Comprehensive Java Metadata Tracking for Attack Detection and Repair

Jeff Perkins
MIT/CSAIL
jhp@csail.mit.edu

Jordan Eikenberry
MIT/CSAIL
jeikenberry@csail.mit.edu

Alessandro Coglio
Kestrel Institute
coglio@kestrel.edu

Martin Rinard
MIT/CSAIL
rinard@csail.mit.edu

***Abstract*—We present ClearTrack, a system that tracks metadata for each primitive value in Java programs to detect and nullify a range of vulnerabilities such as integer overflow/underflow and SQL/command injection vulnerabilities. Contributions include new techniques for eliminating false positives associated with benign integer overflows and underflows, new metadata-aware techniques for detecting and nullifying SQL/command command injection attacks, and results from an independent evaluation team. These results show that 1) ClearTrack operates successfully on Java programs comprising hundreds of thousands of lines of code (including instrumented jar files and Java system libraries, the majority of the applications comprise over 3 million lines of code), 2) because of computations such as cryptography and hash table calculations, these applications perform millions of benign integer overflows and underflows, and 3) ClearTrack successfully detects and nullifies all tested integer overflow and underflow and SQL/command injection vulnerabilities in the benchmark applications.**

***Index Terms*—Computer Security; intrusion detection; Computer errors; Software safety**

I. Introduction

Dynamic taint tracking has been implemented by many systems [35], [39], [55], [53], [26], [17], [22] to address a variety of security vulnerabilities such as SQL injection, command injection, and control-flow integrity [53], [26], [32]. We present ClearTrack, a new metadata tracking system for Java. ClearTrack rewrites Java bytecode to include instrumentation that tracks metadata about the flow of information through the program to make the following contributions:

Metadata: ClearTrack maintains 32 bits of metadata for every value that the program manipulates, enabling ClearTrack to track 32 distinct properties for each value simultaneously. Unlike previous taint tracking systems [26], [17], [42], [30], [53], [39], [35], ClearTrack tracks non-source taint metadata such as the type of integer arguments (numeric or bitwise), overflow status, divide-by-zero status, whether or not a value has been bounds checked, and encoding types (such as xpath, CSS, HTML, etc). Tracking these dynamic properties allows ClearTrack to detect and repair vulnerabilities that lie beyond the reach of other systems.

Overflow/Underflow Errors: ClearTrack tracks the status of integer values to precisely and accurately detect integer overflow and underflow errors. To avoid false positives, ClearTrack deploys the following new techniques:

- **Report Only On Dangerous Operations:** In contrast to previous systems, ClearTrack reports an error only when a value is used in a dangerous operation such as a memory allocation or conditional, not when the overflow occurs.
- **Numeric Type Inference:** ClearTrack introduces the concept of numeric and bitwise integers. Numeric values are constants and those used with arithmetic operators (plus, minus, etc). Bit-wise values are those used with bit operators (bitwise-or, bitwise-and, etc). Overflows occur only on numeric values. ClearTrack metadata is used to set and propagate a tag for numeric vs bitwise values.
- **Clearing Overflows:** ClearTrack identifies operations (such as bitwise-and) that safely clear the overflow or underflow status of a value. Identifying such operations enables ClearTrack to detect when the program clears the overflow or underflow (typically with a bit masking operation) and avoid false positives for legitimate values derived from overflowed or underflowed values.

The experimental results show that our benchmark applications contain millions of legitimate integer overflows and underflows (sources of these legitimate overflows include, for example, cryptography and hashing calculations) and that all of these new capabilities are critical for avoiding millions of false positives on these programs.

Repair: ClearTrack uses the metadata to automatically repair incorrect or malicious inputs to SQL and shell commands. ClearTrack implements a metadata aware tokenizer that ensures that any untrusted data is used properly only as a literal (string, numeric or keyword) within the SQL or command line string and does not contribute to the command syntax. Untrusted data in SQL commands that would otherwise escape the intended application quotes are properly escaped. Untrusted data in shell commands that would not be treated as a single command argument by the shell are forced to be a single argument. Previous systems, in contrast, at most signal an error when tainted data appears inappropriately in the command string [42], [26], [30], [53].

Path Traversal/Equivalence: ClearTrack includes a novel policy to detect pathname traversal attacks. A pathname traversal attack occurs when untrusted input can specify a pathname that specifies directories that are not within the directory intended by the application. ClearTrack automatically checks for both absolute and relative pathnames that escape the application root.

Evaluation: This research was sponsored by an anonymous agency of the United States government. ClearTrack was evaluated by an independent Test and Evaluation (T&E) team hired by the agency that sponsored this research at two separate red team exercises.

The independent T&E team selected 13 benchmark Java programs ranging in size from 9,000 to 540,000 lines of Java source code (Section III). Including the instrumented jar files and Java system libraries, the majority of the applications comprised over 3 million lines of code.

Working independently of the ClearTrack developers, the evaluation showed that, for the 1015 injected vulnerabilities evaluated with 2030 malicious inputs and 10,150 benign inputs, ClearTrack successfully nullified all 2030 presented attacks and correctly processed all 10,150 benign inputs with no false positives and no false negatives. Depending on the application, the ClearTrack overhead ranged from 7.7% to 96.6% with an average of 44.5% (Section III).

II. Implementation

ClearTrack tracks 32 bits of metadata for each value in the program. The metadata includes not only basic taint information, but provenance information (file, socket, command line, etc.) and, for integer values, information about their current state (normal, overflowed, underflowed) and how they are used (numerically or bitwise). The information propagates to appropriately tag newly computed values as they are derived from other values.

ClearTrack instruments the application, its libraries, and the Java system libraries, by modifying the bytecode to track metadata throughout the program. Each program variable has a paired metadata 32 bit integer variable. Each assignment and operation acts on both the primitive value and its corresponding metadata value.

