AutoRand: Automatic Keyword Randomization to Prevent Injection Attacks

Jeff Perkins[1], Jordan Eikenberry[1], Alessandro Coglio[2], Daniel Willenson[1], Stelios Sidiroglou-Douskos[1], and Martin Rinard[1]

[1] MIT/CSAIL Cambridge MA USA
{jhp, jeikenberry, dwillenson, stelios, rinard}@csail.mit.edu
[2] Kestrel Institute, Palo Alto CA USA
coglio@kestrel.edu

Abstract. AutoRand automatically transforms Java applications to use SQL keyword randomization to defend against SQL injection vulnerabilities. AutoRand is completely automatic. Unlike previous approaches it requires *no* manual modifications to existing code and does not require source (it works directly on Java bytecode). It can thus easily be applied to the large numbers of existing potentially insecure applications without developer assistance. Our key technical innovation is *augmented strings*. Augmented strings allow extra information (such as random keys) to be embedded within a string. AutoRand transforms string operations so that the extra information is transparent to the program, but is always propagated with each string operation. AutoRand checks each keyword at SQL statements for the random key. Experimental results on large, production Java applications and malicious inputs provided by an independent evaluation team hired by an agency of the United States government showed that AutoRand successfully blocked all SQL injection attacks and preserved transparent execution for benign inputs, all with low overhead.

1 Introduction

SQL injection attacks are a critical vector of security exploits in deployed applications. SQL Injection [10] is the first entry in the CWE/SANS list of the top 25 most dangerous software errors [23]. Injection errors are also the first entry in OWASP's top 10 web application security problems [21]. Given the demonstrated ability of attackers to exploit such vulnerabilities [8] and the exploitable opportunities that this class of vulnerabilities presents to attackers on an ongoing basis [9], techniques that eliminate SQL injection vulnerabilities and prevent SQL injection attacks are of primary importance to the future security of our information technology infrastructure.

On the surface it would seem that SQL attacks could be prevented by following good coding practices (such as using prepared statements and/or sanitizing inputs) that have been available for many years. Unfortunately, these practices have to be followed 100% of the time or an attack may be enabled. The continued prevalence of SQL injection attacks [9] bears evidence to the fact that a different approach that doesn't rely on error-free development is required. Furthermore there is a large amount of existing SQL code that needs protection. It is unrealistic to expect this code to be retrofitted. The developer resources are often not available and in many cases the source code may not be accessible.

1.1 SQL Keyword Randomization

Instruction set randomization [17] protects systems against code-injection attacks by creating randomized instruction sets. An attacker that does not know the instruction set in use will inject invalid code which will not execute correctly.

SQL keyword randomization applies the same technique to SQL injection attacks. Conceptually the SQL grammar is changed to use randomized SQL keywords that are not known to possible attackers. Any code that is injected will not contain valid keywords and will thus yield an error when parsed thwarting any attack.

Existing randomization systems [5] require the developer to *manually* modify the program to randomize the SQL keywords that appear in constant strings. This requires program source and possibly significant developer time (see Sect 6.2). In many cases, neither of these may be available. An automatic system is needed to address the large numbers of existing potentially insecure applications.

Building an automatic system, however, is challenging. A working solution must randomize all SQL keywords that can reach an SQL statement (by any path) while ensuring that those modifications do not change the semantics of the program or are made visible outside of the program (because that would leak the random key). Operations on strings containing random keys must preserve the keys and the original semantics of the operation. Since the keys change both the length and contents of the string, many operations (e.g., `substring`, `charAt`, `replace`) must be automatically converted.

1.2 AutoRand

We present a new system, AutoRand, that automatically transforms Java applications to use randomized SQL keywords[3].

The resulting transformed Java application is protected against SQL injection attacks that rely on using SQL keywords in the malicious input to change the structure of the SQL command passed to the SQL execution engine.

AutoRand automatically translates the Java bytecodes of the application to randomize any SQL keywords that appear in program constants or in trusted inputs. It transparently propagates the randomized versions of the keywords across string operations. Any use of randomized SQL keywords in other operations (e.g., file/socket writes, string comparisons, etc) are automatically derandomized to ensure that the program's semantics are maintained.

AutoRand also inserts code that checks each SQL command to ensure that all keywords have the correct random value. If any keywords (such as those inserted

[3] We use the term *keyword* to include keywords, operators and comment tokens

by an attacker) are not correct, an exception is thrown. If all of the keywords are correct, the query is de-randomized and passed to the normal SQL routine.

AutoRand operates directly on byte-code and does not require source or manual modifications. It can easily be applied to existing applications without developer assistance. To our knowledge it is the first system to automatically apply SQL keyword randomization to existing programs.

Experimental results on large, production Java applications and malicious inputs provided by an independent evaluation team hired by an agency of the United States government showed that AutoRand successfully blocked all SQL injection attacks with no false positives and negligible overhead.

1.3 Augmented Strings

Our key technical innovation is *augmented strings*. Augmented strings allow additional information to be added to strings. This additional information is handled *transparently* with respect to the application. Augmented strings are designed to ensure that, with the exception of augmented checks (such as SQL query checks), the application exhibits the same behavior with augmented strings as without. The additional information is accounted for in all string operations to ensure that it is propagated across the operation without changing the semantics of the program. To accomplish this transparency, AutoRand automatically modifies string operations to ensure that the presence of the additional information is not visible to the program itself (e.g., conditionals over string values, reflection, etc) or externally (e.g., network writes, environment variables access, etc).

The additional information in an augmented string is identified by a random key. The key is complex enough to ensure that it will not occur (within some arbitrarily small probability) by happenstance in the program's input or constants. This allows the additional information to be precisely identified.

