MECA: an Extensible, Expressive System and Language for Statically Checking Security Properties

Junfeng Yang, Ted Kremenek, Yichen Xie and Dawson Engler
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A.

ABSTRACT

This paper describes a system and annotation language, MECA, for checking security rules. MECA is expressive and designed for checking real systems. It provides a variety of practical constructs to effectively annotate large bodies of code. For example, it allows programmers to write programmatic annotators that automatically annotate large bodies of source code. As another example, it lets programmers use general predicates to determine if an annotation is applied; we have used this ability to easily handle kernel backdoors and other false-positive inducing constructs. Once code is annotated, MECA propagates annotations aggressively, allowing a single manual annotation to derive many additional annotations (e.g., over one hundred in our experiments) freeing programmers from the heavy manual effort required by most past systems.

MECA is effective. Our most thorough case study was a user-pointer checker that used 75 annotations to check thousands of declarations in millions of lines of code in the Linux system. It found over forty errors, many of which were serious, while only having eight false positives.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Languages Constructs and Features; D.2.4 [**Software Engineering**]: Software/ Program Verification—*Statistical methods*; D.4.6 [**Operating System**]: Security and Protection

General Terms

Reliability, Security, Measurement.

Keywords

annotation language, static analysis

1. INTRODUCTION

Static analysis can find many security holes. Bishop and Dilger [2] describe how to statically find "time-of-check-to-time-of-use" (TOCTTOU) race conditions in privileged Unix applications. More recently, there has been work on finding information leaks [17], intrusion detection [21], and checking uses of unsanitized user input [1]. And, of course, significant attention has been paid to finding buffer overflows [6, 15, 18, 20, 22].

While checking analyses have made progress, the features needed to apply these analyses to a given system have not enjoyed similar advances. First, programmers are given relatively limited means of expressing what program constructs to check. Some systems require programmers to specify properties using external text files [1]. Others use source annotations that are largely limited to inserting textual names in front of variables [9, 12]. Their focus on unary type qualifiers makes expressing even simple properties such as "check X before Y" either impossible or painful [7, 23]. Furthermore, past systems provide a fixed, one-size-fits-all set of rules for how annotations propagate. When these hardwired decisions are too aggressive, they cause a significant number of false positives [12]. When they are too weak, they require programmers to compensate by manually inserting an (often impractical) large amount of annotations. For example, systems such as Splint do not propagate annotations across function boundaries, programmers must annotate every function interface that needs to be checked [9]. Even advanced systems such as the race detector in Flanagan and Freund [10] have overheads of roughly one annotation per 50 lines of code at a cost of one programmer hour per thousand lines of code [10]. Finally, past systems do not give programmers a way to exploit the significant amount of *latent specifications* [8] (such as naming conventions) already present in most programs. Instead they must redundantly re-annotate them, rather than simply mapping these pre-existing specifications to those needed by the checker.

This paper describes an annotation system, MECA, that attempts to counter these problems. MECA is an extensible annotation language coupled to a flexible, powerful annotation propagation framework. We have combined MECA with the MC checking system [1, 8, 4], which allows programmers to write custom extensions that can check a rich set of security properties [1] such as "check capability X before doing operation Y" and "sanitize untrusted variables before using them." MECA allows implementors to (1) write a custom checker and (2) define a set of annotations that are relevant to that checker. The annotations are then used by the programmer to annotate checker-relevant parts of their source code. MECA then propagates these annotations automatically through the source code, possibly in a checker-

CCS'03, October 27–31, 2003, Washington, DC, USA.

specific way. Simple propagation examples would be across function calls and from the right-hand side of an assignment to the left. More aggressive examples include propagating annotations across all functions assigned to the same function pointer and using statistical inference to annotate function formal parameters based on the (possibly conflicting) annotation information of the function's callers (§ 5).

MECA was designed with the following goals in mind:

1. Expressiveness. Extension writers should be able to express all program information relevant for their security checkers. We currently focus on the requirements of the most common security properties: attaching annotations to program objects (e.g., functions, variables, field types) and allowing these to be bound together. These allow checking common rules such as: variables must be vetted before use and and certain operations (e.g., permission checks) must precede others (e.g., mutations).
2. Low annotation overhead. We want to minimize user labor, since this also minimizes the amount of programmer mistakes and maximizes the chance that they will use the tool. Programmers should get as much effect as possible from each annotation they add.
3. Intuitive syntax and semantics. Programmers should be able to define annotations that express their properties in a direct, intuitive way.
4. Low false positive rates. We favor effectiveness over soundness and want to find as many bugs as possible while minimizing false positives.

We have explicitly designed MECA to work well with already written source code bases (or, equivalently, new code bases that were initially built without checking in mind). In practice, even a few annotations can be sufficient for the checker to check hundreds of different locations.

MECA combines existing techniques with several novel ones. Its main technical contributions are:

1. A simple but powerful set of annotation primitives. These include data-dependent annotations (§ 3.2), programmatic annotations (§ 3.3), and general n-ary annotation predicates.
2. Annotation inference (§ 5), which uses statistical analysis to detect missing annotations.
3. Measurements of the effectiveness of our propagation methods. These show that MECA can derive hundreds of checks for each manual annotation, which to the best of our knowledge significantly exceeds the ratio of current approaches.

The next section gives a quick overview of the system. Section 3 describes the annotation language in more detail, Section 4 describes the propagation algorithms and Section 5 discusses annotation inference. Section 6 gives a toy example of how to use MECA to find missing permission checks. Section 7-8 presents our main case study, which uses MECA to detect illegal uses of tainted pointers in operating system code. Its experiments show that the system finds many errors with few false positives and derives many checks from a single annotation. Section 9 discusses related work and finally we conclude.

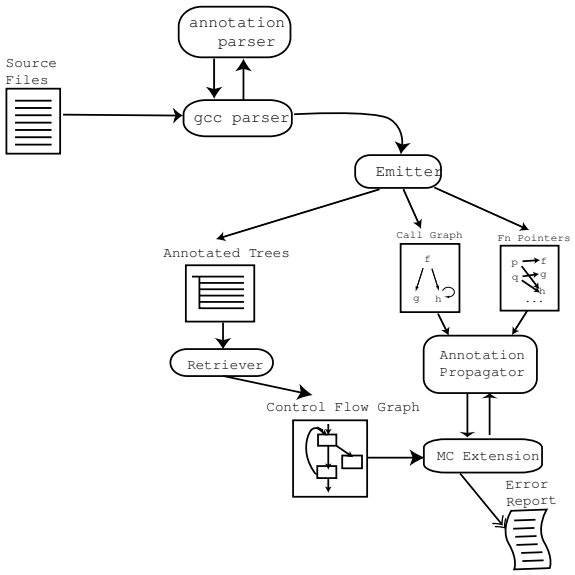

Figure 1: An Overview of the Analysis

2. OVERVIEW

Figure 1 gives an overview of the MECA system. It consists of an emitter, a retriever, an annotation propagator, and one or more checking extensions. The emitter uses a modified version of the GNU C Compiler (GCC 3.1) to parse the checked system's source code and its associated annotations into abstract syntax trees (ASTs). The ASTs are serialized and dumped onto the disk for further processing, along with the call graph and a file containing all the function pointer assignments. The retriever retrieves these emitted trees and constructs a control-flow graph for each function with sets of AST trees as basic blocks and branching statements as edges [4]. The CFGs are linked into a global call graph, which is then processed by the propagator. The propagator propagates annotations throughout this global call graph and then runs the given checking extensions over it. (Section 4 describes propagation in more detail.)

Checking extensions are written in a system based on the MC checking framework [1, 8]. Our variant is written in the ML programming language rather than C, but much of the features are the same: extensions match on program features they care about, potentially associating these features with states (such as "tainted"), and are applied in a flow-sensitive, inter-procedural manner down all control flow paths [4]. Because of the close similarities and the fact that the prior system has been amply documented this paper takes the checking system as a given and instead focuses on the annotation language and the propagation algorithm.