- **Object Fields:** For each primitive object field, ClearTrack adds a corresponding metadata field. The metadata field is initialized to zero (no information). When a field is set, the metadata field is set as well, when a field is read, the metadata field is read as well. Each original field access instruction turns into two bytecode instructions.
- **Parameters and Locals:** For each original primitive parameter and local variable, ClearTrack adds a corresponding metadata parameter/local variable. ClearTrack modifies method interfaces to accept corresponding metadata values for each parameter.
- **Return Values:** ClearTrack maintains a single object, passed into each method, to carry the metadata for the return value from the callee to the caller. The caller extracts the metadata immediately after the call.
- **Stack Usage:** The Java Virtual Machine (JVM) is a stack machine, with all expressions and method calls implemented on the stack. ClearTrack instruments the bytecode as follows. Each time the program pushes a value (field, local, parameter) onto the stack, the instrumentation also pushes the metadata for the value (so that the value and its metadata form a pair on the stack). ClearTrack replaces binary operators (e.g., add, subtract, multiply) with code that manipulates the stack to perform the original binary operation and also the corresponding metadata join operation required to compute the metadata for the newly computed value. When the program executes, the Java JIT translates the stack operations into efficient register code.
- **Native Calls:** ClearTrack supports metadata origination at and propagation through native calls with method summaries that appropriately augment the computed values with metadata when the native call returns. Wrapper methods convert the stack representation from the augmented version used in the Java bytecode to the original Java stack representation. This allows the original native routine to be called normally. The wrapper is implemented in Java making it easy to create summaries and to originate/propagate taint. ClearTrack currently has 451 summaries. Summaries are used for both native calls and for optimization of calls whose taint propagation is expensive. Some summaries (75) are applied to the specified call and any call that overrides them.
- **Arrays:** ClearTrack replaces each primitive array with an object that contains an array storing the original values and a corresponding parallel array storing the metadata. Storing the values and the metadata in separate arrays allows the values to be passed through to native and uninstrumented routines without modification. As with fields and locals, each time an array element is accessed, the instrumentation also accesses the corresponding metadata.
- **Reflection:** ClearTrack automatically instruments all reflective calls to pass metadata correctly to and from the reflective call. Metadata values for primitive arguments and the return value metadata are automatically added to the array of arguments to the callee. Thus, from the point of view of the callee, the arguments and metadata are all passed in their normal manner and as with uninstrumented code, the reflective call is transparent to the callee. Upon completion of a reflective call that returns a primitive, the metadata for that primitive is extracted from the return metadata parameter in the same fashion as it is for non-reflective calls. Furthermore, all of the reflection calls that return information about fields and methods are modified to elide the metadata variables, ensuring that the instrumentation is transparent to the application.
- **System Libraries:** Java applications make heavy use of the system libraries. To ensure correct results, ClearTrack instruments the system libraries in the same manner as the application is instrumented.
  One complication is that the JVM sometimes directly invokes specific system library methods, with the original method type signature hardcoded into the JVM implementation. ClearTrack therefore generates two versions of each library method: the instrumented version (with an augmented type signature that includes the metadata parameters) and a version with the original type signature that simply invokes the instrumented version. This mechanism is critical for enabling the JVM to continue to

operate correctly in the presence of instrumented system library methods.

- **Translation Tables:** ClearTrack supports implicit data flow through translation tables by conjoining the metadata of the index with the metadata of the value.

ClearTrack provides 32 configurable metadata bits for each primitive Java value. In the current ClearTrack implementation these bits are configured as follows:

- **Bits 0-1:** These bits track trusted and untrusted, tracked separately. Values derived only from trusted sources (such as constants trusted files or databases) are classified as trusted. Otherwise it classified as untrusted.
- **Bits 2-14:** These bits track whether or not string values have been encoded for safety for various interpreters (XPATH, CSS, HTML, LDAP, SQL, etc).
- **Bits 15-26:** These bits track the source of the value (Java properties, database, environment variables, command line arguments, files, etc).
- **Bits 27-31:** These bits track the overflow and type of integer values (details in Section II-C).

A. Example

We next present a simple example that illustrates the ClearTrack instrumentation. Note that ClearTrack operates directly on Java bytecode, not Java source code. For readability, we nevertheless present the example at the level of the Java source code. Key concepts that the example illustrates, such as metadata fields, parameters, and return values and the additional instructions required to propagate the metadata, transfer directly to Java bytecode.

The example is a method that shifts a `Point` object right by an amount `dx`:

```
class Point {
  int x, y;
  int shift(dx) {
    x = x + dx;
    return x;
  }
}
```

The ClearTrack instrumentation adds an `int` field `x_t` and `y_t` for each of the `x` and `y` fields in the `Point` object. This additional field stores the metadata for the corresponding `Point` field. ClearTrack similarly adds a parameter `dx_t` to store the metadata for the `dx` parameter of the `shift` method.

Each method also takes an additional `rval` parameter to store the metadata for the return value of the method. The caller extracts this metadata from the `rval` parameter immediately when the invoked method returns.
The instrumented version would be

```
class {
  int x; int x_t;
  int y; int y_t;
  int shift (dx, dx_t, RetVal rval) {
    x = Instrument.iadd (x, x_t, dx, dx_t, rval);
    x_t = rval.metadata;
    return x;
  }
}
```

The instrumented code implements the integer add operation by invoking the `iadd` method in the `Instrument` class. The `iadd` method checks for overflow and underflow and sets the appropriate bits in the metadata return value if these conditions occur. The `iadd` method also propagates the metadata from the operands of the add through to the result.

```
class Instrument {
  int iadd(int v1, int v1_t,
           int v2, int v2_t, RetVal rval) {
    rval.metadata = v1_t | v2_t;
    long result = ((long) v1) + v2;
    if (result > MAX_INT_VALUE)
      ret.metadata |= OVERFLOW;
    else if (result (< MIN_INT_VALUE)
      ret.metadata |= UNDERFLOW;
    return (int) result;
  }
}
```

Applying the instrumentation at the bytecode level enables the Java JIT to productively optimize the metadata tracking code in conjunction with the base code from the application. Potential optimizations include placing metadata values in registers, unrolling loops, eliminating unnecessary checks, and inlining methods (such as the `iadd` method in our example). The result is an efficient register-based implementation with substantially less overhead (as a percentage of the total run time) than would be incurred by an implementation that simply interpreted the instrumented bytecode.

B. Optimized Method Summaries

ClearTrack also applies method summaries to optimize some expensive operations. For example, instrumenting Java I/O operations can incur substantial propagation overhead because they often include data intensive operations such as character encoding. Instead of instrumenting such methods, ClearTrack applies method metadata propagation summaries to implement the metadata propagation tracking at the granularity of method calls (rather than at the granularity of individual operations within the invoked method).

When a stream is created on an entity such as a socket or file, ClearTrack sets the metadata system's current configuration for trusted/untrusted data (which classifies each entity as trusted or untrusted). When the program creates a stream based on an existing stream (such as adding a `BufferedInputStream` to a `FileInputStream`), ClearTrack takes the trusted/untrusted status from the existing stream. If the bytes in the underlying stream have different metadata, ClearTrack invokes an instrumented version of invoked methods to ensure that the metadata is correctly propagated for each byte.

ClearTrack does not generate instrumented versions of many container class methods (examples include some `ArrayList` or `HashMap` methods) — because these methods are known to not access the underlying objects in the container, they do not change the metadata information of the objects in the container or propagate the metadata from the contained objects to the fields of the container object.

We generate the ClearTrack method summaries manually for standard methods with known behavior. In the absence of a summary, ClearTrack instruments the method.

C. Integer Overflow Errors

ClearTrack can precisely detect integer overflows (we use the term overflow to refer to both integer overflow *and* integer underflow errors; ClearTrack tracks both underflows and overflows) without any false positives in our benchmark evaluation programs. A key to the approach is that ClearTrack takes an action *only* when an untrusted overflowed value is used in a dangerous operation and not when an overflow occurs. Dangerous operations are

- Array indexing
- Conditionals (loops or if statements)
- Memory allocation.

Note that printing (to text) is not considered a dangerous operation. ClearTrack simply prints 'Overflow' for the value.