In the case of AutoRand, the random key is placed immediately after each SQL keyword to create a randomized version of the keyword in the augmented string. To our knowledge the augmented strings approach is novel and could be used in broader contexts than SQL injection, such as tracking the detailed provenance (filename, URL) of each token in a string, randomization for other injection issues (such as command injection) or carrying debug information.

1.4 Experimental Evaluation

We evaluate the AutoRand implementation on a set of benchmarks and associated inputs developed by an independent evaluation team hired by the sponsor of this research (an agency of the United States government). The evaluation team started with a set of existing large, production Java applications, inserted SQL injection vulnerabilities into the applications, and developed inputs that exploit the vulnerabilities. The evaluation team was given complete information about the AutoRand implementation. The results of the evaluation show that AutoRand successfully blocked all SQL injection attacks. To test transparency and preservation of functionality, the evaluation also exercised the applications on benign inputs. The results showed identical behavior for each benign input. We note that this evaluation worked with applications that are over an order of magnitude larger than any previous evaluation of SQL injection attack defenses for Java programs of which we are aware [14, 4, 7]. AutoRand's ability to successfully block SQL injection attacks in these applications highlights the effectiveness of AutoRand's techniques and the robustness of the AutoRand implementation.

1.5 Contributions

This paper makes the following contributions:

- **AutoRand:** It presents a system for automatic and transparent SQL keyword randomization to *automatically* eliminate SQL injection vulnerabilities.
- **Augmented Strings:** It presents a technique that transparently adds information (in this case a random key) to strings and propagates that information across string operations. The original semantics of the application are preserved except where explicit checks utilizing the additional information are added (in this case for SQL injection attacks).
- **Experimental Evaluation:** It presents results from applications and inputs developed by an independent evaluation team. These results show that AutoRand successfully blocked all of the developed SQL injection attacks and correctly preserved transparent execution for all of the benign inputs.

2 Example

We next present an example that illustrates how AutoRand nullifies SQL injection attacks.

2.1 Vulnerable Code

Consider the Java fragment

```
String query = "select * from users where "
  "username='" + username + "' and password='" + password + "'";
ResultSet results =  databaseConnection.
  createStatement().executeQuery(query);
```
(1)

which looks up, in the `users` table of a database, the user whose name and password are in the string variables `username` and `password`. The query is constructed by combining a constant SQL code template with variable fragments that should only specify data. If `username` is `jqd` and `password` is `xB34qy5s`, the query sent to the database is

```
select * from users where username='jqd' and password='xB34qy5s'
```
(2)

and the application operates normally. However, if `username` is "`' or 1=1 --`" and `password` is the empty string, the query sent to the database is

```
select * from users where username='' or 1=1 --' and password=''
```
(3)

which always returns all records from the `users` table, since the password check has been commented out by the comment marker `--`. The latter input is crafted to subvert normal operation by executing SQL code that is part of the input data. This kind of subversion may cause loss of confidentiality and/or integrity. E.g., if `username` is "`'; drop table users --`" and `password` is the empty string, the query sent to the database is

```
select * from users where username=''; drop table users
  --' and password=''                                        (4)
```

where the semicolon separates the (now irrelevant) query from an injected `drop` statement that deletes the `users` table from the database.

If `username` and `password` are set from application inputs, the execution of the SQL query in (1) should be preceded by input validation, i.e., checks that `username` and `password` do not contain characters that may alter the structure of the SQL query (e.g., that they only contain letters and numbers). If the check fails, the inputs should be rejected or sanitized (e.g., by removing any character that is not a letter or a number). If the developer fails to include these checks, the code in (1) is vulnerable to SQL injection attacks.

2.2 Automatic Hardening by AutoRand

AutoRand automatically turns the code in (1) into code like

```
String query = "select<key> * from<key> users where<key> username='" +
  + username + "' and<key> password='" + password + "'";
ResultSet results = derandomizeAndExecuteQuery                       (5)
    (databaseConnection.createStatement(), query);
```

where `<key>` is a randomization key, i.e., a randomly chosen sequence of ASCII letters and numbers, e.g., `di83e2371A`. That is, all the SQL keywords that occur in string constants are randomized by appending `<key>`. The AutoRand runtime method `derandomizeAndExecuteQuery` tokenizes the query and checks each SQL keyword to ensure that it is suffixed by `<key>`. If the check succeeds, the query is deemed legitimate, all instances of `<key>` are removed, and the resulting query is executed normally by calling `executeQuery`. This check fails if an attacker injects a non-randomized keyword. For example, if `username` is "`' or 1=1 --`" and `password` is the empty string, the query

```
select<key> * from<key> users where<key> username=''
  or 1=1 --' and<key> password=''                              (6)
```

fails the check because `or` and `--` lack `<key>`. Since the attacker does not know the valid keywords for `or` and `--` (i.e., does not know `<key>`), they are unable to create a successful attack.

AutoRand also automatically transforms other parts of the code to make keyword randomization transparent to non-SQL uses of the mutated strings. For instance, using `String.length()` to take the length of `query` in (5) should return the same value as `query` in (1)—the randomization key should not contribute to the count. Transparency is particularly important for output-related uses of the mutated strings, e.g., `String.out.println(query)`, because if the attacker were to see the randomization key in some output (e.g., error message) they would be able to inject correctly randomized keywords.

3 Technical Approach

AutoRand protects a Java application against SQL injection by statically transforming each class of the application, producing a hardened version of the application.

3.1 Correctness

In Java, strings are objects, whose contents are manipulated exclusively via a standard API, which consists of the classes `String`, `StringBuilder` and `StringBuffer`.[4] AutoRand intervenes in string method calls to ensure that keys are propagated (*propagation*) and do not affect the application (*transparency*)—other than protecting against SQL injection. AutoRand's transformation is correct if it maintains these properties.

