=1000>GLOBAL` The injected `>` is code, so a CIAO has occurred.
4. `SELECT * FROM properties WHERE filename='f.e'` No injected symbols are code, so a CIAO has not occurred.
5. `...pin=exit()` All the injected symbols are code, so a CIAO has occurred.
6. `...pin=aaaa()` Again, all the injected symbols are code, so a CIAO has occurred.
7. `SELECT * FROM t WHERE flag=TRUE` No injected symbols are code, so a CIAO has not occurred.
8. `SELECT * FROM t WHERE flag=aaaa` An open expression (which causes a substitution operation to be performed at runtime) was injected, so a CIAO has occurred.
9. `SELECT * FROM t WHERE password=password` Again, an open expression (which causes a substitution operation to be performed at runtime) was injected, so a CIAO has occurred.
10. `CREATE TABLE t (name CHAR(40))` All the injected symbols are code, so a CIAO has occurred.
11. `SELECT * FROM t WHERE name='x'` No injected symbols are code, so a CIAO has not occurred.

In all of these cases, the partitioning avoids the problems with conventional CIAO definitions described in Section 2 and matches our intuition about which program symbols are code (and consequently cause a CIAO if injected).

Figure 2 compares this paper's partitioning of the example output programs enumerated above with the partitionings used in previous work. The only scenarios in which we believe previous definitions would be favored over this paper's definitions are when the assumptions made by this paper's definitions cannot be satisfied easily, that is, when it's difficult to define the set of closed values in the output-program language.

4. Definition of CIAOs

Defining CIAOs requires subdefinitions of *code* and *injection*. At this point code has been defined; it is time to consider what it means for an attacker to inject symbols into an output program. Intuitively, injected symbols are the ones that propagate unmodified from an untrusted input source to the output program. A CIAO occurs when at least one untrusted input symbol propagates into, and gets used as code in, an output program.

To know when input symbols have propagated, possibly through copy operations, to output programs, one could taint all untrusted inputs to applications and have those applications transparently propagate taints through copy operations (Section 4.2 will consider propagating taints through other operations as well). Then output programs could be tested to determine whether any of their tainted symbols are used as code. Tracking taints to determine which output-program symbols derive from untrusted inputs is a well-studied technique (e.g., [6, 8, 25, 31, 38]).

4.1 Formal Assumptions and Definitions

As in earlier sections, underlines will represent tainted symbols (i.e., those injected from untrusted sources). As a technicality, if some element of Σ is already underlined then all underlines in this paper may need to be replaced with some other annotation not present on any Σ symbol. Then, for all languages L with alphabet Σ, let $\underline{L}$ denote the same language but with alphabet $\underline{\Sigma}$, where $\underline{\Sigma}$ contains tainted and untainted versions of every symbol in Σ. Thus, the *tainted output language* $\underline{L}$ contains exactly those programs in L, except that programs in $\underline{L}$ can have symbols tainted in any way. The following three definitions formalize these ideas.

Definition 4. *For all alphabets Σ, the* tainted-symbol alphabet $\underline{\Sigma}$ *is:* $\{\sigma \mid \sigma \in \Sigma \vee (\exists \sigma' \in \Sigma : \sigma = \underline{\sigma'})\}$.

Definition 5. *For all alphabets Σ and symbols $\sigma \in \underline{\Sigma}$, the predicate $tainted(\sigma)$ is true iff $\sigma \notin \Sigma$.*

Definition 6. *For all languages L with alphabet Σ, the* tainted output language $\underline{L}$ *with alphabet $\underline{\Sigma}$ is:*

$$\{\sigma_1..\sigma_n \mid \exists \sigma'_1..\sigma'_n \in L : \forall i \in \{1..n\} : (\sigma_i{=}\sigma'_i \vee \sigma_i{=}\underline{\sigma'_i})\}$$

Given a regular, non-taint-tracking application, which outputs L-programs, a *taint-tracking application*, which outputs $\underline{L}$-programs, is constructed by ensuring all the following.

1. All symbols input to the application from untrusted sources are marked tainted.
2. Taints propagate through all operations that copy or output symbols.
3. Besides inputting symbols from untrusted sources and copying and outputting already tainted symbols, there are no other ways to introduce tainted symbols.
4. Taints are invisible to the application; they have no effect on its execution.

The only assumption this paper makes of applications is that they can be reasoned about as taint-tracking applications obeying these four rules.

At last, CIAOs can be defined as occurring whenever an injected (i.e., tainted) symbol in an application's output is used as code.

Definition 7. *A* CIAO *occurs exactly when a taint-tracking application outputs $\underline{L}$-program $p = \sigma_1..\sigma_n$ such that $\exists i \in \{1..n\}$: $(tainted(\sigma_i) \wedge Code(p, i))$.*

4.2 Discussion of the CIAO Definition

There are several points of discussion related to Definition 7.

Syntactic versus Semantic Analysis Contrary to previous work [3, 34], Definition 7 does not limit CIAO detection to syntactic analysis. Although testing whether sequences of program symbols denote closed values typically requires only syntactic analysis (e.g., values are defined syntactically for SQL° in Section 3.3), such testing could conceivably require stronger-than-syntactic analysis. For example, semantic analysis may be required to determine whether the output programs `date:=1/1/11` and `balance:=1/1/11` exhibit CIAOs in output-program languages with slashes used in both date-literal and integer-division expressions.