ClearTrack uses several metadata bits to track precise information about the status of each value. These are:

- **Overflow:** Indicates that the value has overflowed or was derived from an overflowed value.
- **Underflow:** Indicates that the value has underflowed or was derived from an overflowed value.
- **Bitwise:** Indicates that the value was generated from a bitwise computation or was derived from such a value. Set on a bitwise operation and cleared on a numeric operation. See Section II-E for more details.
- **Divide by Zero:** Indicates that the value was the result of a divide by zero operation or was derived from such a value. Set when a value is divided by zero. Note that ClearTrack changes the standard Java semantics to not throw an ArithmeticException on divide by zero. ClearTrack instead sets and propagates the divide by zero bit in the metadata. Unlike overflows, there are no bitwise or arithmetic operators that clear the divide by zero. Divide by zero values can, however, be safely printed (they will print as infinity). This behavior allows safe continued execution in the presence of divide by zero. It is also configurable.

Each numeric operation (e.g., add, subtract, multiply, divide, etc) is checked to determine if it overflows (see the `iadd` method in Section II-A). If so, the metadata for the value is marked as such. For all integer types except long, the overflow check is achieved by performing the operation in a higher precision type (e.g., long for integer) and checking the result to ensure it fits in the destination type. Longs are handled with operation-specific checks. For example, add operations overflow iff the sign of each operand is the same and is different from the result.

At each dangerous operation, the metadata for the value is checked and an exception is thrown if the value is overflowed or divide-by-zero.

The key to this approach is that the overflow can be cleared if an operation occurs that removes the overflow. Under Java's semantics, the bits that are present in an integer (the lower order bits) are correct when an overflow occurs. If operations that clear the higher order bits are executed on the value, the result will be exactly correct (as if arbitrary precision arithmetic were used on the value). Operations that clear the high order bits include, for example, masking operations and casting operations (e.g., integer to short). ClearTrack thus clears the overflow bits in the metadata when these operations occur. Figure 1 defines the semantics of the overflow field more precisely.

This approach ensures that ClearTrack does not interfere with common anticipated and desirable sources of overflows (cryptography, hashing, etc.) because the high order bits of the overflowed values are masked off before being used in dangerous operations.

ClearTrack can also optionally clear the overflow bit on mod operations (%) and logical right shift (>>>). The mod operation limits the size of the result in a manner very similar to masking. However, unless the divisor is a power of 2, the result will not be numerically the same as with arbitrary precision arithmetic. In all of the actual use cases we examined, however, mod was used to create a tag (such as a hash table index) where the exact numerical value was not relevant.

Logical right shift shifts in zeroes rather than sign extending the value. The results of this operation are specific to the size of the operand. Since bits beyond the bounds of the operand are being explicitly discarded, clearing the overflow flag on this operation is reasonable.

As described in section III-C, for untrusted values, only clearing on bitwise-and is required for correct operation over all of the evaluation programs. If trusted inputs are considered, then mod and logical right shift need to clear overflows as well on some of the programs.

E	O(E)	Comment
c	-	Constants are clean
V	V_o	The overflow field from the metadata for V
op(E)	O(E)	Unary arithmetic/bit *op*
E_1 *op* E_2	O(E_1) || O(E_2)	Most binary arithmetic/bit ops
E_1 *opc* E_2	-	Bit-and, mod, logical right shift
abs(E)	O(E)	Includes inline abs code
(l-cast)E	O(E)	Cast to larger type
(s-cast)E	-	Cast to smaller type

Fig. 1. Definition of overflow for expressions. Clearing overflow on mod and logical right shift is optional.

Normally conditionals are treated as dangerous operations on overflowed values. However, in some cases, the conditional and its block form a higher-level function over the value. In this case, rather than treating the conditional as a dangerous operation, ClearTrack simply marks the result of the combined operation as overflowed. This is analogous to the way that lower-level operations (such as add and multiply) are treated.

The most common example of this is code that is calculating absolute value. For example:

```
if (x < 0)
  x = -x
```

ClearTrack includes a template-matching component that detects commonly-used code patterns. Matching conditional branches and bodies are treated as a single functional block, and the conditional on the overflowed value is allowed. The resulting value is, however, still marked as overflowed.

The other templates (very similar) are:

```
if (x < 0)               if (x < 0)
  x = ~x;                  x = x * -1;
```

D. Integer Divide-by-zero Errors

Divide-by-zero errors are treated very similarly to overflow errors. The only difference is that the divide-by-zero metadata flag cannot be cleared (as there is no correct value for infinity). When such a value is used in a dangerous operation, an exception is thrown. When printed, divide-by-zero values print as 'Infinity'.

E. Integer Conversion Errors

When an integer value is cast to a smaller type (e.g., integer to short) or between signed and unsigned types (e.g., short and character) the original value may not be able to be represented in the destination type. For example, consider the following code:

```
char c = 0xFFFF;
short s = (short) c;
```

Since characters are unsigned, they can represent 0xFFFF (65535). Shorts are signed, so the same bit pattern is interpreted as -1. Problems can also occur when casting from a larger type to a smaller type (values too large for the smaller type will be truncated).

ClearTrack checks each integer cast operation and marks the result as overflowed if it is not equal to the original value. This check is appropriate if the values being cast are used as numbers.

In some cases, however, the values in integer variables are manipulated as bits. In this case, a different numerical result could be expected. Consider code that reads a short from a binary stream as two bytes and then forms them into short.

```
...
short readShort(InputStream is) {
  byte b1 = is.read();
  byte b2 = is.read();
  int value = (b1 << 8) | b2;
  return (short) value;
}
```

Within a Java expression, all (non-long) integer operations are performed as integers (32 bits). As an integer the result of `(b1 << 8) | b2` will always be a positive value (between 0 and 65535). When it is cast to a short, however, it will then have a possible range of -32768 to 32767. This would fail the overflow test (original value must match the result value). This conversion, however, is expected and the resulting value should *not* be marked as an overflow.

ClearTrack handles these situations by keeping track of the *type* of each value. Values can be either *bitwise* or *numeric*. Bitwise values are those that are created with bit operators (bitwise-and, bitwise-or and compliment). Numeric values are constants and values created with the numeric operators (plus, subtract, minus, divide, and mod). The check for overflow on a cast is only applied to numeric values.

F. Injection attacks

Injection attacks can occur when the program sends text commands that include untrusted inputs to external subsystems (such as SQL servers, command shells, etc). Because the commands are parsed externally, attackers may be able to inject text that subverts the intention of the application. For example, consider a program that looks up some contact information by running the grep program on a file. The application might execute the following command using bash (where the untrusted data is underlined) and send the results to a remote user.

```
grep -i david ~/contacts
```

Rather than entering a name as expected, an attacker could enter 'david; cat /etc/passwd' yielding the following command:

```
grep -i david; cat /etc/passwd ~/contacts
```

This would send the contents of /etc/passwd and the contact file to the attacker.

Injection problems can be addressed by ensuring that all untrusted inputs are limited to numeric and string literals and language-specific reserved values (for SQL, true, false, null, etc). This restriction ensures that the untrusted input cannot change the meaning of the command.