– **Transparency:** A given AutoRand program state and (side-effect free) operation is *transparent* if running the operation in the state produces the same result as running the corresponding original operation in the derandomized state.
– **Propagation:** A given operation satisfies propagation if each keyword that is propagated from its inputs to its outputs is consistently randomized (i.e., the output keyword is randomized if and only if the corresponding input keyword was randomized).

Transparency guarantees that the original semantics of the program hold (except for the added SQL checks). Propagation ensures that randomized keywords in program constants or trusted inputs propagate through string manipulations to SQL statements. This ensures that they will parse correctly (in the absence of injection attacks). If a randomized keyword were not propagated correctly to an SQL statement the statement would not parse correctly and an exception would be incorrectly thrown (a false positive). A propagation error would *not* result in a false negative as the lack of a randomized key will always be treated as an error. There is no path by which an attacker can add the key to their keywords (other than by knowing the key)

Transparency Abstractly, if op is an operation that takes a string S as input and yields a string as output, AutoRand's replacement operation op', in order to achieve transparency, must satisfy

$$op(S) = r^{-1}(op'(r(S))) \tag{7}$$

[4] For simplicity, we use the term 'string' to refer to objects of all three classes.

where r randomizes strings and r^{-1} derandomizes strings.[5] The requirement (7) is easily adapted to operations that take multiple strings as input or yield non-strings (e.g. r^{-1} is a no-op for `String.equals()`). Derandomization r^{-1} removes all instances of the key, not only instances that follow SQL keywords, thus, string operations that modify keywords will not affect transparency.

Some string methods return values other than strings. The derandomization operation r^{-1} is a no-op for non-strings. Thus op' must return the same value as op (as required by equation (7)). Many of the non-string return values are indices into strings. These indices must reference the derandomized version of the string, not the randomized version. AutoRand's replacement operations must also accept index arguments that are with respect to the derandomized version of the string. These operations map any index arguments from the derandomized string to the corresponding index in the randomized string. For example, the following code adds some text to an SQL statement following the `select` keyword. The length of the `select` keyword is hard-coded.

```
StringBuffer sb = new StringBuffer(...);
int offset = sb.indexOf("select") + 6;                    (8)
sb.insert(offset, " field1, field2");
```

For this to work correctly on a randomized `sb`, the index must be translated to the corresponding index in the randomized buffer (after `select<key>`). Note that code similar to this exists in the real-world applications that we tested.

Propagation Propagation is achieved if every randomized keyword in the input operands that is transferred to the result is also randomized in the result. For the purposes of SQL commands, keywords are a unit and only operations over a complete keyword (and not its individual characters) need to support propagation. Such sub-keyword operations may occur if the string is used for non-SQL purposes, but propagation is not required in such cases. As noted above, transparency is not affected by sub-keyword operations.

Abstractly, if op is an operation that takes a string S as input (where S may contain randomized keys) and yields a string as output, AutoRand's replacement operation op', in order to achieve propagation must satisfy

$$(K_r \in S) \wedge (K \in op(r^{-1}(S))) \iff K_r \in op'(S) \tag{9}$$

where r^{-1} derandomizes strings, K_r is a randomized keyword and K is the corresponding keyword. A keyword in the output *corresponds* to a keyword in the input only if it is the same instance of the keyword (i.e., the characters that make up the keyword in the input were copied to the output).

3.2 String Randomization

AutoRand randomizes (each SQL keyword in) each string constant in the application code.

[5] The requirement assumes that the key does not occur in S. The space of keys ensures a sufficiently small probability that the key occurs in the application code or data by happenstance.

AutoRand randomizes each string constant by tokenizing it and then appending the randomization key to all the SQL keywords in the string. The string is left unmodified if no SQL keywords are found in it. The set of tokens that AutoRand regards as SQL keywords is easily configurable. The current default configuration protects against injection of standard SQL [15] as well as non-standard SQL extensions for popular databases. Since SQL keywords are case-insensitive, the AutoRand tokenizer is case-insensitive.

Each keyword is randomized by appending a randomization key consisting of 10 ASCII letters (upper case or lower case) or digits. For example, `select` could become `selecta2831jfy6`. To minimize the possibility that an attacker could generate the key by chance, we use a large space consisting of 62^{10} (i.e., over 800 quadrillion) possible keys. This corresponds to about 60 bits, which is small for cryptographic keys, whose threat model is offline brute force search. However, AutoRand's keys have a different threat model, namely an attacker attempting injections over the network, whose latency limits the rate at which keys can be tried. Nonetheless, AutoRand's key length is configurable and could be easily increased. Increasing the key by 10 characters increases the overhead (see Sect 5.3) by only about 0.73%.

3.3 SQL API Calls

Java applications access SQL databases via a standard Java API. The `java.sql.Statement` class provides methods to execute SQL statements passed as string arguments, e.g., `executeQuery()` in (1).

AutoRand wraps each call by the application to the methods of `Statement` and `Connection` that receive SQL statements and prepared statements as string arguments. Even though a prepared statement is not vulnerable to injections when the template is instantiated, the creation of the prepared statement itself is vulnerable to injection when the string (e.g., "`select * from users where username=? and password=?`") is assembled from parts that are not all trusted.

Each method wrapper first checks that all the keywords in the SQL string include the correct key. If any keyword does not have the key, the SQL string is deemed to result from an attack and the wrapper throws an exception.

If all the keywords have the correct key, the method wrapper removes every occurrence of the key and then calls the method of `Statement` or `Connection` with the resulting string.

3.4 String Manipulations

Just randomizing strings as described in Section 3.2 and wrapping SQL API calls as described in Section 3.3 would nullify SQL injection attacks but could disrupt the normal operation of the application. For instance, if `String.equals` were called on a program constant containing a keyword and an input containing the same keyword, it would incorrectly return `false`, changing the semantics of the application. Also, if a randomized string makes its way to an output that is visible to the attacker (e.g., `System.out.println(query)`), the attacker could learn the key and inject correctly randomized keywords. Thus, AutoRand implements further transformations to make keyword randomization transparent to the application (other than hardening the application against SQL injection), including any output that may be visible to the attacker.