Expected (Nonmalicious) CIAOs Although CIAOs often constitute malicious use of an application, some application programmers expect CIAOs to occur and don't consider them malicious. For example:

- A translator between programming languages may input an expression like `x+y` and output a program containing the same expression or some code like `r1:=r1+r2`, with the nonvalue `+` symbol having been injected. This is not a problem, and authors of programming-language translators would typically not consider CIAOs on their translators harmful.
- Tools like phpMyAdmin provide interfaces for remote users to enter MySQL programs and then have those programs output for other systems to execute [30].
- Applications may check inputs before injecting them as code in output programs, such as the application `if(input= 'safeFunction') then output(input+'()') else raise badNameExn`, or the application `if(input.matches ('Math.*')) then output(input+'()') else raise badNameExn`. Programmers of these applications may not consider CIAOs of checked inputs to be malicious (though it may nonetheless be desirable to detect CIAOs in such programs, for example, to prevent the latter application from outputing `Math.pi+exit()`).

We believe that whether an act is "malicious" or an "attack" or against a programmer's "expectations" or "intentions" is subjective. The only artifact we can examine is the programmer's code, but that code may not capture the programmer's intentions. Definition 7 therefore does not depend on subjective factors like programmers' intentions; CIAOs are defined as occurring whenever an application injects untrusted input into the code of an output program, regardless of whether the application programmer would consider the CIAOs malicious.

To make an analogy to memory safety, there are mechanisms to prevent memory-safety violations, e.g., type checkers. However, some memory-safety violations are not harmful and may be fully intended by programmers. For example, a programmer with knowledge of how integers and floats are encoded may find that writing an arbitrary float value to integer-type memory does exactly what s/he wants very efficiently. As another example, one of the difficulties encountered by "Safe C" projects is that some memory-safety violations are actually intentional and clever optimizations [7, 11, 23].

Definitions of memory-safety violations don't (as far as we're aware) take into account programmer intentions; mechanisms for preventing memory-safety violations disregard programmer intentions and prevent all memory-safety violations, regardless of whether a programmer considers some particular violation malicious. Analogously, Definition 7, unlike the conventional definitions of CIAOs used by previous work, sidesteps the subjective questions of whether output programs are intended or malicious. Definition 7 just focuses on detecting whether code has been injected into output programs.

Non-CIAO injection attacks Some injection attacks on output programs are not *code*-injection attacks on output programs. For instance, consider the output program `SELECT balance FROM acct WHERE password = TRUE`. Here, a type error will occur (assuming that `password` is not of boolean type), potentially causing unexpected failures. Although this output program contains symbols that may have been injected with malicious intent, those symbols are part of a closed value and are therefore not used as code. Because code has not been injected, the output program does not exhibit a CIAO according to Definition 7.

Function values Functions are first-class values in many languages, and it may seem strange to allow arbitrary closed function values to be injected into output programs. However, a function value is dynamically passive; a function value only activates when operated upon, by applying the function. Hence, injecting a function value does not constitute a CIAO, but injecting a function application does (assuming the injected application is not within some other closed value, such as an outer lambda term).

Alternate-encoding and second-order attacks Definition 7 has no problem with "alternate-encoding attacks". Alternate encodings allow attackers to mask injected code, for example, by inputting `exec(char(0x73687574646f776e))` instead of a direct `SHUTDOWN` command [8]. Definition 7 detects such attacks because the injected function calls are recognized as code. Definition 7 also detects "second-order injection attacks" (where an attacker stores some code in a database that an application later retrieves and injects into its output [1, 26]), as long as the database input to the application is considered untrusted (or, as a more precise alternative, the database could store flags indicating which of its entries' symbols are tainted).

Defining "injection" Finally, Definition 7 interprets "injection" as meaning that symbols have been directly copied from input to output. For example, loading a tainted symbol from memory into a register would taint that register's value, but adding two tainted integers involves no direct copying and therefore produces an untainted result. Thus, Definition 7 does not consider CIAOs to occur when applications output programs whose code symbols are "massaged" versions of untrusted inputs—the massaging (i.e., noncopy manipulation) prevents the input symbols from being considered injected. Intuitively, an application like `output(input()+1)` may input a `1` from an untrusted user and then output the program `2`. In this case it seems inaccurate to say that the user "injected" the `2`, given that the user never entered a `2`, and the application produced the `2` by actively transforming its input. If anything, the application and user have collaborated to produce the `2` that got output. One could consider this example demonstrative of a more general class of attacks: data-dependency-based CIAOs.

4.2.1 Data-dependency-based CIAOs

Following this train of thought leads us to define *data-dependency-based CIAOs* in exactly the same way as regular CIAOs (which henceforth will also be called *copy-based CIAOs*), except that for data-dependency-based CIAOs we broaden taint propagation to occur on all data operations, not just copies and outputs. That is, for any data dependency in which the value of a symbol σ depends on the value of at least one tainted symbol, σ must also be tainted. In the example above, we would taint the `2` produced by adding a tainted `1` with an untainted `1`. As a better example, consider the application `output(toUpper(input())+'()')`, which outputs `EXIT()` after inputting `exit`. Definition 7 does not consider this `exit` input to be a copy-based CIAO because with copy-based tainting, the output is just `EXIT()`, with no symbols tainted/underlined. However, the `exit` input is a data-dependency-based CIAO because with data-dependency-based tainting, the output is EXIT(). Note that every copy-based CIAO is also a data-dependency-based CIAO.

In many cases, such as the all-caps-function-name application above, it may be helpful to detect and prevent data-dependency-based CIAOs. In other cases, data-dependency-based-CIAOs may be expected, so system administrators may not find it helpful for them to be caught and flagged (similar to expected copy-based CIAOs, discussed above). For example, an application for managing online courses might hash an input student number to obtain a discussion-group number g and then output a program like `SELECT numPosts FROM group_`g` where threadNum=4`. Assuming g is obtained through noncopy operations on the untrusted student-number input, this application exhibits a non-copy-based, data-dependency-based CIAO. But the application programmers and system administrators would likely not consider this data-dependency-based CIAO malicious.