ClearTrack contains a novel *metadata-aware tokenizer* that considers both the character and its metadata when tokenizing. ClearTrack adjusts the tokenization to ensure that untrusted data follows the above policy for untrusted inputs. When processing trusted characters, the metadata-aware tokenizer acts in the same manner as a traditional tokenizer. It processes untrusted data as follows.

First, when processing string literals (quoted strings), the tokenizer includes all characters starting with the initial application (trusted) quote character up to the enclosing application quote character. Untrusted quote characters do *not* terminate the quoted string. Consider the following example:

```
String sql = String.format
  ("...where ... and passwd='\%s'", passwd);
```

Regardless of the input values for `passwd`, the tokenizer will treat them as a single quoted string (surrounded by application quotes).

Second, when tokenizing tokens other than string literals, the token terminates only on a trusted character. This ensures that the untrusted input is treated as a single entity. Consider the command injection example earlier in this section. In the command

```
grep -i david; cat /etc/passwd ~/contacts
```

the ClearTrack metadata-aware tokenizer produces the following tokens: ’grep’, ’-i’, ’david; cat /etc/passwd’, and ’ /contacts’.

ClearTrack detects injection attacks only at the point where the command is executed. This ensures that the check and possible repair are subsystem (e.g., SQL, command shell) specific. It also allows the same untrusted input to be passed to different subsystems (with possibly different repairs).

G. SQL Injection Detection and Repair

In Java, SQL commands are executed via the `execute()` methods in the `sql.Statement` class, the `prepare()` methods in the `sql.Connection` class and the `createQuery()` call in the `org.hibernate.Session` class. ClearTrack checks queries at each of these calls using the metadata-aware tokenizer to analyze the input.

With metadata-aware tokenization, an SQL injection attack is present if either of two conditions hold. In string literals, an attack is present if there are unescaped special characters (quotes or backslashes) within the string literal (between the two trusted application quotes). ClearTrack repairs these attacks by properly escaping the characters. This ensures that the input does not change the meaning of the command so that the application executes correctly for both malicious inputs and valid inputs that happen to contain embedded special characters (e.g, O’Malley).

In other tokens, an attack is present if the token is neither a valid SQL numeric constant nor a SQL reserved constant. Consider the following example (where untrusted input is underlined):

```
select * from table t
  where value=123;drop table t
```

In these cases, ClearTrack could be configured to repair the input by discarding the invalid value characters (in this case: `; drop table t`). However, unlike quoted string repairs, this risks database interactions that were not intended by the user (if, for example, the invalid characters were simply a typo). Thus, by default, ClearTrack throws an exception when it finds an invalid input in a non-quoted string.

We tested ClearTrack on the 11 canonical examples of injection attacks described by Ray and Ligatti [36]. Our taint-aware tokenizer approach matches their suggested results for 10 of their 11 examples. The one example with a different result is:

```
create table t (name CHAR(40))
```

Ray and Ligatti classify this as code injection. We do not as we consider integer literals, even in SQL type definitions, to be values and thus not an attack.

In a later paper [37], Ray and Ligatti describe non-code injection attacks. These are injections that, while not code themselves, change the interpretation of later portions of the command. For example a trailing backslash can remove an application quote and thus change the interpretation of the rest of the command. Because ClearTrack checks for all special characters (including backslashes) within trusted application quotes it correctly handles and repairs this example as well. Essentially, any user input is constrained to remain within the trusted application quotes achieving the same result as parameterized queries without requiring program modifications.

Using the taint-aware tokenizer allows ClearTrack to work effectively without requiring a full SQL parser. This allows it to support multiple dialects of SQL and versions for which the full grammar may not be available.

H. Command Injection Detection and Repair

Command injection is treated in a similar fashion to SQL injection except that the grammar is that of shell commands (such as bash). In Java shell commands are executed by the `Runtime.exec()` calls or the `ProcessBuilder` class using the `-c` command line option to bash (or other shells). For example:

```
bash -c 'grep -i david ~/contacts'
```

Similarly to SQL injection, the policy for command injection is that untrusted input should only specify single arguments to the command. Untrusted input should also never specify a command itself.

Command injection attacks are automatically repaired so that the command executes as intended by the application. As with SQL injection, any untrusted special characters are properly escaped so that they do not escape the trusted application quotes. If application quotes are not present (perhaps because the application presumed the untrusted input was a single token) ClearTrack also automatically repairs commands for which application quotes are not present (perhaps because the application presumed the untrusted input was a single token) by quoting contiguous tokens that contain untrusted input. For example, the input:

```
grep -i david; cat /etc/passwd ~/contacts
```

is modified to:

```
grep -i 'david; cat /etc/passwd' ~/contacts
```

This passes the entire input value (david; cat /etc/passwd) to grep as a single argument. This approach also fixes inadvertent input errors such as embedded blanks. For example, the untrusted input ’David Smith’ would also be quoted ensuring that it is treated as a single argument.

I. Pathname Traversal

ClearTrack includes a policy to detect pathname traversal attacks. A pathname traversal attack occurs when untrusted input can specify a pathname that specifies directories that are not within the directory intended by the application. This can occur with absolute pathnames (those that begin with slash) and with relative pathnames that reference parent directories (using `../`).

Absolute pathnames are straightforward to detect. The leading slash should not be specified by untrusted data. Relative pathnames are more complex. Consider a simple FTP server that allows the user to traverse the user’s directory tree, but does not intend the user to be able to enter a different user’s tree. User fred should be able to specify a directory of

```
fred/root/dir1/../dir2
```

where, as shown, the application specifies the root, and the user can use `..` to traverse the tree under the root. However, the following input is not valid because it leaves the users directory tree and enters a different users tree:

```
fred/root/../../david/root/dir1
```

Simply disallowing parent directory references would not allow valid traversals within the users directory tree. ClearTrack implements a more precise policy that allows only valid traversals. Untrusted parent directory references can only reference directories below the trusted root of the path. In this case, the trusted root is 'fred/'. If there is no trusted root, a trusted './' is added to the beginning of the reference.

Note that Java does not provide a call to change the current working directory. All pathnames must be either absolute or relative to the initial working directory.

III. Experimental Evaluation

The United States government hired an independent test and evaluation (T&E) team to evaluate ClearTrack [47]f. We next present the results of this evaluation.

A. Applications and Vulnerabilities

The T&E team identified a set of Java applications (see Figure 2) to drive the evaluation. The applications range in size from 9K to 540K lines of Java source code. These numbers do not include classes from external jar files or the Java system libraries, which ClearTrack also instruments. Including these classes, most of the applications comprise over 3 million lines of code each.

The T&E team inserted vulnerabilities [45] into the 5 largest programs and developed malicious inputs to exercise the vulnerabilities. The vulnerabilities were chosen to cover a range of scenarios over five axes: data source (environment, file, socket), data type (primitive, array, reference), data flow (constant, return value, basic, index alias, Java generics, variable arguments), control flow (labeled break, callback, overloaded functions, recursion, indirect recursion, infinite loop, call depth (1, 2, 10, 50), interrupt, sequence) and vulnerability type (SQL injection, command injection, path traversal, numeric overflow, numeric underflow, unexpected sign extension, signed to unsigned conversion, unsigned to signed conversion, numeric truncation). To test for false positives, the T&E team also developed benign inputs to exercise the standard functionality of each program.