When necessary, AutoRand intervenes in string method calls by replacing calls to string methods with calls to methods in the AutoRand string library.

The Java string methods fall into a few basic categories. AutoRand's approach for accomplishing transparency and propagation for each of those categories is described in the following subsections. The category, propagation, and transparency for each string method are shown in Fig 2.

Observer Methods Observer methods do not create or modify strings. They are handled by derandomizing each of the string arguments and then invoking the original method. Transparency is trivially accomplished as the original method is run on the derandomized arguments. There are no keyword propagation issues since strings are not created or modified.

For example, the implementation for `String.length` and `String.equals` are:

```
AutoRandLength (String s) {
  return derand(s).length();
}
```

```
AutoRandEquals (String s1, String s2) {
    return derand(s1).equals(derand(s2));
}
```

Complete String Methods Complete string methods operate on entire strings, and not on portions of them. Since the random keys are incorporated into the string itself, any operations that only involve complete strings will work correctly without modification. For example, `String.concat()` and `StringBuffer.append()` function correctly on randomized strings without modification.

Fortunately, these are amongst the most commonly used of the String functions, which is partially responsible for AutoRand's low overhead.

Partial String Methods Partial string methods may operate on pieces of a string. The pieces are often specified by indices, but can also be specified by a string match (such as in `String.replace()`). For these methods, AutoRand transfers the operation from the original (derandomized) string to the randomized string.

The three basic operators for partial strings are `substring`, `insert`, and `delete`. The location in the string is specified by one or more indices.

AutoRand creates an index map between the derandomized and randomized versions of the string (an example is shown in Fig 1). This maps characters in the original (derandomized) string to the corresponding character in the randomized string. AutoRand implements `substring`, `insert`, and `delete` by looking up each index in the map and calling the original method on the randomized string using the mapped indices. This both propagates random keys and preserves transparency for each operation.

The `substring` method takes a substring from start (inclusive) to end (exclusive). Any substring that contains a keyword will include both the beginning

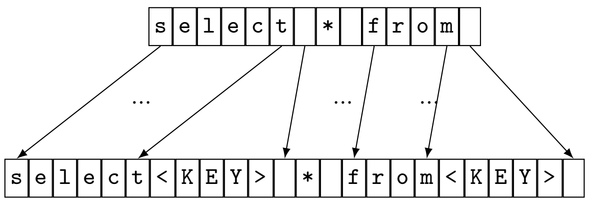

Fig. 1: Each character in the original (unrandomized) string is mapped to the corresponding character in the randomized string. There is no mapping to any of the characters in the randomization key. This ensures that no operation over mapped indices can create a partial key and that any operation over complete keywords will include the corresponding randomization key.

character of the keyword and the character immediately after the keyword. Since the map of the character after the keyword will point after the randomization key, any substring that includes the keyword will also include its key. For example, consider `substring(9,13)` on the string in Fig 1. This call would return the keyword `from` in the original string. After applying the index map this call is transformed into `substring(14,23)` on the randomized string. This will return `from<KEY>` in the randomized string, preserving the randomization key.

The `delete` method takes the same parameters as `substring` and works in the same fashion. For example, `delete(9,13)` would be transformed to `delete(14,23)` and would remove `from<KEY>` from the randomized string.

The `insert` method inserts its string argument before the specified index. The map ensures that inserts cannot occur between the keyword and its randomization key or in the middle of a randomization key, because there are no maps to those locations.

All other partial string methods can be built up from these core methods (substring, delete, and insert), and the observer and complete string methods. For example, the `String.replace(target,replacement)` can be implemented as

```
StringBuffer sb = new StringBuffer();
int start = 0;
int offset = this.indexOf(target);
while (offset != -1) {
  sb.append(this.substring(start,offset));
  sb.append(replacement);                              (10)
  start = offset + target.length();
  offset = this.indexOf(target,start);
}
sb.append(this.substring(start));
```

AutoRand re-implements each of the other non-core partial string methods in the same fashion.

Character methods Character methods convert (portions of) strings to their underlying characters, bytes, or code points (e.g., `toCharArray()`, `getChars()`, `getBytes()` and `charAt()`).

AutoRand derandomizes the string before making the conversion, preserving transparency. Since the result is not a string, random keys are not propagated (see Sect 4 for more information)

Miscellaneous Methods The `reverse()` method reverses the characters in a string. AutoRand derandomizes the string before making the conversion, preserving transparency. Propagation is not an issue as there are no single character keywords and thus keywords can not be transferred to the result. The `capacity()`, `ensureCapacity()`, and `trimToSize`() methods are not modified by AutoRand.

The `intern()` method returns a canonical representation for the string object. This is commonly used to conserve memory and also allows reference equality checks between interned strings. Since string constants are automatically interned and AutoRand modifies entries in the constant table, the randomized versions of constants are interned. This does not affect transparency unless reference equality is used to compare a constant with an interned input value. This kind of reference equality did not occur in any of the real-world programs used in the evaluation. Nonetheless, AutoRand could be extended to modify reference equalities (via the `if_acmp<cond>` bytecode) on strings to compare the (derandomized) contents of the strings if both sides of the equality test are interned.