4.2.2 Code-interference Attacks

Broadening taint propagation further, one might consider taints to propagate even through control dependencies. To illustrate, let's consider the following application, which is semantically equivalent to the input-echoing application `output(input())` and performs what [3] calls a "conditional copy".

```
while(there are more input symbols) {
  switch(next input symbol) {
    case 'a' : output('a'); break;
    case 'b' : output('b'); break;
    ...
  }
}
```

This switch-based application is *invulnerable* to (copy-based and data-dependency-based) CIAOs because there are no data dependencies between input and output symbols—every symbol output is a constant hardcoded into the application source code. On the other hand, the semantically equivalent input-echoing application *is* vulnerable to (copy-based and data-dependency-based) CIAOs because it directly copies input symbols into the output. These are reasonable consequences of only dealing with code-*injection* attacks; CIAOs only occur when code symbols in output programs directly depend on untrusted input.

Still, it may be desirable to prevent applications from behaving as the switch-based application above does, and more generally, to prevent untrusted inputs from interfering at all (even indirectly, through control dependencies) with the code symbols an application outputs. To do so, we propose studying *CIntAOs*—code-*interference* attacks on outputs. The switch-based application above *is* vulnerable to CIntAOs because its input can interfere with the code symbols that get output.

An application is vulnerable to CIntAOs whenever inputs differing in untrusted symbols can cause the application to output programs differing in code symbols. In other words, applications invulnerable to CIntAOs must, when given the same trusted inputs, always output programs containing the same code symbols.

Definition 8. *A CIntAO occurs exactly when:*

1. *Application A, on trusted and untrusted input sequences $T{\in}\Sigma^*$ and $U{\in}\Sigma^*$, outputs L-program p.*
2. *There exists another untrusted input sequence $U'{\in}\Sigma^*$ such that:*
   (a) *On T and U', A outputs L-program p'.*
   (b) *The subsequence of code symbols in p is not equal to the subsequence of code symbols in p'.*

The switch-based application above is invulnerable to CIAOs but vulnerable to CIntAOs. It also is possible for applications to be invulnerable to CIntAOs but vulnerable to CIAOs. For example, the application `if input='1+1' then output(input) else output('1+1')` exhibits a CIAO on input '1+1' but cannot exhibit a CIntAO because there is no way to change the subsequence of code symbols in the output program by changing the untrusted input.

Although it may sometimes be desirable to detect CIntAOs, the strictness with which they're defined causes many reasonable applications, which are free of CIAOs, to exhibit CIntAOs. For example, an application could accept some untrusted input indicating which currency to output an account balance in; if the desired currency is not the default, the application might output some code to multiply the fetched balance by a conversion rate. This application exhibits neither data-dependency-based nor copy-based CIAOs because the code symbols it outputs (e.g., the multiplication symbol) are not data-dependent on the input currency. However, this application does exhibit CIntAOs because the input currency affects (through a control dependency) the code that gets output (i.e., whether or not a multiplication gets included in the output program). Hence, this example application illustrates that CIntAOs, like copy- and data-dependency-based CIAOs, may be expected and not considered malicious for some applications.

5. Implications of the Definition of CIAOs

Analyzing the previous sections' definitions provides insight into the pervasiveness of CIAO and CIntAO vulnerabilities, as well as various mechanisms' effectiveness at mitigating CIAOs.

5.1 Pervasiveness of CIAO and CIntAO Vulnerabilities

We've been surprised to find that any application that always blindly copies some untrusted input verbatim into its SQL° output is vulnerable to a (copy-based) CIAO at runtime. Theorem 9 formalizes this result; it states that if an application always includes an untrusted input (i_m) verbatim in its output (without even inspecting that input), and the same application has some input ($v_1, .., v_n$) for which it outputs a valid SQL° program, then there exists a way to construct the untrusted input (a_m) such that the application's output will contain an injected code symbol. The proof is constructive; it shows how to inject code into any such application using a detailed case analysis of the kind of value the untrusted input (v_m) gets injected into. Although the proof is tailored to SQL°, the proof techniques are general.

Theorem 9. *For all n-ary functions A and (n-1)-ary functions A' and A'', if $\forall i_1, .., i_n \colon A(i_1, .., i_n) = A'(i_1, .., i_{m-1}, i_{m+1}, .., i_n)\underline{i_m}A''(i_1, .., i_{m-1}, i_{m+1}, .., i_n)$, where $1 \leq m \leq n$ and $\exists v_1, .., v_n \colon (v_m \in \Sigma^+_{SQL^\circ} \wedge A(v_1, .., v_n) \in SQL^\circ)$, then $\exists a_1, .., a_n \colon A(a_1, .., a_n) \in SQL^\circ$ and $A(a_1, .., a_n)$ exhibits a (copy-based) CIAO.*

Proof. By assumption, $\exists v_1, .., v_n \colon A(v_1, .., v_n) \in SQL^\circ$. First, if $A(v_1, .., v_n)$ exhibits a CIAO, then simply set $a_1, .., a_n$ to $v_1, .., v_n$. On the other hand, if $A(v_1, .., v_n)$ does not exhibit a CIAO, then, by the definition of CIAOs, v_m must be a substring of a value, because v_m is not empty and appears verbatim in the output of $A(v_1, .., v_n)$. Note that changing the untrusted v_m input to a_m, without changing any of the other $n-1$ inputs, will cause A to output $A'(v_1, .., v_{m-1}, v_{m+1}, .., v_n)\underline{a_m}A''(v_1, .., v_{m-1}, v_{m+1}, .., v_n)$, that is, the same output program but with a_m instead of v_m. We will show that no matter the type of the value that v_m is a substring of, there exists an a_m that will cause $A(v_1, .., v_{m-1}, a_m, v_{m+1}, .., v_n)$ to exhibit a (copy-based) CIAO but still remain valid.