Figure 3 summarizes the resulting inputs. There are a total of 1015 distinct vulnerabilities, 2030 malicious inputs, and 10,150 benign inputs. Each of the benign inputs exercises a different application setup with different arguments to exercise a distinct execution path.

The T&E team also developed a Test and Evaluation system (TEXAS) [46], [48], which comprises an interconnected set of virtual machines to compile and execute the benchmark applications. TEXAS also includes required support services such as the MySQL, PostgreSQL, SQLServer (Microsoft), and Hibernate database systems.

B. Application Executions

Working with the delivered ClearTrack system independently from the ClearTrack development team, the T&E team produced two versions of each application for each vulnerability. The *unprotected* version executes in the standard Java execution environment. The *hardened* version executes with the ClearTrack metadata tracking, vulnerability detection, and repair enabled. The T&E team next ran the unprotected versions of each application on the malicious and benign inputs to verify that the malicious inputs successfully triggered the vulnerability and that the benign inputs produce the expected correct results.

The T&E team next ran the hardened versions of each application on the malicious and benign inputs. The results show that ClearTrack successfully nullified all of the exercised vulnerabilities for all of the malicious inputs and generated no false positives on any of the benign inputs. All executions were performed on Debian 6.03 with the virtual machines executing on a 12 core machine using Xeon 3.47Ghz processors. We next discuss each class of vulnerabilities in more detail.

Program	Application Lines of Code
Ant[4]	256K
Barcode4J[28]	28K
FindBugs[34]	208K
FTPS[5]	40K
HtmlCleaner[52]	9K
JMeter[7]	178K
PMD[19]	100K
SchemaSpy[18]	16K
CoffeeMUD[56]*	537K
Elastic Search[21]*	297K
Apache Jena[6]*	377K
Apache Lucene[8]*	440K
Apache POI[9]*	292K

Fig. 2. Evaluation applications and their lines of code. Those marked with a star (*) were injected with vulnerabilities

Vulnerability	Coffee MUD	Elastic Search	Apache Jena	Apache Lucene	Apache POI
Path Traversal	38	37	43	39	38
Path Equivalence	15	14	10	14	15
Command Injection	18	23	17	22	19
SQL Injection	61	57	63	58	60
Numeric Overflow	27	29	22	25	26
Sign Extension	9	6	11	11	7
Signed to Unsigned	7	10	11	8	11
Unsigned to Signed	11	7	10	8	8
Numeric Truncation	9	9	9	9	9
Divide by zero	9	9	8	8	11
Totals	204	201	204	202	204

Fig. 3. Number of test cases by vulnerability and base program

C. Numeric Vulnerabilities

The T&E team inserted numeric vulnerabilities in the following CWE (Common Weakness Enumeration) categories [1]. Each reads a value from an untrusted source that can trigger the vulnerability.

- Numeric Overflow (CWE 190) and Underflow (CWE 191): Performs an operation that overflows or underflows an integer type, then uses the result in a dangerous operation.
- Unexpected Sign Extension (CWE 194), Unsigned to Signed Conversions (CWE 195), Signed to unsigned Conversions (CWE 195): Performs operations that move data between signed and unsigned (character in Java) types such that the destination does not match the source.
- Divide by Zero (CWE 369): Bypasses the standard Java implementation to divide without checking for zero.
- Numeric Truncation (CWE 197): Unexpected truncation that occurs when casting to a smaller type.

With the exception of divide by zero, all of these issues result in a value that does not fit correctly within the size of the destination type. For simplicity, we refer to all of these vulnerabilities as overflows.

We compare the ClearTrack implementation with an implementation that triggers an error immediately when an overflow occurs. We distinguish two cases: when the overflow occurs in a *trusted* value derived from internal program values and when the overflow occurs in an *untrusted* value derived (at least in part) from user input.

The Overflow columns in Table 4 present the number of such overflows that occur in each of the applications when running on the benign inputs. As these numbers show, all of the applications encounter significant numbers of overflows even under normal execution. All of these overflows would be false positives under implementations that trigger an error immediately when an overflow occurs.

The next columns (Dangerous) present the number of times that an overflown value (or a value derived from an overflown value) is used in a dangerous operation *disregarding operations such as bitwise and, logical shift, or mod that eliminate the overflow*. The remaining columns present how many values survive when each of these operations is taken into account. So, for example, all of the 27.6K uses of trusted overflown values in CoffeeMUD are actually cleared by a bitwise and before they are used in a dangerous operation. Similarly, the 19.9M uses of trusted overflown values in Elastic Search that are not cleared by either bitwise and or logical shift are cleared by a mod operation before they are used in a dangerous operation.

These results show that ClearTrack correctly determines that no overflown value reaches a dangerous operation for any of the benign inputs — in other words, ClearTrack has no false positives on these inputs.

D. SQL Injection

As shown in Figure 3 there were 299 inputs for SQL injection vulnerabilities, 2 malicious and 10 benign inputs for each vulnerability. These inputs cover the MySQL, PostgreSQL, and Hibernate databases. ClearTrack detected all of the malicious inputs with no false positives on the benign inputs. With repair enabled, ClearTrack successfully repaired all of string attacks triggered by the malicious inputs to enable successful continued execution.

E. Command Injection

As shown in Figure 3 there were 99 inputs for command injection vulnerabilities, 2 malicious and 10 benign inputs for each vulnerability. Each input included untrusted input as part of the issued command string. ClearTrack detected all of the malicious inputs with no false positives on benign inputs.

With repair enabled, ClearTrack modified all of the command strings so that the command could execute with no negative impacts. In most cases, the modified command simply returned an empty data set. For example, one attack looks like:

```
find . -iname "*" -a -exec cat /etc/passwd;
```

ClearTrack ensures that the user input is treated as a single value by quoting it as follows:

```
find . -iname '"*" -a -exec cat /etc/passwd;'
```

In this case, the malicious input will all be treated as the argument to `-iname` and will simply not match any files. The program will continue to operate correctly.

F. Path Traversal

As shown in Figure 3 there were 195 path traversal tests and 68 path equivalence tests. A path equivalence (CWE-41) attack uses special characters in file names to make equivalent filenames that will not be treated as equivalent. For example the filename `execute.jsp` and the filename `execute.jsp/` refer to the same filename but may not be treated as equivalent. Malicious patch equivalence inputs are often designed to avoid black list checks. For example, a check that rejects requests for files that end in `.jsp` might not detect the same file when named with a trailing slash.

ClearTrack detected all of the malicious inputs with no false positives on the benign inputs. Each test included untrusted input as part of the pathname.