To illustrate how the pieces fit together, consider a checker that enforces the property "operating system code should not dereference user (i.e., tainted) pointers." (For concreteness, we will refer to this example repeatedly in the subsequent sections.) Instead operating system programmers must access the pointed-to data using special "paranoid" routines (e.g. `copyin` and `copyout` on BSD-derived systems). A single unsafe dereference can crash the system or, worse, give a malicious party control of it. A checker

```
annot_def ::= 'annot' annotation 'prop' proptype
            'annotates' binding;

proptype ::= 'FNPTR' | 'CALLCHAIN'
    | '(' proptype ')'
    | '(' proptype '|' proptype ')'
    ;
binding ::=
     '$variable' | '$parameter' | '$ret' | '$function'
    | '(' binding ')'
    | '(' binding ',' binding ')'
    ;
```

Figure 2: Annotation declaration grammar.

for this rule would define a set of annotations for specifying: (1) which variables, parameters and fields are tainted, (2) which functions produce tainted values and (3) which variables or fields indicate kernel "back-doors" (where an ostensibly "tainted" pointer is actually safe and can be explicitly dereferenced). Programmers would then apply these annotations to their source code, both manually and possibly also using a programmatic-annotation, such as one that automatically marks all pointers passed in through system calls as tainted. MECA would then propagate these annotations throughout the source code, statistically infer additional ones, and run the checker.

3. ANNOTATION LANGUAGE

This section gives an overview of our annotation language.

3.1 Syntax and grammar

Annotation keywords are defined by the extension writer. They must be declared before use; typically they reside in a header file that is included by the checked code. Undefined annotations are flagged as errors to catch misspellings and mismatches between annotations and extensions. Annotations appear in comments. Programmers place these where appropriate. Our system applies an extension to each program point (e.g., statement, expression) down every path in the source code and the extension searches for annotations and constructs it cares about.

A simple declaration of a "tainted" annotation would be as follows:

```
annot tainted annotates ($variable)
```

This declaration specifies that the `tainted` annotation can be used to bind to any variable ("`$variable`"), where variable includes the actual value returned by a function, as well as the more obvious parameters and global and local variables. Figure 3 shows some sample uses.

Figure 2 gives the general annotation declaration. The specifier "`binding`" specifies how the annotation should bind in a use, and is used to resolve ambiguities. In the example above, we could have simply used the C language default bindings of type-qualifier to declarations, but this causes difficulties when annotations are more than just simple type qualifiers. Programmers can specify that annotations bind to variables in general (as above), to functions (`$function`), to parameters (`$parameter`) or to return values (`$ret`). To specify that an annotation binds to more than one type, the programmer gives a comma-delimited list. However, some combinations are either redundant or conflict and will

```
/*@ tainted */ int *p; // p is a tainted variable.
struct foo {
  /*@ tainted */ int *p; // the field p is tainted
};

// foo takes a tainted parameter p and returns
// a tainted pointer.
/*@ tainted */ int *foo(/*@ tainted */int *p);

// neither dst nor src can be tainted.
void memcpy( /*@ !tainted */void *dst,
        /*@ !tainted */ void *src, unsigned nbytes);

// the data copied into dst is tainted, and the
// pointer src is tainted. Label POST tells the
// system that the content of dst is tainted only
// the call to copyin
void copyin(/*@ POST:tainted (*dst) */ void *dst,
       /*@ tainted */ void *src, unsigned len);
```

Figure 3: Sample Annotations for the tainted checker in Section 7.

```
// implied: p is tainted
void *f_root(/*@ tainted */ void *p);

// p becomes tainted because of f_root
void g_root(void *p) {
  *p;   // error: "deref of tainted pointer"
}

struct ops { void (*fptr)(void *p); };
struct ops f = { f_root };
struct ops g = { g_root };
```

Figure 4: A contrived example to illustrate `FNPTR` annotation propagation. The implied annotation of `f_root` propagates across the function pointer that `f_root` is assigned to, tainting the parameter `p` passed to `g_root`.

be flagged. Since `$variable` includes `$ret`, "`($variable, $ret)`" would be a redundant combination. Since annotations can only bind to the return type or the function definition itself, "`($function, $ret)`" would be an error. In general, `$variable` and `$function` make up the majority of all declarations.

The `prop` part of the declaration allows the extension writer to specify two built-in propagation methods: `LOCAL` and `FNPTR`. `LOCAL` indicates that the analysis is solely intra-procedural; by default our analysis is inter-procedural and formal parameter annotations are propagated to callers. `FNPTR` propagates annotations across function pointer assignments: if one function has an annotation, and it is assigned to a function pointer `fp`, then the annotation is propagated to all other functions assigned to the same function pointer. Figure 4 gives a contrived example of how such propagation works.

3.1.1 Binding annotations together

Often a checker needs to express relationships between multiple program objects or between different annotations. The stylized way to do this is for the checker to define an annotation that takes multiple arguments, where the objects to be bound together are placed in the argument slots. For

```
gen_annotation := label annotation args;
args ::= /* empty */
     | '(' arg_list ')'
     ;
arg_list ::= arg
     |  arg ',' arg
     ;
arg ::= gen_annotation | C−expr
     ;
label := /* empty */
     | 'PRE:' | 'POST:'
     ;
```

Figure 5: Grammar for the fully-general annotation.

```
struct foo {
  /* Non-zero value implies struct data comes from the user. */
  int user;

  /* If user is unknown mark the field
       as tainted, otherwise mark as untainted. */
  /*@ user != 0 || unknown ==> tainted*/ void *conservative;

  /* If user is non-zero mark field as tainted,
       otherwise do not annot. */
  /*@ user != 0 ==> tainted */ void *non_conservative;
};
```

Figure 6: Example of structure field annotation coupled with the use of the `unknown` keyword.

example, a buffer overflow checker could define a "`set_length`" annotation that specifies that an integer binds to a pointed-to object. The programmer could then define a "`check_access`" annotation to specify that the length field to `memcpy` binds both the `dst` and `src` parameters:

```
/*@ check_access(src, len) check_access(dst, len) */
void memcpy(void *dst, void *src, unsigned len);
```

Another possibility is that the returned value of `malloc` is bound by its `size` parameter:

```
/*@ set_length($ret, sz) */
void *malloc(unsigned sz);
```

The `check_access` annotation will be matched by the buffer-overflow checker, which extracts the two arguments and uses the given bound to check them.

Figure 5 gives the more general grammar for these annotations. The annotation is a comma-separated list containing zero or more elements. Each element can be a checker-defined annotation or a C expression (`C-expr`). There are two restrictions on the C-expression. First it cannot be a compound expression such as "(x, y)." Second, the expression must be able to be parsed by the C compiler at this point. Besides these restrictions, the annotations can refer to arbitrary program values: function addresses, variables, general arithmetic expressions, and macros. Additionally, the expression can refer to field names and parameters before they are defined, though they cannot refer to undefined variables. Annotations for functions or parameters can be labeled with "`PRE`" or "`POST`," which means the annotations are bound to the targets before or after the function call. By default annotations are bound to the targets before calls.

Allowing the binding of n-ary expressions provides a nice increase in expressiveness over constructs such as unary type qualifiers.

3.2 Support for data-dependencies

Our checking system is path-sensitive, and will suppress many common infeasible paths [4].[1] However, in general pruning all false paths and resolving all data dependencies are undecidable problems. Thus, we allow programmers to provide help when calculation fails using a built-in predicate to express data dependencies:

```
expr ==> annot
```

Here `expr` can be any valid C expression and `annot` any checker-defined annotation. If `expr` is true, the annotation is bound to the associated object. If the expression is false, then the negation of the expression is bound (e.g., "not tainted"). We call such annotations `imply` annotations, and the predicate expression `expr` is an `imply-condition`.

A possible use with the tainted checker would be to specify parameter-dependencies that control whether an argument is actually tainted or not. For example, the following declaration states that `foo`'s argument `p` is tainted only when the bitwise-or of `flag` and the constant `USER_FLAG` is non-zero:

```
void foo(/*@ flag&USER_FLAG ==> tainted*/ char* p,
                                          int flag);
```

We use similar declarations to specify when a structure field indicates that a pointer in the structure is tainted.