- *Case* STR_LITERAL: a_m = if (the first character of v_m is not an apostrophe or is the first apostrophe of a double-apostrophe escape sequence) then ’ + GLOBAL + ‘v_m else ‘ ’ + GLOBAL + v_m

Examples:

- ‘fname’ becomes ‘f’ + GLOBAL + ‘name’.
- ‘ ‘ ’ ’ becomes ‘ ’ + GLOBAL + ‘ ‘ ’ ’.
- ‘fname’ becomes ‘ ’ + GLOBAL + ‘fname’.

Let s_m denote the string literal that v_m is a substring of. If v_m does not start s_m (which could happen if v_m begins with the first apostrophe of a double-apostrophe escape sequence), then the string literal must have been started earlier, as $A(v_1, .., v_n) \in SQL^\circ$. In this case, our construction of a_m terminates the string literal that has been started, inserts a code symbol (the concatenation operator), a global variable, another code symbol, and then begins a second string literal. If v_m terminated s_m, then this new string literal will also be terminated by v_m. If v_m did not terminate s_m, then it must have been terminated later, again because $A(v_1, .., v_n) \in SQL^\circ$. As a result, this second string literal will also be terminated later. On the other hand, if v_m did start s_m (or is the second apostrophe of a double-apostrophe escape sequence), then our construction creates an empty string literal (or finishes the escape sequence and terminates the literal) and concatenates a global variable and then concatenates another second string literal started by v_m. Again, we know that this second string literal will be terminated, either by v_m or the characters following it, for the same reasons as earlier. Thus, our construction of a_m causes s_m, when a_m has been substituted for v_m, to be parsed as $s' + GLOBAL + s''$, where s' and s'' are both valid string literals. Note that `expr OP expr` is a valid `expr`, and as long as GLOBAL is of type string, $s' + GLOBAL + s''$ will be of the same type as s_m. As any `expr` can by replaced by another `expr` of the same type, the program will remain valid after the substitution of a_m for v_m. As a_m contains a code symbol (i.e. 2 concatenation operators, as well as whitespace), a CIAO is exhibited.

- *Case* `INT_LITERAL`, `FLOAT_LITERAL`: $a_m = \underline{v_m\text{1*exit()*2}}$

  Examples:

  - -100 becomes -1 * exit() * 2100.
  - 11E34 becomes 11E1 * exit() *234.

  This construction works for similar reasons as the `STRING_LITERAL` case above; In addition to the 2 multiplication code symbols, this construction also has a function call.

- *Case* `TRUE`, `FALSE`, `NULL`: Let ID_m denote the keyword that v_m is a substring of; hence ID_m can be written as $ID_{m-}v_m ID_{m+}$ (where ID_{m-} and ID_{m+} are in $\Sigma^*_{SQL^\circ}$). If ID_m has a boolean type, then let OP be `OR` and let SUB be $1000 > GLOBAL$. Otherwise, let OP be `+`, and let SUB be $exit()$ if ID_m has an integer or float type, or $GLOBAL$ otherwise. Then let $a_m = \underline{v_m ID_{m+}\ \ OP\ SUB\ OP\ \ ID_{m-} v_m}$.

  Examples:

  - FALSE becomes FALSE OR 1000> GLOBAL OR FALSE.
  - NULL + 3 becomes NULL + exit() + NULL + 3.

  By assumption, v_m is a substring of a keyword ID_m. We assumed earlier that $A(v_1, .., v_n) \in SQL^\circ$, so ID_m must be a valid keyword. We also know that in $A(v_1, .., v_n)$, v_m is preceded by ID_{m-} and followed by ID_{m+}. We construct a_m such that it finishes the identifier or keyword started by the existing ID_{m-}, inserts a code symbol depending on the type of ID_m, conducts an attack, then inserts another code symbol, and begins a new identifier or keyword to be finished by the existing ID_{m+}. As a result, where originally the program used ID_m as an `expr`, the modified program uses $ID_m\ OP\ SUB\ OP\ ID_m$. Furthermore, the type of the `expr` remains unchanged, as if ID_m has a boolean type, then OP will be `OR`, and a boolean OR'd with a boolean is a boolean. If ID_m has a float or integer type, then OP will be the arithmetic operator `+`, which will return either a float or an integer type. If ID_m has a string type, then the `+` operator denotes concatenation, and two stings concatenated with each other form a string. Note that the only keyword that is a value and can have a type of int, float, or string is `NULL`, as it can assume any type.

  Note that a CIAO has already occurred when `NULL` is used in `expr IS NULL`, because `NULL` is only a value when parsed as an entire `expr`. □

Furthermore, any application that verbatim copies untrusted input into the (SQL°) output program is either vulnerable to CIntAOs or can be made to output an invalid program. Again, the proof is constructive; it shows how to create an untrusted input that changes the sequence of code symbols in, or invalidates, the output program.

Theorem 10. *For all n-ary functions A and (n-1)-ary functions A' and A'', if* $\forall i_1, .., i_n\colon A(i_1, .., i_n) = A'(i_1, .., i_{m-1}, i_{m+1}, .., i_n)\underline{i_m}A''(i_1, .., i_{m-1}, i_{m+1}, .., i_n)$, *where* $1 \leq m \leq n$ *and* $\exists v_1, .., v_n\colon (v_m \in \Sigma^+_{SQL^\circ} \wedge A(v_1, .., v_n) \in SQL^\circ)$, *then A either exhibits a CIntAO or can be made to produce an invalid SQL° program.*

Proof. Observe that every symbol in an SQL° program is either part of a value or not. If v_m contains a symbol recognized as part of a value, then the input can be modified in the manner described in the proof of Theorem 9, and the sequence of code symbols will be modified; by definition, A exhibits a CIntAO. On the other hand, if v_m contains a code symbol, then a different symbol can be provided. If the SQL° program is still valid, then a CIntAO has occurred, as the sequence of code symbols has changed. If changing the code symbol made the program invalid then the second condition of the implication is satisfied. □

Given that (program-outputting) applications commonly copy some untrusted input verbatim into the output, Theorems 9–10 show that vulnerabilities to CIAOs and CIntAOs are pervasive.