G. Overhead

To measure the execution time overhead, we measured the execution time of each of the applications with and without the ClearTrack instrumentation (with the exception of CoffeeMUD, whose custom client precludes accurate execution time measurements). The experiments executed each application 100 times for each set of benign inputs, measuring the total wall-clock time for each execution. We repeated this process with the hardened version of each program and compared the times. As shown in Figure 5 the execution time overheads range from 7.7% for JMeter to 96.6% for Apache Lucene with a mean overhead of 44.5% across all applications.

We also measured server overhead on OpenCMS [2] (an open source content manager with over 100k lines of code). OpenCMS runs as a web application in the Apache Software Foundation's Tomcat framework [3]. It uses a database to store web site content and configurations. ClearTrack instrumented both the OpenCMS application and Tomcat (another 340K lines of code).

	Trusted values					Untrusted values		
App	Overflow	Dangerous	bitwise-and	logical shift	mod	Overflow	Dangerous	bitwise-and
CoffeeMUD	68.0K	27.6K	0	0	0	16.7K	1.03K	0
Elastic Search	41.2M	51.1M	28.2M	19.9M	0	8.33M	9.72M	0
Apache Jena	15.8K	0	0	0	0	5.78K	5.76K	0
Apache Lucene	19.6M	9.07M	2.21K	0	0	84.7K	878K	0
Apache POI	3.92K	0	0	0	0	345	139K	0

Fig. 4. Operations that overflow and overflowed values that reach dangerous operations for trusted and untrusted values. The *bitwise-and*, *local shift*, and *mod* columns indicate how many overflows reach dangerous operations when the overflow bit is cleared on those operations. Results are totaled over runs on each of the ten different application setups.

Program	Number Inputs	Average Overhead
Ant	5	33.6%
Barcode4J	1	45.7%
FindBugs	2	54.4%
HtmlCleaner	1	33.0%
JMeter	1	7.7%
PMD	1	62.0%
SchemaSpy	1	22.8%
Apache Jena	10	37.6%
Apache Lucene	10	96.6%
Apache POI	10	55.8%
Elastic Search	10	51.5%

Fig. 5. Overhead percentage over 10 runs of each program. As indicated, some of the programs were run over multiple input sets. Results over different input sets are averaged together.

We measured the ClearTrack overhead with a script developed to send 1,000 benign URLs to an OpenCMS installation and record the resulting 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 ClearTrack. The average overhead was was 13.8%. A comparison of the recorded HTML responses shows that ClearTrack did not alter the functionality.

We performed OpenCMS experiments 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 for negligible network delays.

The average overhead is significant, but the ramifications of security errors are dangerous enough to justify substantial overhead at least in security sensitive applications. Additionally, servers (a common use case) are often not CPU bound and the overall impact on the load might well be significantly less (as seen in the OpenCMS experiment above). The system is also not heavily optimized and significant performance improvements via common compiler optimization techniques such as specialization are likely possible. For example, an Android provenance system [25] was able to reduce metadata overhead to 14% with optimizations such as loop/method specialization and array aggregations. Also, Iodine [10] was able to achieve an overhead for DIFT of 9% on C programs using an optimistic hybrid analysis. While neither are directly applicable to ClearTrack (because ClearTrack modifies metadata on numeric operations), they illustrate that further significant optimizations are likely possible.

IV. RELATED WORK

A. Integer Overflows

1) Run-time Instrumentation for Overflow Errors: IOC [20] dynamically instruments C/C++ code to identify possible overflow errors. IOC signals overflows as they occur and not when they are used in dangerous operations. The results show that the vast majority of overflows that IOC found are false positives. RICH [12] compiles C programs to dynamically check integer operations for overflows. Like IOC, it signals overflows as they occur and the majority of the detected overflows are false positives.

Because ClearTrack signals an error only when overflowed values are used in dangerous operations, and because ClearTrack correctly clears the overflow flag on operations such as `bitwise-and`, it supports legitimate uses of overflows in operations such as hashing, cryptography, and random number generation. Our results show that programs generate millions of legitimate overflows, all of which, in the absence of techniques such as those deployed in ClearTrack, would be false positives.

SoupInt [49] diagnoses exploited integer overflow from captured attack instances and automatically generates patches to fix the vulnerability. Unlike SoupInt, ClearTrack does not require attack instances and addresses all numeric vulnerabilities in the program automatically.

IntPatch [54] and IntTracker [43] insert overflow checks only at operations that they can statically determine (via type or static analysis) may flow to memory allocations. They argue that such operations do not have benign overflows. Both analyses are conservative so they may check operations that do not lead to memory allocations. And some programs may perform operations with benign overflows even in allocation size calculations. Either condition may lead to false positives.

ClearTrack differs from both IntPatch and IntTracker in that it tracks all overflowed values precisely at run-time with no false positives. This approach enables it to check a variety of possibly dangerous operations (such as comparisons and array indexing) as opposed to only memory allocations. Since benign overflows are certainly possible in non-memory allocation cases, the IntPatch and IntTracker approach cannot be applied to them.

2) Symbolic Test Generation: A number of tools [13], [14], [23], [24], [27], [29], [40], [50] employ various forms of symbolic test generation (e.g., concolic testing) to look for possible overflow bugs. By design, these tools can only uncover overflow problems on paths they are able to fully explore. Fully analyzing complex significant real-world programs is beyond the current state of the art.

DIODE [41] focuses on overflowed values that are used in memory allocation sites. It looks at memory allocation sites that are exercised by its seed inputs and can either find an input value that triggers an overflow or show that there is no input that would trigger an overflow for the observed target expression at that site. It is, however, limited to the seed-input/allocation site combinations exercised by its test inputs.

By using run-time instrumentation, ClearTrack, however, has the capability to catch all overflows that lead to dangerous operations without incurring false positives.

3) Static Analysis: Several static analysis tools have been proposed to find integer overflow and/or sign errors [15], [38], [51], [16], [31]. KINT [51], for example, generates constraints from source code and user annotations to determine if an integer error can occur. A substantial number of false positives still exists. KINT also proposes *NaN integers* that track whether or not they have overflowed. Unlike ClearTrack, NaN integers require a dedicated NaN value (which could occur normally) and have no facility to clear overflowed values (which is critical to resolving false positives (see Section III-C). IntPTI [16] uses a type analysis to find errors and generate possible fixes for C programs that are evaluated by users for correctness. While IntPTI has good results, it still has both false negatives and false positives and unlike ClearTrack requires users to verify its results. IntREPAIR [31] uses static symbolic execution to find possible overflows. Unlike ClearTrack, it detects *all* overflows and does not attempt to determine which ones are intentional. Users of the tool must determine the validity of each overflow.

B. Java Taint Tracking Systems

Phosphor [11] is a taint tracking system for Java that, like ClearTrack, tracks 32 bits of metadata for each primitive Java value. The reported overhead numbers for the two systems are similar. Unlike ClearTrack, Phosphor only implements source taint/propagation and does not implement any actions based on taint information. In particular, it does not identify, track, or clear numeric errors such as overflows or underflows. It also does not identify or repair SQL or command injection attacks.