Because we evaluate implication expressions at compile time, their value may be unknown. Programmers can control the implication in this case by using the special keyword `unknown`, which evaluates to true if the expression cannot be resolved at compile time. They can thus make the annotation conservative (e.g., `unknown` implies `tainted`) or non-conservative (e.g., `unknown` implies not `tainted`). This is illustrated in Figure 6.

Our system implements `imply` annotations by keeping track of a set of known predicates along each path. When it encounters a program object with one or more `imply` annotations, it evaluates these against the set of known predicates.

3.3 Programmatic annotations

Traditionally, annotating source code is brute force: programmers insert them in every place the checker will need them at. This reliance on manual labor is both tedious and error-prone, since a single missed annotation can mean that a check does not occur, or that the process is so strenuous that the programmer quits after only annotating a few hundred thousand lines of code.

MECA lets programmers automate this process using *programmatic annotations*, which conceptually are applied to all points in the program and, when their conditions are satisfied, mark the program point with their annotation.

For example, to specify that all system call parameters should be tainted we would write:

```
/*@ global
  $param: ${!strncmp (current_fn, "sys_", 4)}
  ==> tainted */
```

This programmatic annotation specifies that it should be applied globally over the entire checked system (`global`) and

[1]This implicit pruning obviates much of the need for explicit data-flow flags introduced in the Vault language [5].

```
prog_annot ::=
      scope objs ':' '${' C−expr '}'
        '==>' annotation ;

scope ::= 'global' | 'file_global';
objs ::= obj;
obj ::= '$variable' | '$parameter' | '$ret' | '$function'
        | obj
    | obj OR obj
    ;
```

Figure 7: General syntax for programmatic annotation declarations.

that it cares only about parameters (`$param`). It will be applied over each parameter in each function. The part in curly braces checks if the current function name is prefixed by "`sys_`" which is the Linux kernel naming convention for system calls. If so, it returns true, and each parameter will be marked as tainted.

The general form of the rule is depicted in Figure 7. "Scope" controls whether the annotation is applied over the entire system (`global`) or just within one file (`file_global`). (Note that these annotations can be overridden by local annotations.) The "object" specification controls what it is applied to: functions, return values, parameters or variables. The `C-expr` can be a normal `C-expr` described in Section 3.1.1 or a callout to helper functions the system provides. It can refer to program objects using special variable names such as `current_fn` (the current function), `current_file` (the current file), `current_param` (the current parameter), and `current_var` (the current variable). Programmers can also checks the typename of a program object by `type_is(typename)`.

In practice, a major use of programmatic annotations is to translate system-specific naming conventions (which can be viewed as ad hoc pre-existing annotations) into checkable annotations. The example above falls into this category. Another example would be exploiting a naming convention where a pointer parameter has an associated length that contains its name as a prefix (e.g., the length for parameter `foo` is specified by `foo_len`).

Programmatic annotations are useful even when programmers are supposed to manually annotate all relevant program objects. In this case they can be used as "annotation assertions" that prevent false negatives by detecting missing annotations.

4. ANNOTATION PROPAGATION

This section describes MECA's flow-sensitive, bottom-up, inter-procedural analysis for propagating annotations from callees to their callers.

The analysis is initialized by retrieving the base annotations from annotated code. These annotations consist of annotations for functions, their parameters and return values. The annotations are used to build *summaries*, which are a set of ⟨`guard`, `pre-condition`, `post-condition`⟩ triples. Here `guard` is a truth assignment to the set of `imply` conditions (if any), and `pre-condition` and `post-condition` describe the annotation bindings before and after the call-site. For example, the code

```
/*@ tainted */ int * foo(/*@ tainted */int *p,
                          /*@ POST: tainted(*q) */ void *q);
```

will generate the following summary for `foo`:

$$\langle <>, \{tainted(\$1)\}, \{tainted(\$ret), tainted(\$1), tainted(*\$2)\}\rangle$$

The guard here is empty. The pre-condition states that first argument is tainted before the call. The post-condition states that after the call (1) the return value is tainted, (2) the first argument remains tainted and (3) the storage pointed to by the second argument is tainted. There will be one tuple for each different guard expression. Usually there is only one (empty) guard, and thus one tuple for each function. The set of summaries for each function is stored in a *summary map*, indexed by function name.

After constructing these base summaries, the analysis then places all annotated functions and their (transitive) callers into a worklist in topological order (based on the function call graph). Recursive call chains are broken arbitrarily. Each function is dequeued from the worklist and analyzed until its summary converges to a fixed point or a maximum simulation time is hit. Since callees of a function will be analyzed before it, each function only needs to be added to the worklist once.

The analysis analyzes each path in a function (i.e., is flow-sensitive) and uses caching for speed [4]. It tracks the values of variables using a symbolic *store* to record assignments. Currently we only keep track of all the parameter values and their one-level dereferences (e.g. `*parm`, `parm->field`). [2]

Similarly, the analysis uses a predicate store `pred_store` to evaluate conditions and prune false paths. The predicate store records simple conditional expressions encountered on the current path (currently expressions composed from negation, equality, inequality, and simple bit-wise masks). It evaluates each conditional expression it encounters against these recorded value and, if it is false, skips the true (or false) branch.

Extensions can control annotations propagation across expressions in an extension provided method called extension_visit that is called by the analysis on every visited expression. For example, the tainted checker would specify that performing arithmetic on a tainted expression results in another tainted expression.

During intra-procedural analysis (local within a function) we record the annotations associated with each expression on the current analyzed path in an annotation store. This annotation store serves two purposes. First, it allows the analysis to track annotations as the values they correspond to flow through assignments and expressions. Second, it allows it to update pre- and postconditions in the function summaries. When an expression is added to the annotation store, the analysis checks the symbolic store to see if this expression is a parameter or a one-level dereference of a parameter. If it is, then it updates the function's pre-condition. When the analysis reaches the end of a path in a function, the annotation store is used to update the function's post-conditions. Later, when a call to this function is encountered, these post-conditions will be applied.

Figure 8 depicts a contrived example to illustrate how bottom-up propagation works. Here `bar`'s pointer argument

[2]We experimented with deeper value flow analysis (i.e., more than one level of indirection) but the results thus far were not worth it: it dramatically slowed down the analysis, rarely gave useful information, and as the level of indirections increased it became more likely that an approximation error occurred, giving false positives.

```
void bar(/*@ tainted */void *p);
struct S {char* buf;};

void foo (char **p, struct S* s, char* q) {
   char *r, *u, *v;
   struct S* ss;

   r = *p; // r has sym_value *p
   bar(r); // taints r and *p
   ss = s; // ss has sym_value s
   bar(ss->buf); // taints ss and s->buf
   q = v; // q becomes unknown
   // will not taint formal parameter q
   bar(q);
}

// After the bottom-up propogation algorithm
// finishes, function foo will be summarized as:
foo (/*@ tainted (*p) */ char **p,
     /*@ tainted(s->buf) */ struct S* s,
      char* q);
```

Figure 8: Bottom-up propagation example. Formal parameter `q` is not tainted because it is redefined before the final call to `bar`.

is annotated as `tainted` and the annotation propagates using the bottom-up propagation analysis. At the end of the analysis `foo`'s formal parameters are annotated.

We have found that in practice flow-sensitivity is the single most important feature to ensure accurate annotation information. Without it we falsely propagate annotations beyond where they should go, giving many false positives.

5. STATISTICAL ANNOTATION INFERENCE

This section describes how we statistically infer formal parameter annotations. This technique is useful for preventing false negatives caused when a portion of the callgraph (1) contains no annotations or (2) calls a leaf function whose source code is unavailable. It automatically infers the most plausible annotation for unannotated functions, uses a utility metric to order procedures from most to least worthwhile to annotate, and presents this ranking to the user for inspection. They typically inspect the top 10-20 and then annotate them directly. This approach allows users to quickly annotate the parameters whose type values we are most confident about. Once these functions are annotated more code can be annotated (and checked) by re-applying the bottom-up analysis with these new annotations.