Category	Methods
Complete	<init>, append, appendCP, concat, copyValueOf, toString, valueOf
Observer	compareTo*, contains, contentEquals, endsWith, equals*, hashcode, indexOf, isEmpty, lastIndexOf, length, matches, offsetByCPs, regionMatches, startswith
Partial	delete*, format, insert, replace* setCharAt, setLength, split, subSequence, substring, toLowerCase, toUpperCase, trim
Character	charAt, codePoint*, getBytes, getChars, toCharArray
Misc	capacity, ensureCapacity, intern, reverse, trimToSize

Fig. 2: Synopsis of approach for each string method (in `String`, `StringBuffer`, and `StringBuilder`). Similar calls (indicated with *) are grouped together as are calls with the same name but different arguments. CodePoint is abbreviated as CP. *Category* is the type of call for AutoRand instrumentation purposes. See sections 3.4, 3.4, 3.4, and 3.4 for more information.

3.5 External API Calls

Java strings can interact externally to the Java application through a number of Java system library calls. For example, writing to files/sockets, opening files, reading properties, reading environment variables, using reflection etc.

In these cases, the original strings should always be used. AutoRand accomplishes this by converting the application (and the system libraries themselves) to call AutoRand's version of these routines. These routines derandomize their string arguments and then make the original call. This ensures that each external call acts correctly and that the random key is never visible to an attacker (since it is always removed before any external communications).

3.6 Standard Java Library

Strings are also manipulated within the standard Java library. For example, the `equals()`, `compareTo()` and `hashCode()` methods are called in the collection classes. Commonly used classes such as `Pattern` and `Matcher` call string methods and create new strings. AutoRand instruments the libraries in the same manner as it instruments the application. This ensures that any string manipulations within the libraries will correctly propagate random keys and ensures transparency over any strings containing random keys. The only differences are that constant strings within the standard libraries are not randomized (as they will not flow to application SQL commands)

AutoRand statically transforms the byte code of the standard Java libraries and creates a new version of the library. When an application hardened by AutoRand is run, it is run with the transformed version of the library.

3.7 Extensibility

AutoRand could be easily extended to randomize, besides SQL keywords, other kinds of keywords in strings, to provide protection against OS command injection, LDAP injection, XQuery/XPath injection, etc.

4 Threats to Validity

Our current AutoRand implementation is transparent with the following exceptions: 1) AutoRand performs the randomization checks at SQL API calls to detect SQL injection attacks. The lack of transparency at these API calls is one of the goals of AutoRand. 2) Intern calls may not be transparent with respect to reference equality. This is straightforward to implement but not currently implemented (see Sect 3.4).

Our current AutoRand implementation satisfies propagation on all string operations over full keywords. However, there are some possible issues: 1) Converting strings to characters, bytes, or arrays thereof and back to strings. 2) Character-level manipulations that construct strings with SQL keywords (e.g., `"sel"` + `"ect"`). None of these occurred in any of the evaluation programs.

Because characters extracted from strings are manipulated as individual characters and not as strings, the randomization keys cannot be propagated for these methods. AutoRand thus derandomizes the string before making the conversion (preserving transparency). Fortunately, there is little reason to manipulate program constants in this fashion.

We evaluated this hypothesis experimentally by gathering information about how many times each character method is called in the evaluation programs on strings that contain randomization keys and the stack trace for each such call. We then examined each call to determine if it would pose a problem for propagation. Only strings that contain randomization keys are relevant to propagation.

The `getBytes()` method is called only in Ant and FTPS. In both cases it is used to prepare a string to be written to a stream. Strings that are written would be derandomized in any event (see Section 3.5) and are not an issue for propagation.

The `getChars()` and `toCharArray()` methods are called only by JMeter in a class that outputs XML (`PrettyPrintWriter`). Strings that are written out would be derandomized in any event (see Section 3.5) and are not an issue for propagation.

The `charAt()` method is called in 7 of the 8 evaluation applications. There were 12 unique call sites for `charAt()` on randomized strings in the seven applications. We examined each of these to determine how `charAt()` was being used and whether or not it was a problem for propagation. We found that these use cases for `charAt()` query the string for information, but do not use the resulting characters to build new strings. For example, the method `Selectorutils.tokenizePathAsArray` in Ant uses `charAt()` to look for slashes in the path. But the resultant array is built by normal string operations using the locations of the slashes as indices. Since AutoRand uses indices relative to the derandomized string, the offsets determined by querying `charAt()` are compatible.

The code point methods (e.g., `codePointAt()`) return full 32-bit character representations. Their usage would be similar to `charAt()` in programs that support the full Unicode set (and manipulate strings at the character level). These methods were not called in any of the evaluation applications.

None of the character methods were used to create new strings that are later used by the program. The examination of each use indicates that these do not present a propagation problem as they are commonly used. This validates our hypothesis that these calls are not used to manipulate strings but only to create specific output formats or to obtain information about the string. Propagation is not an issue in either case.

5 Experimental Evaluation

AutoRand has been experimentally evaluated using various Java programs.

5.1 Programs with Inserted Vulnerabilities

An independent test and evaluation (T&E) team hired by the government agency that is the sponsor of this research identified a set of Java programs, ranging in size up to 250k lines of Java source, not including common third-party libraries:

- Ant (256k LOC) —A build system.

Test Program	Variant	Database	Cnt	Attack/ Benign inputs	Test Program	Variant	Database	Cnt	Attack/ Benign inputs
Ant	V01	MySQL	43	2/5	JMeter	V05	MySQL	1	2/5
Ant	V02	Postgres	1	2/5	JMeter	V08	SQLServer	2	2/5
Ant	V03	MySQL	9	2/5	JMeter	V10	SQLServer	1	2/5
Ant	V04	MySQL	1	2/5	JMeter	V11	SQLServer	1	2/5
FTPS	V01	MySQL	41	2/5	Barcode	V05	MySQL	1	2/5
FTPS	V03	MySQL	1	2/5	Barcode	V06	Postgres	55	2/5
FTPS	V04	MySQL	13	2/5	Barcode	V07	Postgres	1	2/5
FTPS	V05	MySQL	1	2/5	HtmlCleaner	V06	Postgres	44	2/5
PMD	V01	MySQL	11	2/5	FindBugs	V08	SQLServer	39	2/5
PMD	V12	MySQL	14	2/5	FindBugs	V09	SQLServer	6	2/5
PMD	V13	MySQL	1	2/5	FindBugs	V10	SQLServer	1	2/5
SchemaSpy	V02	Postgres	1	2/4					

Fig. 3: Injected vulnerability programs and variants. Each variant is injected into the base program at *Cnt* different locations creating *Cnt* versions of the program. The attack and benign inputs are then applied to each version. For example, in the first row, 43 versions of Ant are created with the V01 vulnerability code inserted in a different location in each. Then 2 attack inputs and 5 benign inputs are applied to each of the 43 versions of Ant. AutoRand detects each attack with no false positives or semantic changes to the program.