WASP [26] is a taint tracking system for Java strings that tracks trusted, rather than untrusted, data. Trusted data comprises string constants and strings derived from configuration files. 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.

Chin et al. [17] modify the Java String classes to implement a string taint tracking system that distinguishes trusted from untrusted data. The modified String classes are compatible only with the IBM JVM and do not support common string related functions, such as regular expressions and `String.format()`.

Unlike WASP and Chin et. al., ClearTrack tracks all Java primitives, not just strings. ClearTrack can therefore accurately track metadata through character primitives, character arrays, byte arrays, integer types, and transfers between strings and other types. It also tracks 32 bits of metadata instead of a single trusted bit. WASP, unlike ClearTrack, does not track strings created in Java libraries. Unlike Chin et. al., ClearTrack is compatible with standard JVMs.

TaintDroid [22] tracks 32 bits of taint information with a modified Android VM at the level of primitives, strings and arrays (conflating the taint over array elements). TaintDroid is designed to track information leaks specifically and sets the 32 bits to track different information sources. It does not support detection of integer overflow errors. Because it conflates the character taint within strings (it records the same taint for all characters within the same string), it does not support the detection of SQL or command injection attacks (detecting these attacks requires maintaining character-level metadata within each string). Users must flash custom-built firmware to their device to use TaintDroid.

TaintART [44] applies an approach similar to TaintDroid but applies it to newer versions of Android that use the Android RunTime (ART) environment. ART uses ahead-of-time compilation for Android applications. TaintART modifies the ART compiler to track taint. Like TaintDroid it conflates string and array elements for efficiency.

Unlike TaintDroid and TaintART, ClearTrack precisely tracks metadata for all primitive values, including characters within strings and elements within arrays. ClearTrack therefore supports the detection of SQL and command injection attacks. ClearTrack also detects integer errors such as overflows. Because ClearTrack instruments the bytecode, instrumented applications execute without modification on new versions of standard JVMs. TaintDroid and TaintART, in contrast, work only with their modified Android VM and must be manually ported to new Android VMs (and specific devices) as they are released. The latest version of Android supported by TaintDroid is 4.3 (released in July 2012 and no longer supported). TaintART supports Android 5.0 and 6.0. The current version of Android is 8.X.

C. Other Taint Tracking Systems

There are a numerous systems that implement run-time taint tracking on various platforms (discussed in more detail in the following subsections). However, these systems are either limited in the number of bits they track, the precision of their tracking (for example conflating all elements of the same array), cannot be applied to Java, have significant overhead, or some combination thereof.

1) Binary systems: There are a number of taint tracking systems for binaries. These suffer from high overhead (300% or more) and/or cannot be applied to systems (such as the

JVM) that use JITs to dynamically compile code. Even when they can handle dynamic code creation, support for that increases the overhead of the system.

LIFT [35] is a binary taint tracking system built on a dynamic binary instrumentation tool (StarDBT). It has overhead of approximately 360% on SPEC INT2000 applications. It was not tested on systems (such as Java) that include dynamically generated code.

Saxena et al. implement a binary taint tracking system [39] at the byte level that has approximately 100% overhead. It cannot be applied to dynamically compiled Java code as it relies on static binary rewriting to add instrumentation to binaries.

EMS64 [55] is a binary system for memory shadowing system on 64 bit architectures. When configured to support 8 bits of shadow memory for each program byte, it has approximately 300% overhead (including shadow memory propagation).

None of these systems detect or track overflowed value, perform repairs on injection attacks, or handle path traversal attacks.

2) C Source systems: Xu et al. implement a C source taint tracking system [53] with overhead from 61% to 106% on non-server programs.

Iodine [10] also implements a C source taint tracking using an optimistic hybrid analysis that uses profiles to optimize paths based on likely run-time invariants over taint. It is sound as it can revert to a conservative policy if the invariants are violated. Iodine achieves an overhead of 9%.

These (and other similar systems) cannot effectively be directly applied to the JVM because they do not support dynamic code creation. However, at least some of the optimizations used by Iodine could be utilized in bytecode instrumentation.

Neither of these systems detect or track overflowed value, perform repairs on injection attacks, or handle path traversal attacks.

3) PHP string systems: Mui et al. implement complementary encoding [30] to track user input in PHP. Untrusted characters are encoded differently from trusted characters. The system is implemented by modifying the PHP interpreter to treat both versions of the character as the same except at check points. The system is effective at maintaining character taint as long as characters are not manipulated at a low level (such as encoding them to/from byte arrays, using less-than or greater-than comparisons, or using characters as array indices). The system supports SQL queries and cross-site scripting.

Diglossia [42] tracks string taint in PHP as a shadow string. Trusted characters in the shadow string are mapped to different characters (somewhat similar to complementary encoding). The system applies checks by ensuring that untrusted input is not included in the query. Diglossia supports database queries in SQL, JSON, or JavaScript.

Neither of these systems supports integer values or translations to/from bytes and byte arrays, nor do they detect or track overflowed values or repair attacks.

D. Injection Attacks

1) SQL Injection: SQL injection attacks have been addressed by a number of taint tracking systems on a variety of platforms including Wasp [26](Java), Diglossia [42] (PHP), Mui et al. [30] (PHP), and Xu et al. [53] (C).

ClearTrack is the only system, of which we are aware, that automatically repairs queries.

2) Command Injection and Path Traversal: Nguyen-Tuong et al. implement a PHP taint tracking system [33] by modifying the PHP interpreter. It addresses command injection by disallowing some system calls that contain *any* tainted data. This approach would not allow legitimate commands that contain tainted arguments.

Xu et al. implement a C source taint tracking system [53] that supports policies that can be used to address command injection and path traversal. The policies, however, are too simple to precisely define command injection or path traversal. The policy only rejects the presence of dangerous operators and does not support quoted strings or other more subtle restrictions.

ClearTrack is the only system, of which we are aware, that automatically repairs commands and handles path traversal attacks.

V. Conclusion

We present ClearTrack, which precisely and efficiently tracks metadata on all primitive values in Java programs. Results from an evaluation performed by an independent test and evaluation team hired by the United States government demonstrate ClearTrack's effectiveness in leveraging the tracked metadata to detect and nullify malicious inputs without false positives or false negatives on the benchmark applications and inputs. ClearTrack also automatically repaired SQL string injection and command injection attacks.

References

[1] Mitre cwe list. https://cwe.mitre.org.

[2] Alkacon Software. OpenCms. http://www.opencms.org/en/download/, May 2012.

[3] Apache Foundation. Apache Tomcat. http://tomcat.apache.org/, September 2015.

[4] Apach Foundation. Apache Ant. http://ant.apache.org/, July 2015.

[5] Apach Foundation. Apache FtpServer. http://mina.apache.org/ftpserver-project/, July 2015.

[6] Apach Foundation. Apache Jena. https://jena.apache.org/, November 2015.

[7] Apach Foundation. Apache JMeter. http://jmeter.apache.org/, November 2015.

[8] Apach Foundation. Apache Lucene. http://lucene.apache.org/, September 2015.

[9] Apach Foundation. Apache POI. https://poi.apache.org/, September 2015.