More precisely, our goal is to infer an annotation for the for the *ith* formal parameter of function f (denoted $f{:}i$) that agrees with its callers and then order all inferred annotations from most to least likely. We do this in two steps: (1) pick the annotation A and (2) compute how likely A is the correct annotation. The first step is trivial: set the annotation for $f{:}i$) to be the annotation A passed most often as the *ith* argument to f. If all the annotation for the *ith* argument at all callsites to f are known and are the same, the annotation type for $f{:}i$ is considered unambiguous and we set $f{:}i$ to this annotation and stop. In practice about half of the functions we analyze are consistent in this way. For the other half we need to do the second step, and compute the probability that the annotation A is correct (i.e., that $Pr(f{:}i = A)$). We need to do this step exactly when (1) there exists at least two actual parameters corresponding to $f{:}i$ that are annotated but with *conflicting* types and/or (2) some of the actual parameters are `unknown`. We briefly defer the problem of `unknown`s until Section 5.1, and for now assume that all actual parameters are annotated (either directly or through annotation propagation).

A naïve way to compute $Pr(f{:}i = A)$ would be as a percentage. Unfortunately this ignores population size. For example, suppose we have two functions `foo` and `baz` whose first parameter was passed a `tainted` pointer 3 out of 4 times and 18 out of 24 times respectively. While both have a ratio of 0.75, we have much more confidence this is the true ratio for `bar`. In contrast, the ratio for `foo` could be coincidental and could easily change dramatically with more observations. Thus, instead of percentages we use *z-ranking* [8, 14], a ranking scheme based on statistical hypothesis testing [13]. It incorporates the intuitions outlined above to institute ranking, and it utilizes the population size in a statistically sound way.

For type inference it works as follows. We have two binary types A and $\neg A$. We wish to compute a value that tells us how likely a formal parameter $f{:}i$ has type A; this will be done however by examining the behavior we get if we *instead* assign the type $\neg A$ to $f{:}i$. Let n be the number of callsites to function f, k the number of actual parameters corresponding to $f{:}i$ that have type A, and $n-k$ the number that have type $\neg A$. Note if $f{:}i$ is annotated as type $\neg A$ we will have $X = k$ type errors. We quantify the reliability of the type assignment to $f{:}i$ by computing $P(X \geq k)$, or the likelihood we would observe k *or more* type errors. This done by assuming that errors have a fixed, *a priori* error rate p_0. We then model type errors as independent binary trials, or tosses of a biased coin that has a probability of p_0 of turning up as "type error." By modeling type errors as binary trials, $P(X \geq k)$ is computed using the cumulative Binomial distribution [19]:

$$P(X \geq k) = \sum_{j=k}^{n} \binom{n}{j} {p_0}^j (1-p_0)^{n-j} \qquad (1)$$

The value computed by Equation 1 is called the p-value. If k represents the number of type errors we get if we assign type A to $f{:}i$, we denote the corresponding p-value as $p(A)$. The value $p(\neg A)$ is defined analogously. A low $p(\neg A)$ implies that $f{:}i$ is unlikely to have type $\neg A$. Because we are using binary types, however, this means that we have strong confidence in the alternative explanation, namely $f{:}i$ has type A. Thus a *low* $p(\neg A)$ implies strong confidence in the type assignment A to $f{:}i$. Because of this implication, we let $s(A) = p(\neg A)$ to represent the confidence "score" for an assignment of type A to a formal parameter $f{:}i$.

The value computed by Equation 1 is called the p-value. Because we sum from k to n, smaller values are better, since they correspond to more successes than we expected. For type inference we typically set $p_0 = 0.1$ since generally the expected error rate is low. Our experiments were not that sensitive to the exact value chosen for p_0.

5.1 Unknowns as meta-annotations

Of course, we often cannot determine the annotation of all arguments at a given callsite. The presence of such `unknown`s indicates a (possibly checker-specific) analysis failure. We have only limited experience with `unknown`s in the context of

annotation inference. However, initial results indicate that they are either (1) innocuous and can be safely ignored or (2) that they instead indicate a construct or code that the checker cannot handle. In this latter case, the higher the proportion of `unknown`s the less confidence we should have in our inferred annotations. Section 7.3 gives an example of how to incorporate this information into our annotation inference.

5.2 Next-best annotation

So far we have discussed ranking formal parameters by type confidence. Although this is useful, it does not reflect the *impact* of an annotation. The impact or "utility" of an annotation is the increased annotation coverage when the bottom-up analysis is re-applied. This may be cumbersome to compute; an approximate measure of the impact of annotating a formal parameter is the number of actual parameters it will annotate (i.e., the number of `unknown`s). Our estimate of the impact of an annotation is also based on our confidence of a type assignment; formal parameters whose type we are very confident about but have many `unknown` actual parameters will be the annotations we expect to have the highest impact.

Consequently, if we wish to rank formal parameters by the utility of annotating them with a type A, we use the following metric:

$$utility = [1 - s(A)] \times u \quad (2)$$

Here u is the number of actual parameters marked `unknown`. We use the complement $1 - s(A)$ so that larger values of *utility* are better. In practice utility ranking is effective; in the tainted checker we use it rank formal parameters that are most likely to be `tainted` and cause the greatest impact by being annotated.

6. A TOY CAPABILITY CHECKER

Operating systems such as Linux use capabilities to enforce access control to certain sensitive data in the kernel. Missed capability checks allow user processes to bypass security policies and potentially gain unauthorized access to sensitive data. In this section, we use a toy example to illustrate how MECA can be used by a checker that flags missed capability checks.

The checker defines two predicates, `guard` and `noguard`. The `guard` predicate specifies that an annotated type or variable is protected by a given capability. Conversely, `noguard` exempts certain structure fields from being protected by an enclosing annotation.

These two predicates are defined on lines 1-2 in Figure 9. The `$variable` flag in the definitions denotes that they describe properties of program variables.

The annotation `guard(cap, SYS_ADMIN)` on line 5 indicates that the `data` field defined on that line is protected by the field `cap`, which must contain the `SYS_ADMIN` capability. Line 15 gives a more exuberant use of the annotation, which uses it to protect the entire structure `S2` rather than just a single field. To make things more interesting, suppose the `useless` field on line 10 does not need any protection. In this case, we use the `noguard` predicate to exempt `useless` from the enclosing protection.

Linux uses the function `capable` to do capabilities checks. We wrote a simple checker that tracks all successful capability checks on each path and records these in a "capability set." It emits an error if a protected object is accessed without its required capability being held. For example, on the true branch of the `capability` check on line 18, the extension records that the structure field `s2->cap` holds the capability `SYS_ADMIN`. The checker uses this information to determine that the access to `s2->data` on line 19 is safe. However, it reports an error on the false branch at line 21, since `s2` does not have the right capability. It will similarly report one error on line 23. No error will be reported on line 24 since `useless` is not protected.

```
1 : /*@ annot guard annotates ($variable);
2 :     annot noguard annotates ($variable); */
3 :
4 : struct S{
5 :   /*@ guard(cap, SYS_RAWIO) */ int data;
6 :   int cap;
7 : };
8 :
9 : struct S2{
10:   /*@ noguard */ int useless;
11:   /* local annotation noguard overwrites guard */
12:   int data;
13:   int data2;
14:   int cap;
15: }/*@ guard(cap, SYS_ADMIN) */;
16:
17: void foo (struct S2* s2) {
18:   if (capable(s2->cap, SYS_ADMIN))
19:     s2->data = 0; /* OKAY */
20:   else
21:     s2->data2 = 1; /* ERROR : no permission */
22:   if (capable(s2->cap, SYS_RAWIO))
23:     s2->data2 = 0; /* ERROR : wrong permission */
24:   s2->useless = 1; /* OKAY */
25: }
```

Figure 9: An example for the capability checker.

7. CHECKING USER-POINTER ERRORS

This section is an in-depth case study of of how to use MECA annotations to find uses of tainted pointers. We apply these ideas to Linux and measure their efficacy and annotation overhead.

7.1 The annotations

At a high level the checker mirrors the description in Section 3. It defines a single `tainted` annotation. The programmer then manually inserts these annotations, writes global annotators, and suppresses false positives from kernel backdoors. The system then uses the flow-sensitive worklist algorithm described in Section 3 to propagate the annotations along call chains and across function pointers.