- Barcode4J (28k LOC)—A barcode generator.
- FindBugs (208k LOC)—A bug finder.
- FTPS (40k LOC)—An FTP server.
- HtmlCleaner (9k LOC)—A reformatter of HTML files.
- JMeter (178k LOC)—A performance measuring tool.
- PMD (110k LOC)—A source code analyzer.
- SchemaSpy (16k LOC)—A database inspecting tool.

The T&E team introduced SQL vulnerabilities into each program, and produced a set of malicious inputs to exercise the vulnerabilities. The T&E team also produced a set of benign inputs to exercise each program's standard functionality. They created 13 vulnerability variants to insert into the base programs. Each test case inserts one of the variants into the base program. The same variant can be applied to multiple locations in a base program. See Fig 3 for details. As the figure shows, there are a total of 289 distinct test cases (base program + variant + injection location), 578 attack inputs, and 1444 benign inputs.

The malicious and benign inputs were sent to the program after hardening with AutoRand and the results observed to determine if the vulnerability was exploited in the case of malign inputs, and if functionality was preserved in the case of benign inputs. The inputs were also sent to the unaltered programs as a control. The AutoRand-hardened programs successfully blocked all of the attack inputs (i.e., injection attacks) and preserved functionality for all of the benign inputs.

Test Program	Lines of code	Database	Attack/ Benign inputs	Test Program	Lines of code	Database	Attack/ Benign inputs
TC	1055	MySQL	3/1	TC-3073	198	Hibernate	2/2
TC-3008	1723	MySQL	1/2	TC-3078	192	Hibernate	1/3
TC-3010	1680	MySQL	1/2	TC-3104	197	Hibernate	1/3
TC-3014	1166	MySQL	1/2	TC-3105	199	Hibernate	1/3
TC-3015	1127	MySQL	3/1	TC-3106	194	Hibernate	1/3
TC-3016	1055	MySQL	1/1	TC-3166	1221	MySQL	2/1
TC-3017	1780	MySQL	1/2	TC-3174	1298	MySQL	1/1
TC-3044	1054	MySQL	3/1	TC-3177	370	MySQL	1/1
TC-3045	1730	Postgres	4/1	TC-3178	315	MySQL	1/1

Fig. 4: SQL injection tests written by the T&E team. Each test applies the benign and attack inputs to the same SQL statement. AutoRand detected each attack with no false positives.

The experiments were run using the Test and Evaluation Workbench (TEW) developed by the T&E team. The TEW works on an interconnected set of virtual machines where variant creation, compilation, and instrumentation are performed on one machine and execution of test cases performed on separate machine(s). The tests were performed on Debian 6.03 and the virtual machines were run on a 12 core machine using Xeon 3.47Ghz processors. The TEW also includes support services such as the MySQL, PostgreSQL, SQLServer (Microsoft) and Hibernate database systems.

5.2 SQL Injection Test Programs

The same T&E team also wrote 17 small programs (see Fig 4) for the purpose of testing systems like AutoRand that protect against SQL injection. Each program reads inputs and uses them in SQL queries. The programs work as expected with benign inputs but are subject to SQL injection with malicious inputs. The tests covered the MySQL, Hibernate, and PostgreSQL database engines, a variety of SQL query syntax, and the `Statement.execute()`, and `Connection.prepareStatement()` Java SQL API calls.

Several different types of attack inputs were used across the tests including:

– **String Tautology** - Closing the application's quote of a string input early and then adding a tautology. For example one attack input is: `'OR'1'='1`. The resulting SQL is: `...password=''OR'1'='1'...` which will always be true (thus evading the password check).
– **Adding Code** - After a valid string or numeric input, additional code is added.
– **Comment out code** - After a valid string or numeric input, comment characters are added that stop processing of any remaining characters in the command. This can be combined with *Adding Code* to execute arbitrary commands.

Test Program	Variant	Runs	Overhead Percent
Ant	V01	25	4.6
Ant	V02	25	0.5
Ant	V03	25	2.2
Ant	V04	25	2.7
Ant	All	100	2.5
Barcode	V05	25	3.5
Barcode	V06	25	3.4
Barcode	V07	25	17.2
Barcode	All	75	8.1
FindBugs	V08	25	10.3
FindBugs	V09	25	11.1
FindBugs	V10	25	23.8
FindBugs	All	75	15.1
FTPS	V01	25	0.0
FTPS	V03	25	0.0
FTPS	V04	25	0.0
FTPS	V05	25	0.0
FTPS	All	100	0.0
HtmlCleaner	V06	25	6.8
HtmlCleaner	All	25	6.8
JMeter	V05	25	2.7
JMeter	V08	25	0.3
JMeter	V10	25	0.3
JMeter	V11	25	0.4
JMeter	All	100	0.9
PMD	V01	25	6.1
PMD	V12	25	7.2
PMD	V13	25	8.6
PMD	All	75	7.3
SchemaSpy	V02	20	3.4
SchemaSpy	All	20	3.4
All	All	570	4.9

Fig. 5: Overhead for test programs. One example of each program/variant was run (native and instrumented) five times over each of its inputs.