[10] S. Banerjee, D. Devecsery, M. P. Chen, and S. Narayanasamy. Iodine: Fast dynamic taint tracking using rollback-free optimistic hybrid analysis. *ieee symposium on security and privacy*, 2019.

[11] J. Bell and G. Kaiser. Phosphor: Illuminating dynamic data flow in commodity jvms. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 83–101, New York, NY, USA, 2014. ACM.

[12] D. Brumley, T. Chiueh, R. Johnson, H. Lin, and D. Song. Rich: Automatically protecting against integer-based vulnerabilities. In *In Symp. on Network and Distributed Systems Security*, 2007.

[13] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[14] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[15] E. N. Ceesay, J. Zhou, M. Gertz, K. Levitt, and M. Bishop. Using type qualifiers to analyze untrusted integers and detecting security flaws in c programs. In *Proceedings of the Third International Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, DIMVA'06, pages 1–16, Berlin, Heidelberg, 2006. Springer-Verlag.

[16] X. Cheng, M. Zhou, X. Song, M. Gu, and J. Sun. Intpti: Automatic integer error repair with proper-type inference. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 996–1001, 2017.

[17] E. Chin and D. Wagner. Efficient character-level taint tracking for Java. In *Proceedings of the 2009 ACM Workshop on Secure Web Services*, 2009.

[18] J. Currier. SchemaSpy. http://schemaspy.sourceforge.net/, August 2010.

[19] A. Dangel and R. Pelisse. Pmd. https://pmd.github.io/, October 2015.

[20] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in c/c++. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 760–770, Piscataway, NJ, USA, 2012. IEEE Press.

[21] Elastic. Elastic Search. https://www.elastic.co/products/elasticsearch, October 2015.

[22] W. Enck, P. Gilbert, B. Chun, and L. Cox. TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones. In *OSDI*, 2010.

[23] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[24] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, Jan. 2012.

[25] M. Gordon, J. Eikenberry, A. Eden, J. Perkins, and M. Rinard. Precise and comprehensive provenance tracking for android devices. Technical report, MIT/CSAIL, October 2019. https://hdl.handle.net/1721.1/122968.

[26] W. G. J. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. pages 175–185, 2006.

[27] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 49–64, Berkeley, CA, USA, 2013. USENIX Association.

[28] J. Marki. Barcode4j. http://barcode4j.sourceforge.net/, February 2012.

[29] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 67–82, Berkeley, CA, USA, 2009. USENIX Association.

[30] R. Mui and P. Frankl. Preventing web application injections with complementary character coding. In *Proceedings of the 16th European Conference on Research in Computer Security*, ESORICS'11, pages 80–99, Berlin, Heidelberg, 2011. Springer-Verlag.

[31] P. Muntean, M. Monperrus, H. Sun, J. Grossklags, and C. Eckert. Intrepair: Informed repairing of integer overflows. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[32] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[33] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In R. Sasaki, S. Qing, E. Okamoto, and H. Yoshiura, editors, *SEC*, pages 295–308. Springer, 2005.

[34] B. Pugh, A. Loskutov, K. Lea, and D. Hovemeyer. Findbugs. http://findbugs.sourceforge.net/, March 2015.

[35] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 135–148, Dec 2006.

[36] D. Ray and J. Ligatti. Defining code-injection attacks. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 179–190, New York, NY, USA, 2012. ACM.

[37] D. Ray and J. Ligatti. Defining injection attacks. In S. S. M. Chow, J. Camenisch, L. C. K. Hui, and S. M. Yiu, editors, *Information Security*, pages 425–441, Cham, 2014. Springer International Publishing.

[38] D. Sarkar, M. Jagannathan, J. Thiagarajan, and R. Venkatapathy. Flow-insensitive static analysis for detecting integer anomalies in programs. In *Proceedings of the 25th Conference on IASTED International Multi-Conference: Software Engineering*, SE'07, pages 334–340, Anaheim, CA, USA, 2007. ACTA Press.

[39] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 74–83, New York, NY, USA, 2008. ACM.

[40] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[41] S. Sidiroglou-Douskos, E. Lahtinen, N. Rittenhouse, P. Piselli, F. Long, D. Kim, and M. Rinard. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 473–486, New York, NY, USA, 2015. ACM.

[42] S. Son, K. S. McKinley, and V. Shmatikov. Diglossia: detecting code injection attacks with precision and efficiency. In *Proceedings of the 2013 ACM SIGSAC conference on Computer &#38; communications security*, CCS '13, pages 1181–1192, New York, NY, USA, 2013. ACM.

[43] H. Sun, X. Zhang, C. Su, and Q. Zeng. Efficient dynamic tracking technique for detecting integer-overflow-to-buffer-overflow vulnerability. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 483–494, New York, NY, USA, 2015. ACM.

[44] M. Sun, T. Wei, and J. C. Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 331–342, New York, NY, USA, 2016. ACM.

[45] TASC Inc, Ponte Technologies LLC, and i_SW LLC. STONESOUP: Test Case Weakness Variants (Snippets) Documentation. https://samate.nist.gov/SARD/resources/STONESOUP_Weakness_Variants_Snippet_Documentation.pdf, 2014.

[46] TASC Inc, Ponte Technologies LLC, and i_SW LLC. Test and Evaluation Execution and Analysis System (TEXAS). https://samate.nist.gov/SARD/resources/STONESOUP_TEXAS_CLI_User_Guide.pdf, 2014.

[47] TASC Inc, Ponte Technologies LLC, and i_SW LLC. Test and Evaluation Execution and Analysis System (TEXAS): System Design Document. https://samate.nist.gov/SARD/resources/STONESOUP_System_Design_Document.pdf, 2014.

[48] TASC Inc, Ponte Technologies LLC, and i_SW LLC. TEXAS Communications API User Guide. https://samate.nist.gov/SARD/resources/STONESOUP_TEXAS_CLI_User_Guide.pdf, 2014.

[49] T. Wang, C. Song, and W. Lee. Diagnosis and emergency patch generation for integer overflow exploits. In S. Dietrich, editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 255–275, Cham, 2014. Springer International Publishing.

[50] T. Wang, T. Wei, Z. Lin, and W. Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *NDSS*. Citeseer, 2009.

[51] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Improving integer security for systems with KINT. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 163–177, Berkeley, CA, USA, 2012. USENIX Association.

[52] S. Wilson, P. Moore, and V. Nikic. htmlcleaner. http://htmlcleaner.sourceforge.net/, October 2015.

[53] 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 Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[54] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou. Intpatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In *Proceedings of the 15th European Conference on Research in Computer Security*, ESORICS'10, pages 71–86, Berlin, Heidelberg, 2010. Springer-Verlag.

[55] Q. Zhao, D. Bruening, and S. Amarasinghe. Efficient memory shadowing for 64-bit architectures. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 93–102, New York, NY, USA, 2010. ACM.

[56] B. Zimmerman, L. Fox, and T. Thrin. CoffeeMud. http://sourceforge.net/projects/coffeemud/, January 2015.