Figure 10 depicts representative examples of annotations inserted by the programmer for Linux code. These come in two categories: (1) programmatic annotations that mark chunks of code tainted or untainted and (2) more specific annotations that suppress false positives by selectively marking code as untainted or expressing data dependencies. We discuss each below.

As described in Section 3 the programmer uses a global annotation to mark all functions prefixed with the substring "`sys_`" as tainted. They then do more precise annotations such as:

```
1 : /* linux-2.5.63/include/asm-i386/uacess.h */
2 : static inline unsigned long
3 : copy_from_user(/*@ POST:tainted (*to) */ void *to,
4 :                /*@ tainted */ const void *from,
5 :                unsigned long n)
6 :
7 : /* linux-2.5.63/include/asm-i386/string.h */
8 :
9 : /*@ file_global
10:    $param : ${!type_is_int()} ==> !tainted */
11:
12: /* linux-2.5.63/drivers/base/sys.c */
13: /*@ file_global $param: !tainted */
14:
15: int sys_register_root(struct sys_root * root){
16: }
17:
18: /* linux-2.5.63/ipc/shm.c */
19: asmlinkage long
20: sys_shmat (int shmid, char *shmaddr,
21:            int shmflg, /* !tainted */ ulong *raddr)
22:
23: /* linux-2.5.63/drivers/isdn/i4l/isdn_tty.c */
24: isdn_tty_write(struct tty_struct *tty, int from_user,
25: /*@ from_user ==> tainted */ const u_char * buf,
26:                int count)
27:
28: /* linux-2.5.63/drivers/char/random.c */
29: static ssize_t extract_entropy(struct entropy_store *r,
30: /*@ flags & EXTRACT_ENTROPY_USER ==> tainted */
31:                                void * buf,
32:                                size_t nbytes, int flags);
33:
34: /* linux-2.5.63/include/linux/module.h */
35: struct kernel_symbol
36: {
37:         unsigned long value;
38:         const char *name;
39: }/*@ mc_ignore */;
```

Figure 10: Representative programmer-inserted annotations taken from from Linux source files.

1. Lines 1-5: annotate the `copy_from_user` routine, which is one of many "paranoid" functions used by Linux to move data between user and kernel space. It has similar annotations as `copyin` described in Section 3.1.
2. Lines 9-10: a file-scope global annotation that marks all non-integer function parameters in "string.h" with "`!tainted`" (i.e., not tainted), which implies it is an error to call any of these functions with a tainted pointer. These functions are string functions (such as `strlen`, `strcpy`) which dereference their arguments but are coded in assembly, preventing our checker from analyzing them. The ability to do a single global annotation gives a safe, easy way to express this constraint.

The following two examples are representative of overriding tainting annotations:

1. Lines 13: uses a file-scope annotation to untaint all the parameters in "drivers/base/sys.c." This file violates the "`sys_`" naming convention: none of these functions are true system calls, despite the prefix. This file annotator overrides the previously described global annotator that marks all `sys_` functions as tainted.
2. Lines 19-21: gives a more precise override example, where we mark the final parameter to `sys_shmat` as not tainted.

The following two annotations suppress kernel backdoors:

1. Lines 24-26: shows an annotation for a "backdoor" function. Here, if the `from_user` parameter is non-zero, then the pointer `buf` is a user pointer. It is a safe, untainted kernel pointer otherwise.
2. Lines 29-31: gives another more complex example. If the bitwise-and of the `flag` parameter and the constant `EXTRACT_ENTROPY_USER` is non-zero then the parameter `buf` should be tainted and not otherwise.

As described in Section 4 the system will analyzes functions with these data-dependent annotations in two passes. In the first example above, the system will assume `from_user` is zero (and `buf` is not tainted) when analyzing the body of `isdn_tty_write`. A crucial feature is that during these passes it automatically prunes all control paths that assume `from_user` is not zero. In the second, it will assume `from_user` is not zero (and `buf` is tainted). Again, the system prunes paths that assume `from_user` is zero. When a call to a backdoor function is encountered, the `from_user` condition will be evaluated in the current calling context. If it is true, the corresponding actual argument will be set to be tainted.

Linux has a small number of backdoor functions. The majority of them are functions implementing one of three interfaces: (1) `usb_serial_device_type.write`, (2) `tty_driver.write` and (3) `isdn_if.readstat`. For such grouped functions we only need to annotate one of them to make the others annotated through function pointer propagation. In Linux, adding 36 annotations is enough to annotate almost all backdoor functions.

By precisely suppressing the tainting caused by kernel backdoors, we not only eliminate many false positives but also find more bugs, since they are not hidden in hundreds of false error messages. In the case of Linux this suppression allowed us to find five additional errors, four of which would allow a malicious user to print out arbitrary kernel data.

Finally, lines 35-39 eliminate dangerous imprecision in function pointer propagation. In Linux all exported functions via `EXPORT_SYMBOL` will be assigned to the structure field `kernel_symbols.value`. This assignment contains no checking information whatsoever. Doing function pointer-based propagation would be disastrous: if one function had a tainted argument, then they all would, which would lead to thousands of false positives. The programmer explicitly annotates `struct kernel_symbols` with the built-in annotation "`mc_ignore`" to suppress its use when propagating annotations.

Annotations for this checker essentially follow the procedure outlined in Section 4. We describe the checker specific aspects of both the bottom-up and top-down propagation of annotations respectively in the next two subsections.

7.2 Bottom-up analysis

We use the bottom-up inter-procedural propagation described in Section 4 to propagate annotation up call-chains. Within an individual function we use an extension to perform an intra-procedural analysis, annotating individual pointers as `tainted` or `!tainted`. The extension deploys a set of customized rules to propagate annotations among expressions.

Figure 11 shows the propagation rules used in the tainted checker. The `Assign` rule means that if `p` has annotation

$$\frac{p : \tau,\ \tau \in \{\texttt{tainted},\ \texttt{tainted(*p)},\ \texttt{tainted(p -> f)}\}}{q = p \ \vdash\ q : \tau}\ \textsf{Assign}$$

$$\frac{p : \tau,\ \tau \in \{\texttt{tainted}, \texttt{tainted(*p)}, \texttt{tainted(p -> f)}\}}{q = (type)p \ \vdash\ q : \tau}\ \textsf{Cast}$$

$$\frac{p : \texttt{tainted}}{q = \&p \ \vdash\ q : \texttt{tainted(*q)}}\ \textsf{Addr}$$

$$\frac{p : \texttt{tainted}}{q = p \bigoplus x \ \vdash\ q : \texttt{tainted}}\ \textsf{Arith}$$

$$\frac{(p \bigoplus x) : \texttt{tainted}}{q = p \ \vdash\ q : \texttt{tainted}}\ \textsf{ArithReverse}$$

$$\frac{s : \tau \rightarrow s : \texttt{tainted}}{s.f : \tau' \rightarrow s.f : \texttt{tainted}}\ \textsf{StructTrans}$$

Figure 11: Tainting Checker propagation rules.

`tainted`, `tainted(*p)` or `tainted(p->f)`, when the checker sees `q = p` it will give `q` the same annotation. The `Cast` rule shows that the annotations are preserved across casts. This rule is needed because OS code often casts pointers to integers and back. The `Arith` rule specifies that doing pointer arithmetic on a `tainted` pointer will not change its taintedness. The `ArithRerverse` rule is the reverse of the `Arith` rule. The `Addr` states that assigning the address of a `tainted` pointer `p` to `q` will annotate `q` with `tainted(*q)`. The `StructTrans` rule is different than the other rules because it specifies a propagation rule when annotation changes. Whenever a structure `s` becomes tainted, all its fields become tainted, too.

7.3 Statistical annotation inference

We apply the techniques discussed in Section 5 to derive annotations for formal parameters from actual parameters. For a given formal parameter, we have k number of actual parameters marked as `tainted`, and $n-k$ marked as `!tainted`. Furthermore, we have u additional parameters marked as `unknown`. We wish to rank formal parameters by (1) how likely they are to be `tainted` and (2) the impact they will have if we annotate them (§ 5.2).