5.2 Limitations of Static and Black-box Mechanisms to Detect CIAOs

Determining whether an application is vulnerable to CIAOs requires knowing which input symbols propagate to the output program. This makes it undecidable to precisely detect (both copy-based and data-dependency-based) CIAOs using static code analysis or black-box analysis.

Theorem 11. *There exists an application A, which inputs a string of symbols over alphabet Σ and outputs L-programs, such that it is undecidable, when given only an input string* $s \in \Sigma^*$ *and a (e.g., Turing-machine) encoding of A, to determine whether A exhibits a (copy-based or data-dependency-based) CIAO on s.*

Proof. Let A be an application that inputs a string s, executes subprogram p, and then outputs s if s equals “1+1” but otherwise outputs just “1”. This A outputs programs in any language having integers and addition. Also, A exhibits a (copy-based and data-dependency-based) CIAO iff its subprogram p halts and its input s is “1+1”. Statically determining whether A exhibits a CIAO on input “1+1” therefore reduces to the halting problem. □

Theorem 12. *There exists an application A, which inputs a string of symbols over alphabet Σ and outputs L-programs, such that it is impossible, when given only an input string* $s \in \Sigma^*$ *and the*

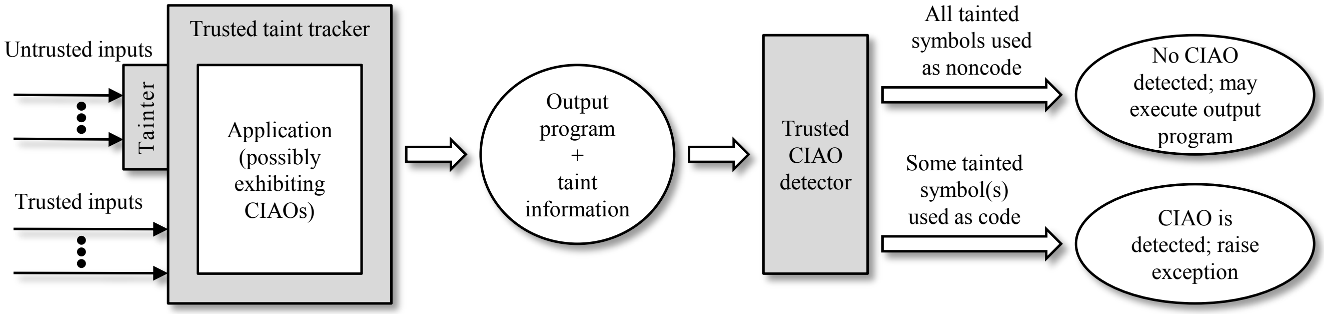
Untrusted inputs
Tainter
Trusted inputs
Trusted taint tracker
Application (possibly exhibiting CIAOs)
Output program + taint information
Trusted CIAO detector
All tainted symbols used as noncode
Some tainted symbol(s) used as code
No CIAO detected; may execute output program
CIAO is detected; raise exception

Figure 3. Overview of a CIAO-Preventing Mechanism. Trusted components are shaded.

ability (i.e., an oracle) to predict the output of A on any input, to determine whether A exhibits a (copy-based or data-dependency-based) CIAO on s. In other words, there exist observationally equivalent applications A and A' and a string s such that $A(s)$ exhibits a CIAO but $A'(s)$ does not exhibit a CIAO.

Proof. Let A be the input-echoing application `output(input)` and A' the conditional-copy application from Section 4.2.2. Recall from Section 4.2.2 that A and A' are observationally equivalent, A is vulnerable to CIAOs (e.g., on an input like `1+1`), and A' is invulnerable to CIAOs. □

Theorems 11–12 are interesting because they rule out certain classes of mechanisms from being able to precisely detect CIAOs. Some of the mechanisms ruled out were previously thought to precisely detect CIAOs; an example is SQLCHECK's black-box, "bracket"-based tainting mechanism (in which untrusted inputs get surrounded by special characters, and output symbols are considered tainted iff they're surrounded by those characters) [34]. Although previous work showed that SQLCHECK's tainting mechanism is flawed [3], Theorems 11–12 are more general, in that they rule out entire classes of mechanisms from being able to precisely detect CIAOs.

Of course, Theorems 11–12 don't rule out static analysis and black-box mechanisms as being useful for mitigating CIAOs. Although such mechanisms can't detect CIAOs precisely, they can detect CIAOs conservatively (i.e., soundly but not completely) with no/low runtime overhead, while avoiding the many practical obstacles to monitoring taints dynamically (some of which are described in Section 5.4).

However, sound static mechanisms for detecting CIAOs must be so conservative as to reject a large class of common applications, which may limit their appeal. Recall that Theorem 9 showed that all applications that copy some untrusted input verbatim into an output program can be made to exhibit a CIAO at runtime. Hence, sound static mechanisms for detecting CIAOs must reject all of this large class of common applications. (Similarly, Theorem 10 implies that sound CIntAO-detecting static mechanisms must reject all such applications as well.) Related work on static-analysis techniques for detecting CIAOs [12, 35, 40] appear to be consistent with this result; none seem to allow applications to copy untrusted input verbatim into output programs. Having a formal definition of CIAOs makes it possible to prove that this characteristic is mandatory for all sound, static, CIAO-detecting mechanisms.

5.3 Design of Mechanisms for Precisely Detecting and Preventing CIAOs

Theorems 11–12 prove that precisely detecting CIAOs requires a dynamic white-box mechanism. A high-level design of such a mechanism follows straightforwardly from the definitions in Sections 3–4.