We hardened each program using AutoRand and executed the programs with each of their benign and attack inputs. The AutoRand-hardened programs successfully blocked every attack input while leaving behavior unchanged for every benign input.

5.3 Overhead

To measure the overhead incurred by randomization, we randomly chose one example test case from each program/variant combination (Fig 3) for a total of 23 applications.[6] We ran each over each of its benign inputs five times and measured the total wall clock time. We repeated this process with the hardened version of each variant and compared the times. The average overhead ranged from 0% for FTPS to 15.1% for FindBugs with an average of 4.9%. See Fig 5.

We also measured server overhead (a common use case for SQL injection defenses). OpenCMS [1] is an open-source Java program (consisting of over 100k lines of code) for managing web sites. It runs as a web application in the Apache Software Foundation's Tomcat framework [2]. It uses a database to store web site content and configurations. SQL injection attacks might thus be possible by sending customized URLs to the OpenCMS web application.

To measure the overhead incurred by randomization, a script was developed to send 1,000 benign URLs to an OpenCMS installation and record the resulting

[6] The full test suite runs in a special environment and is difficult to instrument. The subset allowed for more manageable experiments.

HTML responses. (The URLs were captured while interacting with the installation to manage a web site.) The total time required to process all of the URLs was measured both before and after hardening of the OpenCMS code by AutoRand. The average overhead was 4.5%. The recorded HTML responses were also compared to ensure that functionality was not altered.

The OpenCMS test was performed on a virtual machine running Ubuntu 12.04 on a 3.6Ghz 4 core iMac with 32 GBytes of memory. Both the client and the server ran on the same machine using localhost with negligible network delays.

6 Related Work

6.1 Manual Prevention

The most common approach to preventing SQL injection attacks is defensive coding practices such as carefully validating all inputs and using parameterized query APIs [26, 20]. Unfortunately, as evidenced by the continuing prevalence of successful SQL attacks [9], these practices have not been sufficient to prevent attacks.

Defensive coding practices require trained developers that *always* follow the correct approach. A single shortcut can lead to a vulnerability. And they can be very expensive and time consuming to apply to legacy code. And they provide no protection without access to developers and source.

AutoRand, by contrast, allows code to be immediately protected without source code modifications or developer involvement.

6.2 Randomization

SQLRand [5] introduced a manual method to randomize SQL queries. To apply the method, a developer finds each string containing SQL keywords, determines whether or not that string is used to build an SQL command, runs the string through the SQLRand tool, and copies the result back into their program. SQL requests are checked by a database proxy. Requests that do not contain the correctly randomized keywords will result in an exception. SQLRand does not derandomize SQL keywords except in the proxy. Thus, if the modified strings are used for any other purpose, changes to program semantics may result (including accidental disclosure of the randomization key). SQLRand does not support strings that are used for multiple purposes (e.g., SQL and error messages)

AutoRand automatically transforms the program to randomize SQL keywords and ensure semantic correctness (e.g., string length, accidental disclosures, etc.). In addition, AutoRand does not require any additional network components (i.e., a proxy).

6.3 Dynamic Tainting

A popular technique for preventing SQL injection attacks is dynamic taint tracking [14, 7, 19, 22, 24]. Taint-tracking systems instrument applications with the ability to track the provenance of inputs and are thus able to determine if an SQL query contains any untrusted inputs. Unfortunately, most taint tracking systems have either (*a*) non-negligible performance overhead [14, 7] or (*b*) reduce the scope of tracking they perform (i.e., they do not track character level information) that can lead to false positives and false negatives [19, 22].

Chin et al [7] implement a comprehensive taint tracking system (using character-level tainting), through modifications to the Java string library, that reports a modest overhead of about 15%. Unfortunately, their performance evaluation numbers do not include any safety checks using the taint information. Safety checks typically contribute significantly to the overhead of taint tracking systems. Furthermore, their evaluation does not test the system on real-world applications; they focus on unit tests designed to test taint propagation. Their implementation requires changes to the string library that are only compatible with the IBM JVM and does not support common string related functions, such as regular expressions and `String.format()`.

WASP [14] is a taint tracking system that tracks trusted, rather than untrusted, data. WASP uses its MetaStrings library to mimic and extend the behavior of Java's standard string classes. It replaces strings allocated in the application with the MetaStrings equivalent. WASP does not, however, instrument the Java libraries (except to remove the `final` flag from the string classes). Strings allocated within the Java library will thus not include meta-data. Any operations within the library that creates a new string based on application strings (such as those in `Pattern`, `Matcher`, and `Formatter`) will not propagate taint. Also, the string classes contain methods (e.g., `format()` and `split()`) that are implemented using these classes. Unless MetaStrings re-implemented these without using the libraries these may suffer from the same propagation issues. Propagation failures in WASP can lead to false positives. In contrast, AutoRand propagates random keys through the Java libraries and has less overhead.

WASP could be extended to instrument the system libraries to avoid these issues, but one would expect its overhead to be significantly increased.

Diglossia [24] tracks taint in PHP by modifying the interpreter to create a shadow string that uses a mapped character set for trusted characters. It then parses the shadow string and the original string to ensure that tainted input doesn't change the parse tree. PHP interpreter based approaches are not directly applicable to Java as the complex Java JIT makes it significantly more difficult to efficiently modify the interpreter. AutoRand's bytecode transformation approach is more portable and maintainable.

6.4 Parse Tree Structure

Another technique for detecting SQL injection attacks is based on the observations that most attacks modify the SQL query structure (i.e., parse tree) as intended by the developer [6, 25, 3, 4].

SQLGuard [6] and SQLCheck [25] are developer tools that can be used to statically define and dynamically check the integrity of SQL query structures. While successful at detecting a number of SQL inject attacks, they require manual modifications to the application. In contrast, AutoRand is fully automatic.