The presence of `unknown`s posses an interesting problem. For the tainted checker, a large number of `unknown` actual parameters indicates that the value passed to a formal parameter is often not used as a pointer at all. Otherwise, a single dereference of the actual parameter would have led to an annotation of `!tainted`. Consequently, the presence of `unknown`s, regardless of whether the function was passed `tainted` pointers or not, may indicate that it is a polymorphic function that can take any pointer type. For our ranking we want the parameters most likely to be annotated `tainted`, so we wish to suppress such functions.

To accomplish this, we introduce the notion of *composite types*. For this problem, we have two types; the first is the type indicating whether or not the formal parameter is tainted, the second is whether or not the value passed to the formal parameter is really used as a pointer. We wish then to rank parameters by their composite type assignment – we want parameters that are likely to be tainted and whose values tend to be used as pointers.

To be general, suppose we have two separate annotations we can assign to a formal parameter. The first annotation, denoted $\mathcal{A}_1$, can take on types A or $\neg A$ and the second annotation $\mathcal{A}_2$ can take on types B or $\neg B$. The set of possible types for the aggregate annotation $\mathcal{A}_{1,2} = \langle\ \mathcal{A}_1, \mathcal{A}_2\ \rangle$ is then the cross-product of the two sets of types: $\{\langle A, B\rangle, \langle A, \neg B\rangle, \langle \neg A, B\rangle, \langle \neg A, \neg B\rangle\}$.

Suppose we wish to rank formal parameters by how likely they have annotation $\mathcal{A}_{1,2} = \langle A, B\rangle$. We know how to compute the score of each type $s(A)$ and $s(B)$ in isolation, and wish to compute $s(A \wedge B)$ for the aggregate annotation. A score $s(\cdot)$, however, is just the p-value probability $p(\neg\cdot)$, and from DeMorgan's laws and the basic axioms of probability we get:

$$s(A \wedge B) = p(\overline{A \wedge B}) = s(A) + s(B) - s(A \vee B) \qquad (3)$$

Furthermore, if we assume the type assignments to the first and second annotation are uncorrelated (and hence independent), we have that:

$$s(A \vee B) = p(\overline{A \vee B}) = s(A) \cdot s(B) \qquad (4)$$

These rules easily generalize for aggregate types involving more than two annotations. We can then apply these rules to rank formal parameters based on the score of their aggregate types.

These axioms allow us to rank formal parameters for the tainted checker in the desired manner. The composite value want to rank by is $\langle$ `tainted`, `isPointer` $\rangle$. Using Equations 3 and 4 we have the following score:

$$s(\texttt{tainted} \wedge \texttt{isPointer}) = s(\texttt{tainted}) + s(\texttt{isPointer}) - s(\texttt{tainted})s(\texttt{isPointer}) \qquad (5)$$

Finally, we wish to rank parameters using our utility metric (§ 5.2); this will cause formal parameters to be ranked both by how likely they are to be `tainted` and the expected impact they will have on increased annotation coverage and error checking. Using Equations 2 and 5, our final score we use to rank parameters by is:

$$\mathcal{S} = [1 - s(\texttt{tainted} \wedge \texttt{isPointer})] \times u \qquad (6)$$

8. RESULTS

This section measures the effectiveness of MECA. Several terms are used throughout this section, so we define them here.

1. **Manual Annotation**: The annotation string that must be added by programmers by hand.
2. **Global Annotator**: Each programmatic annotation is called a Global Annotator.
3. **Root Annotations**: Annotations that either come from programmers (global annotators excluded) or annotated by global annotators.
4. **Derived Annotations**: the number of formal parameters p or their one-level dereference $*p$ that were automatically labeled by bottom-up analysis. Each such p or $*p$ that went from being labeled `unknown` to `tainted` or `!tainted` is counted once.
5. **Check**: we define a check as a program point where the programmers can potentially make a mistake. For the

```
1 : void bar (char* p) {
2 :   copyin (_, p, _); /* Not a check. Infers p tainted */
3 : }
4 :
5 : void foo(char* r) {
6 :   copyin (_, r, _); /* Not a check. infers r tainted */
7 :   /* A check. Annotations for both r and bar's first
8 :      parm have already been infered*/
9 :   bar(r);
10: }
11:
12: void foo2(char* r) {
13:   bar(r);
14: }
```

Figure 12: This figure shows how we count checks and construct the propagation graph.

user-pointer checker, we consider checks to occur at every (1) pointer dereference and (2) when a pointer is passed to an annotated function (the annotation can be inferred). Note that we underestimate the number of checks because we only count program points where no inference happens as checks. For example, the first dereference of an unknown pointer implies the pointer is not tainted but performs no check (since the pointer is unknown); we would only count subsequent dereferences of the same pointer (if any) as checks. Figure 12 gives an example of how we count.

6. **Propagation Graph**: Each node in the propagation graph represents either (1) a derived annotation or (2) a callsite argument where the argument a has some annotation (`tainted` or `!tainted`) and the function parameter it is passed to has a derived annotation. Note, this latter count is done in this way to avoid counting "vacuous" annotations — i.e., when an annotated variable is passed to a function that has no annotation and hence leads to no check. Each edge in the graph represents a possible flow of annotations from one parameter to another.

   We elide local variables and expressions from the graph, since these counts vastly inflate the graph size and make it harder to evaluate effectiveness.

   For example, the propagation graph for Figure 12 contains two subgraphs: one subgraph rooted from `foo:r` consists of `foo:r`, the callsite `bar:r` at line 9, and the callsite `copyin(_, p, _)` at line 2. The other consists of `foo2:r`, the callsite `bar:r` at line 13, and the callsite `copyin(_, p, _)` at line 2.

8.1 Annotation overhead

One of our key design goals is to minimize annotation overhead. This section approximately measures how well MECA meets this goal by counting the number of manually-supplied annotations as well as the annotations derived from these.

Table 1 counts the number of global annotators (4) and the number of distinct places they automatically annotated (694). The most effective of these was the global system-call annotator which tainted 637 system call parameters. There were three file-scope annotators that marked 57 parameters as `!tainted` (i.e., it is illegal to call them with a tainted pointer). As stated before, the benefit of these annotators is not just the labor saved, but the fact that they eliminate the opportunity to forget an annotation.

Annotation Type	Global Annotator	Roots Generated
`tainted`	1	637
`!tainted`	3	57
Total:	4	694

Table 1: The number of global annotators for the `tainted` and `!tainted` qualifiers, and the count of the places they annotated.

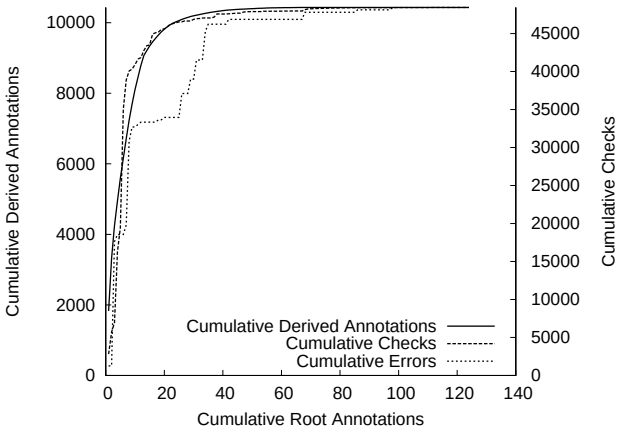

Figure 13: A cumulative view of the data presented in Table 2. There are total 154 errors messages, uniqued by file, function and line number, which is scaled up in order to fit in the graph.

Table 2 gives the number of manual annotations and counts for formal parameters and expressions. On average one manual annotation derives 147 annotations, marks 1098 expressions and leads to 682 checks. Figure 13 gives a cumulative view for the same data. It shows that 15% of the root annotations are able to derive 85% of all the derived annotations and do 85% of the total checks.