A dynamic white-box mechanism can precisely detect and prevent CIAOs by:

- tainting all symbols input to an application A from untrusted sources,
- transparently tracking one taint bit per symbol,
- propagating taints through exactly A's copy and output operations (for copy-based CIAOs) or all data operations (for data-dependency-based CIAOs),
- intercepting A's output programs, and
- forbidding execution of output programs that contain at least one tainted symbol used outside a value (i.e., as code).

Figure 3 illustrates such a mechanism.

Theorem 13. *Assuming a mechanism M performs these operations on an application that outputs programs in a language with decidable free-variable (FV) and value (Val) functions, it is decidable for M to precisely detect and prevent CIAOs.*

Proof. Immediate by Definition 7 and the definitions of data-dependency CIAOs and mechanism M given above. □

Algorithm 1 directly implements this generic design of dynamic white-box mechanisms for preventing CIAOs. The algorithm relies on auxiliary functions for (1) adding taint tracking to applications, (2) signaling that untrusted inputs are tainted, (3) calculating the set of free variables in a sequence of program symbols, and (4) deciding whether program symbols constitute a value.

Each of the three nested *for* loops in Algorithm 1 executes $O(n)$ times, where n denotes the size of the output program. Hence, if we ignore the complexities of the FV and Val functions (which are dependent on the output-program language), the top-level *for* loop of Algorithm 1 runs in $O(n^3)$ time. Assuming that the FV and Val functions run in time linear in their input size, then, the top-level *for* loop of Algorithm 1 runs in $O(n^4)$ time.

The space required by the top-level *for* loop of Algorithm 1 consists of the i, low, and $high$ counters (each of size $O(\lg n)$), the $IsCiao$ flag (of size $O(1)$), and whatever space is required to invoke and execute the FV and Val functions. Assuming that invoking and executing the FV and Val functions uses space linear in their input size, then, the top-level *for* loop of Algorithm 1 uses $O(n)$ space.

Optimized CIAO-Preventing Mechanism Algorithm 1 can be optimized to run in $O(n)$ time and space, under the assumption that the output-program language has an LR(k) grammar in which every closed value matches some syntactic category (e.g., in SQL$^\circ$ every

Algorithm 1: Basic CIAO-Preventing Mechanism

```
Input: Application $A$ and inputs $T$, $U$ (trusted, untrusted)
Ensure: $A$'s output is executed iff it doesn't exhibit a CIAO
  $A' \leftarrow$ AddTaintTracking($A$)
  $Output \leftarrow A'$ ($T$, Taint($U$))
  for $i \leftarrow 1$ to $|Output|$ do
      if tainted($Output[i]$) then
          $IsCiao \leftarrow$ true
          for $low \leftarrow 1$ to $i$ do
              for $high \leftarrow i$ to $|Output|$ do
                  if FV($Output$, $low$, $high$) $= \emptyset$ and
                    Val($Output$, $low$, $high$) then
                      $IsCiao \leftarrow$ false
                  end if
              end for
          end for
          if $IsCiao$ then
              throw $CiaoException$
          end if
      end if
  end for
  Execute($Output$)
```

Algorithm 2: Optimized CIAO-Preventing Mechanism (for applications whose output-program language has an LR(k) grammar in which every closed value matches some syntactic category)

```
Input: Application $A$ and inputs $T$, $U$ (trusted, untrusted)
Ensure: $A$'s output is executed iff it doesn't exhibit a CIAO
  $A' \leftarrow$ AddTaintTracking($A$)
  $Output \leftarrow A'$ ($T$, Taint($U$))
  let
      function clearTaints ($p_1, \ldots, p_n$) =
          for $i \leftarrow 1$ to $n$ do
              $Current \leftarrow$ dereference($p_i$)
              if $Current$ is a terminal then
                  for $j \leftarrow Current$.begin to $Current$.end do
                      $Output[j]$.Tainted = false
                  end for
              else if $Current$.isVal=false then
                  clearTaints($Current$.children)
              end if
              if $i < n$ then  // clear tainted whitespace, if any
                  $Next \leftarrow$ dereference($p_{i+1}$)
                  for $j \leftarrow Current$.end to $Next$.begin do
                      $Output[j]$.Tainted = false
                  end for
              end if
          end for
      end function
  end let
  in
      $tokens \leftarrow$ run tokenize ($Output$)
          on recognition of token $t$ do
              $t$.begin $\leftarrow$ position of first symbol of $t$ in $Output$
              $t$.end $\leftarrow$ position of last symbol of $t$ in $Output$
          end on
      end run
      run shift-reduce-parse ($tokens$)
          on reducing by production $N ::= s_1\ s_2\ \ldots\ s_n$, where
          $s_1..s_n$ is a closed value do
              $N$.isVal $\leftarrow$ true
              clearTaints(list of pointers to $s_1, \ldots, s_n$)
              $N$.begin $\leftarrow s_1$.begin
              $N$.end $\leftarrow s_n$.end
          end on
          on reducing by production $N ::= s_1\ s_2\ \ldots\ s_n$, where
          $s_1..s_n$ is not a closed value do
              $N$.isVal $\leftarrow$ false
              $N$.children $\leftarrow$ list of pointers to $s_1..s_n$
              $N$.begin $\leftarrow s_1$.begin
              $N$.end $\leftarrow s_n$.end
          end on
      end run
      for $i \leftarrow 1$ to $|Output|$ do
          if tainted($Output[i]$) then
              throw $CiaoException$
          end if
      end for
  end in
  Execute($Output$)
```

closed value matches the `expr` category). When output-program languages satisfy this assumption, Algorithm 1's top-level *for* loop can be replaced with a shift-reduce parse of the application's output program. When reducing a closed-value right-side R of a production to a nonterminal N, the parser sets an `isVal` attribute for N and erases taints on all output-program symbols represented by R (except for any symbols represented by nonterminals in R for which `isVal` has been set—such symbols have already had their taints erased). After running this taint-erasing parser, all output-program taints in closed values will have been erased, so a CIAO is detected if and only if some tainted symbol remains in the output program.