An alternative approach is to automatically learn query structure [13, 3, 4, 12]. AMNESIA [13] and Halder et al. [12] use static analysis to create a model of query structure and a run-time system to detect structure violations. To scale its static analysis to real-world applications, AMNESIA is context- and flow-insensitive and thus susceptible to false-negatives and false-positives [3]. AutoRand is a dynamic technique and hence not susceptible to the imprecision introduced by static analysis.

CANDID [3, 4] is a dynamic technique for extracting query structure. CANDID automatically transforms the application code to create a parallel, shadow data set for strings. Where the program assigns to a string variable, CANDID inserts code to assign to a shadow variable which will be used in the reference query. If the real variable is assigned a string constant, the shadow variable gets the same value. If the real variable receives a value from user input, the shadow variable gets a dummy value. String operations like concatenation are performed on both data sets in parallel. CANDID's published overhead is four times slower than AutoRand, most likely due to its added complexity.

6.5 Static Analysis

Several methods use static analysis to detect SQL injection attack vulnerabilities [16, 18, 11]. These systems identify unsanitized data flows from user input to SQL queries (i.e., they check whether every flow from input to query is subject to input validation). These techniques can verify that a sanitization technique is called on unsanitized flows but not whether the sanitization is correct, which can lead to false negatives. Given that static data-flow analysis must be conservative, these techniques, inescapably, also suffer from false positives.

7 Conclusion

SQL injection vulnerabilities comprise a prominent, serious, and ongoing source of security vulnerabilities. By delivering an automated, transparent, and efficient implementation of SQL keyword randomization, AutoRand provides one solution to this problem. Our results show that, on examples developed by an independent evaluation team, AutoRand, as designed, successfully blocked all SQL injection attacks and provided transparent execution for benign inputs, all with low overhead in large production Java applications.

Acknowledgements: We thank the MITRE Corporation test and evaluation team for creating an automatic and thorough testing apparatus. We thank Stephen Fitzpatrick and Eric McCarthy of Kestrel Institute for their contributions to the project. We thank Michael Gordon of Aarno Labs for comments that greatly improved the manuscript.

References

1. Alkacon Software: OpenCms. `http://www.opencms.org` (May 2012)

2. Apache Foundation: Apache Tomcat. `http://tomcat.apache.org/` (January 2012)
3. Bandhakavi, S., Bisht, P., Madhusudan, P., Venkatakrishnan, V.N.: Candid: Preventing sql injection attacks using dynamic candidate evaluations. CCS '07 (2007)
4. Bisht, P., Madhusudan, P., Venkatakrishnan, V.N.: Candid: Dynamic candidate evaluations for automatic prevention of sql injection attacks. ACM Trans. Inf. Syst. Secur. 13(2), 14:1–14:39 (Mar 2010)
5. Boyd, S.W., Keromytis, A.D.: SQLrand: Preventing SQL injection attacks. In: ACNS 2. Lecture Notes in Computer Science, vol. 3089, pp. 292–302 (2004)
6. Buehrer, G., Weide, B.W., Sivilotti, P.A.G.: Using parse tree validation to prevent sql injection attacks. SEM '05 (2005)
7. Chin, E., Wagner, D.: Efficient character-level taint tracking for Java. In: Proceedings of the 2009 ACM Workshop on Secure Web Services (2009)
8. Clarke, J.: SQL Injection Attacks and Defenses. Syngress, second edn. (2012)
9. Code Curmudgeon: SQL injection hall of shame. `http://codecurmudgeon.com/wp/sql-injection-hall-of-shame/`, fetched on 2014-06-24
10. Common Weakness Enumeration (CWE) 89: Improper neutralization of special elements used in an SQL command ('SQL injection'). http://cwe.mitre.org
11. Fu, X., Lu, X., Peltsverger, B., Chen, S., Qian, K., Tao, L.: A static analysis framework for detecting sql injection vulnerabilities. COMPSAC 2007 (2007)
12. Halder, R., Cortesi, A.: Obfuscation-based analysis of sql injection attacks. pp. 931–938. ISCC '10 (2010)
13. Halfond, W.G.J., Orso, A.: Amnesia: Analysis and monitoring for neutralizing sql-injection attacks. pp. 174–183. ASE '05 (2005)
14. Halfond, W.G.J., Orso, A., Manolios, P.: Using positive tainting and syntax-aware evaluation to counter sql injection attacks. SIGSOFT '06/FSE-14 (2006)
15. ISO/IEC 9075:2011 – Information technology – Database languages – SQL
16. Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). SP '06 (2006)
17. Kc, G.S., Keromytis, A.D., Prevelakis, V.: Countering code-injection attacks with instruction-set randomization. pp. 272–280. CCS '03 (2003)
18. Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in java applications with static analysis. pp. 18–18. SSYM'05 (2005)
19. Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., Evans, D.: Automatically hardening web applications using precise tainting (2005)
20. OWASP: Sql injection prevention cheat sheet. `https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet`, fetched on 2014-08-01
21. OWASP Foundation: OWASP Top Ten Project. https://www.owasp.org/index.php/Top_10_2013-Top_10 (Jun 2013)
22. Pietraszek, T., Berghe, C.V.: Defending against injection attacks through context-sensitive string evaluation (2006)
23. SANS Institute, MITRE, et al.: CWE/SANS Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25 (Sep 2011)
24. Son, S., McKinley, K.S., Shmatikov, V.: Diglossia: detecting code injection attacks with precision and efficiency. pp. 1181–1192. CCS '13 (2013)
25. Su, Z., Wassermann, G.: The essence of command injection attacks in web applications. pp. 372–382. POPL '06 (2006)
26. Veracode: Sql injection cheat sheet and tutorial. `http://www.veracode.com/security/sql-injection`, fetched on 2014-08-01