8.2 Coverage

The checker validates that every node in a propagation graph has the same annotation as all the other nodes. Larger subgraphs are better than small ones since they force more nodes to be internally consistent. Additionally, larger number of "inference events" (those that cause a pointer to be annotated as `tainted` or `!tainted`) per graph are better since they make it more likely that a labeling occurs (and is cross checked). The ideal would be two subgraphs for a system: one labeled as `tainted`, one as `!tainted` with many

	`tainted`	`!tainted`
Average Graph Size	31	18
Events per Graph	4	6

Table 3: Average subgraph size (i.e., the number of related parameters that are checked against each other) and average number of inference events per subgraph (e.g., a pointer dereference or passing a pointer to tainted function).

	tainted	!tainted	imply	Total
Manual Annotations	19	9	36	71 (7 ignore)
Derived Parameter Declarations	2498	7881	57	10436
Declarations per Manual Annotation	131	876	1.6	147
Derived Expressions	13495	64470	-	77965
Expressions per Manual Annotation	710	7163	-	1098
Checks	3329	44930	169	48428
Checks per Manual Annotation	175	4992	4.7	682

Table 2: Total number of manual annotations, and the count of derived declarations and expressions. `Derived Parameter Declarations` counts the number of unique formal parameters that were annotated as `tainted` or `!tainted`. `Derived Expressions` counts the total number of expressions that are automatically tainted or untainted. These are uniqued by file name, function name and line number. There are 71 manual annotations in total (global annotator excluded). Nearly 2/3 of them are for suppressing false positives. On average one manual annotation derives 147 annotations, 1098 expressions and does 682 checks.

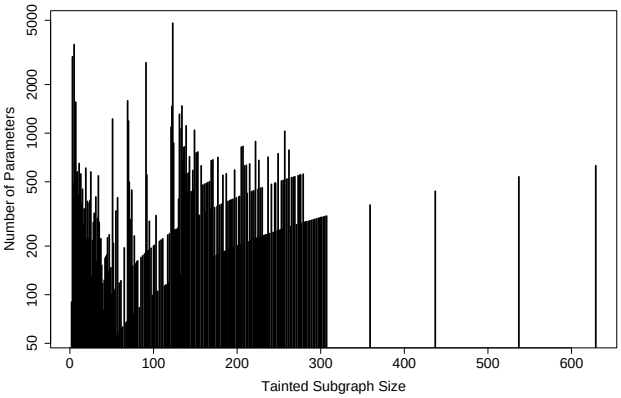

Figure 14: This graph shows the `tainted` subgraph sizes and the total number of pointers in subgraphs with this size. The rightmost four subgraphs are formed by function pointer propagation.

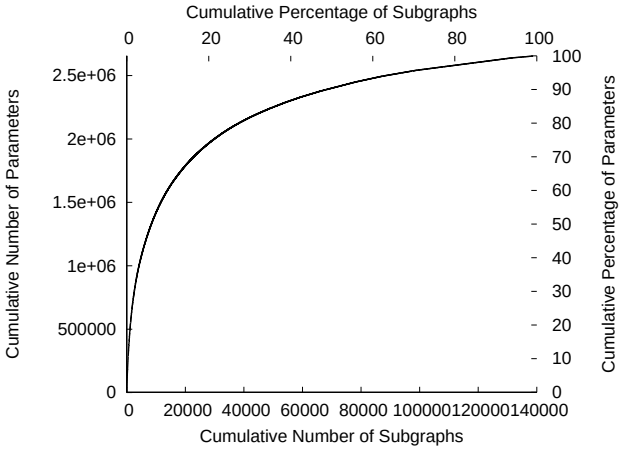

Figure 15: This graph shows the cumulative number of subgraphs and the cumulative number of pointer parameters. Subgraphs are ordered by decreasing order of the their sizes. 25% of the subgraphs contains about 75% of the total pointers annotated.

inference events in each. Table 3 shows the average sizes of the `tainted` and `!tainted` subgraphs, and the average number of tainting or untainting events per subgraph.

Figure 14 shows the tainted subgraph size and the total number of pointer parameters in subgraphs with this size. There are four enormous graphs on the right that cross-check 359, 437, 537, and 629 parameters against each other.

Figure 15 orders subgraphs labeled as `tainted` or `!tainted` by size and shows how many subgraphs are needed to cover a given percentage of pointers. For example, that 25% of the subgraphs are sufficient to label 75% of all pointers that were annotated.

8.3 Robustness

Annotation propagation must be stable and robust. Otherwise noise in the inputs such as missing manual annotations or program errors are likely to pollute the propagation graph and generate thousands of false positives or false negatives. MECA achieves robustness by massive redundancies and statistics. For example, one annotation for a function is sufficient to annotate all other functions which are assigned to the same function pointer. Statistical inference of formal parameters annotations can infer other missing annotations. We believe this is the key difference between MECA and traditional type inference.

We measure the effectiveness of function pointer propagation by computing the average node degree E/N, where E is the number of edges in the propagation graph and N is the number of nodes. This can be viewed as an approximation for how many paths can lead to a specific annotation. If this number is high, breaking one path will unlikely stop the propagation since there are many redundant paths that keep the propagation graph connected. Table 4 shows that function pointer propagation increases the average node degree by a factor of 7.

To measure the effectiveness of statistical inference of formal parameter annotations, we first run MECA with only one `tainted` annotation for function `copy_from_user`, then annotates all the missing roots inferred and run MECA again. Table 5 shows the top eleven parameters statistically inferred as described in Section 5. Bottom ranked parameters are not shown. We inspected a few of them and they are either `!tainted` parameters or non-pointers. We also use the number of distinct sources for a derived annotation as a metric for robustness. Not surprisingly, statistical in-

	without FNPTR	with FNPTR
tainted nodes	12662	14503
tainted edges	16331	250214
Average Degree	1.29	17.3
!tainted nodes	480783	542778
!tainted edges	613497	5212683
Average Degree	1.28	9.60
Total nodes	493445	557281
Total edges	629828	5462897
Average Degree	1.28	9.80

Table 4: Propagation results with and without function pointer propagation.

ference increased the average number of distinct sources by a factor of 12.

8.4 Security holes and false positives

We found 44 bugs in Linux; 19 of them allowed a malicious user to take control of the machine. Error messages are uniqued by file and functions since once a user pointer is dereferenced it tends to be dereferenced many times in one function. Table 6 shows the bugs we found, broken down by the ease and severity of exploit:

1. Arbitrary write: there were eleven cases where a user could write to arbitrary kernel memory.
2. Arbitrary read: there were eight cases where the user could read out arbitrary kernel memory (usually by passing a user-chosen pointer to a kernel print function).
3. Fault at will: there were nineteen places where a user could cause the kernel to crash in a straightforward way.
4. Always fail: these were six cases where a kernel pointer was passed to a routine that expected user pointers. These calls would always fail with an error code. Two of such errors are intentional where the return values of the "paranoid" functions are not checked intentionally.

After the annotations described at the beginning of Section 7, there were eight false positives that remained. Two of them are because in Linux "paranoid" functions sometimes can take kernel pointers when the global data segment is set to be in the kernel. Two are due to wrong pointer arithmetic propagation, where `user_base - kernel_base + kernel_pointer` which computes `user_base + offset` is considered as a kernel pointer. Two are caused by false paths. The other two are because our predicate analysis is not sophisticated enough.

To measure the effectiveness of suppressing false positives, we rerun the analysis without the `imply` annotations and the `ignore` annotations. We left the one for `struct kernel_symbols` unremoved because removing it will cause too many false positives to inspect. Not surprisingly, 98 more false positives (uniqued by file name and function name) were generated in that run.

Figure 16 gives a rare security hole in the base kernel that was found in `fs/quota.c`, which is well-tested, well audited code. Function `sys_quotactl` is a system call. The global annotator taints all of its parameters, including `special`. This tainted pointer is passed into `lookup_dev`, which dereferences it. A malicious user can trivially cause the kernel to crash or read unfortunate device memory addresses by passing in a value for `special` of their choosing.

Type	Warnings	Fixed
Arbitrary Write	11	11
Arbitrary Read	8	8
Fault at Will	19	17
Always Fail	6	3
Total	44	39
False Positives	8	

Table 6: User-pointer bugs we found in Linux 2.5.63, broken down by severity and ease of exploit.

```
/* linux-2.5.63/fs/quota.c */
asmlinkage long sys_quotactl(unsigned int cmd,
        const char *special, qid_t id, caddr_t addr) {
        bdev = lookup_bdev(special);
}

/* linux-2.5.63/fs/block_dev.c */
struct block_device *lookup_bdev(const char *path) {
        if (!path || !*path)
                return ERR_PTR(-EINVAL);
}
```

Figure 16: A security hole in fs/quota.c. Only relevant code is shown

Cross-checking. Cross checking by propagating annotations across function pointers was extremely effective. Security errors seem to cluster: if a programmer is unaware of an interface rule (e.g., that a parameter should be tainted) they stay unaware, blithely violating the rule. For Linux, all eleven of the write bugs and four of the read bugs were found by propagating annotations across functions assigned to the same function pointer. For most of these bugs, there was not a single check in the functions (or even the files) that contain the bugs.