Algorithm 2 presents pseudocode for this optimized CIAO-preventing mechanism. The algorithm relies on auxiliary functions for (1) adding taint tracking to applications, (2) signaling that untrusted inputs are tainted, (3) tokenizing output programs, and (4) shift-reduce parsing output programs.

Theorem 14. *The* let-in *block of Algorithm 2 runs in $O(n)$ time and space.*

Proof. The tokenization portion of Algorithm 2 runs in $O(n)$ time and space (where again n is the size of the application's output program). A standard shift-reduce parse of the output program, without the additional actions performed on reductions, runs in $O(n)$ time and space; the total number of right-hand-side-production symbols reduced to nonterminals during the parse is $O(n)$ [15]. Because the total number of right-hand-side-production symbols reduced to nonterminals during the parse is $O(n)$, all the non-taint-clearing reduction actions in Algorithm 2 (i.e., N.isVal$\leftarrow$`true`, N.begin $\leftarrow s_1$.begin, etc.) occur in $O(n)$ total time and space. The *for* loop in Algorithm 2's *in* block also runs in $O(n)$ time and space, so Algorithm 2's entire *in* block uses linear time and space, in addition to the time and space used to clear taints.

To determine the total time and space used by taint-clearing operations, observe that clearTaints is always initially invoked, in Algorithm 2's *in* block, on symbols matching a nonterminal N such that N.isVal=`true`. During execution, clearTaints may call itself recursively only on parse-tree-descendent nonterminals with `false` isVal attributes. Because parsing is bottom-up, then, pointers to the same syntax-tree symbol may never be passed as arguments to clearTaints more than once, and every output-program taint may be cleared at most once (technically this result also relies on the facts that isVal attributes are constant once set, taints can only be cleared by clearTaints, and clearTaints, when called on pointers to symbols $s_1..s_n$, can only clear taints at output-program positions s_1.begin to s_n.end). Also observe that, ignoring recursive clearTaints calls, each run of clearTaints executes in time proportional to the number of pointers to nonterminals passed as arguments plus the number of taints cleared. Then because (1) pointers to the same syntax-tree symbol may never be passed as arguments to clearTaints more than once, (2) the total number of nonterminals in the parse tree is $O(n)$ [15], (3) every output-program taint may be cleared at most once, and (4) each run of clearTaints executes in time proportional to the number of pointers to nonterminals passed as arguments plus the number of taints cleared, we have that the total time (and therefore space) used by all clearTaints operations is $O(n)$.

Hence, Algorithm 2's entire *let-in* block uses $O(n)$ time and space, as required. □

5.4 Obstacles to Monitoring Taints in Practice

Many taint-monitoring mechanisms and frameworks exist for mitigating CIAOs (e.g., [6, 8, 25, 31, 38]). None separate code from noncode the way this paper has, but one framework, Dytan [6], which has not yet been publicly released, implements (for x86 applications) the taint-tracking functionality our definitions require. Hence, it appears possible to use Dytan to precisely detect (copy-based and data-dependency-based) CIAOs in x86 applications (by ensuring that all operations in Section 5.3's bulleted list are performed).

Even with powerful taint-monitoring frameworks like Dytan, there are several obstacles to ensuring that taint-monitoring mechanisms obey the four tainting constraints listed in Section 4.1. This subsection briefly summarizes these obstacles, most of which are discussed in greater length elsewhere (e.g., [6, 8, 24, 31, 33]).

The first of the four tainting constraints in Section 4.1 requires all symbols input to the application from untrusted sources to be tainted. Untrusted inputs may come from many sources (e.g., HTTP GET and POST requests, cookies, server variables, or a database), and enumerating all these untrusted sources may be difficult and error prone. Hence, following Halfond, Orso, and Manolios, one might instead use *positive* tainting [8] (i.e., tracking which output-program symbols derive from *trusted* sources, often just the string literals hardcoded in an application). It would be straightforward to adjust this paper's definition of CIAOs to use positive (rather than negative) tainting: CIAOs would occur when some code symbol in an output program is not positively tainted.

The second and third of the four tainting constraints require that taints propagate through exactly copy and output operations (for copy-based CIAOs), or all data operations (for data-dependency-based CIAOs). Because a taint bit must be tracked for every input symbol, the tainting mechanism must operate with fine granularity, which previous work has found to induce high runtime overhead (e.g., many thousands of percent of overhead) [6, 24, 38]. In addition, monitoring taints typically requires executing applications in modified runtime environments, which limits portability [9]. And propagating taints through output operations, so output programs can be caught and checked prior to being executed, may be difficult; it may be hard to enumerate all the ways an application can output programs (e.g., to files, remote hosts, or standard output). If an application's outputs can circumvent a CIAO-mitigating mechanism, the mechanism is unsound. Applications might also circumvent taint-monitoring mechanisms by executing external (e.g., native) code [8].

The last of the four tainting constraints requires taints to be transparent. This transparency ensures that taint tracking does not affect application behaviors; CIAO-preventing mechanisms should only modify application behaviors when attacks are detected (in which case the behavior must be modified to prevent injected code from being output). To be transparent, tainting mechanisms have to isolate taints from applications. Hence, CIAO-mitigating mechanisms cannot use bracketing techniques to track taints (e.g., [34])—the brackets are visible to applications [3]. Another important obstacle to ensuring transparency in practice is that runtime mechanisms generally induce overhead on application performance, and this overhead may make time-sensitive applications behave differently. Perfect transparency may therefore be difficult or impossible to achieve for time-sensitive applications in practice.

6. Summary

This paper has defined code-injection attacks on outputs. The definition simply considers CIAOs to occur when untrusted inputs get used as nonvalues (or open values) in output programs. This definition avoids problems with conventional CIAO definitions, which sometimes consider CIAOs to be non-CIAOs and vice versa.

The new definition of CIAOs has been used to:

- Distinguish between copy-based CIAOs, data-dependency-based CIAOs, and CIntAOs based on whether taints propagate through copy, data, or all (data and control) dependencies.
- Prove that a large class of applications (i.e., those that always blindly copy some untrusted input to the output program) are inherently vulnerable to CIAOs and CIntAOs, so sound static mechanisms must disallow these applications from executing.
- Prove that precisely detecting CIAOs requires dynamic white-box mechanisms. The generic design of such mechanisms follows immediately from the definition of CIAOs. Under reasonable assumptions these mechanisms can be optimized to detect CIAOs in output programs in $O(n)$ time and space. Nonetheless, due to their reliance on taint tracking, many obstacles impede implementation of precise CIAO-mitigating mechanisms in practice.

Hence, the new definition of CIAOs has been used to analyze precisely when they occur, how they can be mitigated, and how efficiently they can be mitigated. We hope these results can serve as a foundation for improving the effectiveness of future CIAO-mitigating mechanisms.

Acknowledgments

We are grateful to the anonymous reviewers of this paper, who provided many valuable comments. This research was supported by NSF grants CNS-0716343 and CNS-0742736.

References

[1] C. Anley. Advanced SQL injection in SQL server applications. White paper, Next Generation Security Software, 2002.

[2] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan. Candid: preventing SQL injection attacks using dynamic candidate evaluations. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 12–24, 2007.

[3] P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan. CANDID: Dynamic candidate evaluations for automatic prevention of SQL injection attacks. *ACM Trans. Inf. Syst. Secur.*, 13(2):1–39, Feb. 2010.

[4] M. Bravenboer, E. Dolstra, and E. Visser. Preventing injection attacks with syntax embeddings. *Science of Computer Programming*, 75(7): 473–495, July 2010.

[5] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using parse tree validation to prevent sql injection attacks. In *SEM '05: Proceedings of the 5th international workshop on software engineering and middleware*, pages 106–113, 2005.

[6] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the ACM International Symposium on Software Testing and Analysis*, pages 196–206, 2007.

[7] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. Ccured in the real world. *SIGPLAN Notices*, 38:232–244, May 2003.

[8] W. Halfond, A. Orso, and P. Manolios. Wasp: Protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Trans. Softw. Eng.*, 34(1):65–81, 2008.

[9] W. G. Halfond, J. Viegas, and A. Orso. A Classification of SQL-Injection Attacks and Countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, March 2006.

[10] R. Hansen and M. Patterson. *Stopping Injection Attacks with Computational Theory*, July 2005. In Black Hat USA.

[11] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *Proceedings of the General Track of the USENIX Annual Technical Conference*, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[12] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 258–263, 2006.

[13] A. Kieżun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of the International Conference on Software Engineering*, May 2009.

[14] K. Kline and D. Kline. *SQL in a Nutshell*, chapter 4. O'Reilly, 2001.

[15] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[16] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1963.

[17] Z. Luo, T. Rezk, and M. Serrano. Automated code injection prevention for web applications. In *Proceedings of the Conference on Theory of Security and Applications*, 2011.

[18] Microsoft. *SQL Minimum Grammar*, 2011. http://msdn.microsoft.com/en-us/library/ms711725(VS.85).aspx.

[19] Microsoft. *CREATE FUNCTION (Transact-SQL)*, 2011. http://msdn.microsoft.com/en-us/library/ms186755.aspx.

[20] *CWE/SANS Top 25 Most Dangerous Software Errors*. The MITRE Corporation, 2009. Document version 1.4, http://cwe.mitre.org/top25/archive/2009/2009_cwe_sans_top_25.pdf.

[21] *CWE/SANS Top 25 Most Dangerous Software Errors*. The MITRE Corporation, 2010. Document version 1.08, http://cwe.mitre.org/top25/archive/2010/2010_cwe_sans_top25.pdf.

[22] *CWE/SANS Top 25 Most Dangerous Software Errors*. The MITRE Corporation, 2011. Document version 1.0.2, http://cwe.mitre.org/top25/archive/2011/2011_cwe_sans_top25.pdf.

[23] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27:477–526, May 2005.

[24] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2005.

[25] A. Nguyen-tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the IFIP International Information Security Conference*, pages 372–382, 2005.

[26] G. Ollmann. Second order code injection attacks. Technical report, NGS Software, 2004.

[27] Oracle. How to write injection-proof PL/SQL. An Oracle White Paper, December 2008. URL http://www.oracle.com/technetwork/database/features/plsql/overview/how-to-write-injection-proof-plsql-1-129572.pdf. Page 11.

[28] Oracle. *CREATE FUNCTION Syntax for User-Defined Functions*, 2011. http://dev.mysql.com/doc/refman/5.6/en/create-function-udf.html.

[29] Oracle. *CREATE FUNCTION*, 2011. http://download.oracle.com/docs/cd/E11882_01/server.112/e17118/statements_5011.htm.

[30] php. *phpMyAdmin*. http://www.phpmyadmin.net.

[31] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of Recent Advances in Intrusion Detection (RAID)*, 2005.

[32] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[33] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010.

[34] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–382, 2006.

[35] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of web applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 87–97, 2009.

[36] S. Tzu. The art of war. The Project Gutenberg eBook. Translated by Lionel Giles. http://www.gutenberg.org/cache/epub/17405/pg17405.txt.

[37] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2007.

[38] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[39] Y. Younan, P. Philippaerts, F. Piessens, W. Joosen, S. Lachmund, and T. Walter. Filter-resistant code injection on ARM. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 11–20, 2009.

[40] X. Zhang and Z. Wang. A static analysis tool for detecting web application injection vulnerabilities for ASP program. In *International Conference on e-Business and Information System Security (EBISS)*, pages 1 –5, May 2010.