Figure 17 gives a representative example. The function `sg_read` taints its second argument `buf` using a call to function `verify_area`. The function `sg_read` is also statically assigned to the field `read` in a structure of type `file_operations`. This will cause the system to taint the second argument in all other functions assigned to a structure of this type. In our example, this happens when the function `do_read` is assigned to the variable `Divas_fops`. This taints `do_reads` second argument; this annotation is then propagated to the variable `ClientLogBuffer`, which causes an error when it is passed to `memcpy`. Interestingly, even the user pointer itself has the name `pClientLogBuffer` and `pUserBuffer`.

9. RELATED WORK

There have been numerous annotation languages designed for program checkers. Systems that are most related to MECA are Splint [9], CQual [12], and ESC/Java [16].

Splint is an annotation based tool for detecting a variety of programming errors such as null-pointer dereferences and potential buffer overrun vulnerabilities. It employs a simple flow-sensitive analysis assisted by user provided annotations that is fast and scalable. However, Splint has the following drawbacks that limits its usefulness in effectively checking large systems. First, its annotation propagation is

Formal Parameter	Annotation Rank	p-value Rank	Utility	Actual Parameters		
				`tainted`	`!tainted`	`unknown`
`copy_to_user`:1	1	2	496	931	8	496
`put_user`:2	2	1	422	644	1	422
`get_user`:2	3	3	354	347	1	354
`__put_user`:2	4	8	125.9	33	0	126
`verify_area`:2	5	5	84	103	0	84
`__get_user`:2	6	9	58.9	18	2	59
`__copy_from_user`:2	7	18	37.6	8	0	44
`__copy_to_user`:1	8	19	33.1	7	0	43
`access_ok`:2	9	4	26	81	1	26
`__user_walk`:1	10	80	11	3	0	27
`clear_user`:1	11	10	6.9	6	0	7

Table 5: Annotation ranking using statistical inference and annotation utility.

more limited than MECA (e.g., it lacks true inter-procedural propagation). Second, except for a small set of pre-defined operators, Splint only allows unary predicates for expressing program properties rather than n-ary predicates (§ 3.1.1). Finally, other than the "ignore" primitive which essentially turns off checking for a segment of the target code, Splint provides few means of systematic suppression of false positives. MECA, on the other hand, allows extension-specific suppression using hints from user-provided annotations (e.g. the `from_user ==> tainted` example in Section 3).

CQual is a type-based analysis tool for defining, inferring, and checking flow-sensitive type qualifiers in C programs. It employs an efficient constraint-based type inference algorithm to propagate user provided information to minimize manual annotation. It is more ambitious than MECA in that it is sound by design. However, because it forces soundness, it must always use conservative alias analysis, which can give many false positives. Extra user-provided alias specifications are needed in order to suppress a large portion of those false positives. The large amount of work required limits the applicability of CQual on large existing systems [23]. Furthermore, it only supports unary type qualifiers, which limits the properties it can express.

ESC/Java descends from the intellectual tradition of program verification. It allows users to write arbitrary first-order logic formulas for annotations, which it checks using an automatic theorem prover. Its annotation language can express a significantly richer set of concepts than MECA. However, it appears that in practice MECA can match much of this power because it lets checkers define their own builtin predicates (§ 3.2). ESC/Java lacks extensibility in terms of defining new checks and it appears that MECA applies much more easily to large code bases than ESC/Java does. While ESC/Java has a high annotation burden, recent work on Houdini [11] has shown how to use annotation templates to automatically derive ESC annotations. One difference between our approach and theirs is that our use of statistical inference allows us to handle noisier samples when deriving our annotations.

The GCC attribute extension allows users to annotate declarations in C programs. The main objective of the GCC attributes is to provide the GCC compiler with hints for error reporting (mostly for syntax and type errors detected in the frontend) and optimization purposes.

Zhang *et al* [23] modified GCC and used a Perl script to annotate all the local variables of certain types to be "unchecked." This can be viewed as hardwired programmatic annotation. Compared with their approach, our system allows programmers to easily write such programmatic annotations in the source without any compiler knowledge.

MOPS [3] is another system that checks for security properties, which is loosely related to MECA. It uses finite state automatons to represent security rules, slices a program based on the state transitions specified in these rules, transforms the sliced program into a pushdown automaton and uses model checking techniques to check for security errors and verify their absence. Compared with MECA, it provides no annotation support for programmers to express general security rules in the source code. The range of the security rules it can check seems limited since it does no dataflow analysis except simple syntactical matching on variable names.

The taintedness problem in Section 7 has been explored in [1, 12, 9]. Capability checking has been explored in [7].

10. CONCLUSION

This paper has described a system and language for expressing and checking general security rules.

The annotation language is expressive and direct. It gives programmers novel powers. One is the ability to write programmatic annotations that automatically annotate a large bodies of source code. Another is the ability to use computationally flexible predicates to control whether an annotation is applied. We used this ability to handle kernel backdoors and other false-positive inducing constructs.

The system is tailored for getting results on real systems. It is designed to make it easy to suppress false positives. Additionally, its propagation abilities mean that a single manual annotation leads to many derived annotations (e.g., hundreds in our experiments) freeing programmers from the crushing manual effort of most traditional systems.

The system is effective. Our most through case study was a user-pointer checker that used 75 annotations to check thousands of declarations in millions of lines of code in the Linux system. It found over forty errors, many of which were serious, while only having eight false positives.

While the system is still a prototype, our initial experiences indicate that it can give significant traction when checking large bodies of real code.

```
/* linux-2.5.63/drivers/scsi/sg.c */
static ssize_t
sg_read(struct file *filp, char *buf, size_t count, loff_t * ppos) {
    // [META]: taints second argument buf
    if ((k = verify_area(VERIFY_WRITE, buf, count)))
        return k;
}
static struct file_operations sg_fops = {
   /* Assigns: sg_read to the read field in file_operations. Since
    * the second parameter of sg_read is tainted (from the code
    * above) this will taint the second parameter of all
    * functions assigned to this field. */
   .read = sg_read,
   .write = sg_write,
   .poll = sg_poll,
   .ioctl = sg_ioctl,
   ...
};
/* linux-2.5.63/drivers/isdn/eicon/lincfg.c */
struct file_operations Divas_fops;
int DivasCardsDiscover(void) {
   /* Assign do_read to the read field in file_operations: causes
    * its parameter to be marked as tainted. */
   Divas_fops.read = do_read;
}
/* linux-2.5.63/drivers/isdn/eicon/linchr.c */
ssize_t do_read(struct file *pFile, char *pUserBuffer,
size_t BufferSize, loff_t *pOffset)
{
    /* pUserBuffer tainted from function pointer prop. */
    klog_t *pClientLogBuffer = (klog_t *) pUserBuffer;
    if (pHeadItem) {
       /* ERROR: dereferencing tainted pointer. */
       memcpy(pClientLogBuffer, pHeadItem, sizeof(klog_t));
}
```

Figure 17: A security hole found by cross checking through file_operations.read. Only relevant code is shown

Acknowledgements

This research was supported in part by DARPA contract MDA904-98-C-A933 and by a grant from the Stanford Networking Research Center. Dawson Engler is partially supported by an NSF Career Award and Ted Kremenek received funding from an NSF Graduate Fellowship. We are also grateful for helpful comments from Xiaowei Yang, Ken Ashcraft, and the anonymous reviewers.

11. REFERENCES

[1] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2002.

[2] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing systems*, pages 131–152, Spring 1996.

[3] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 235 – 244. ACM Press, 2002.

[4] A. Chou. *Static Analysis for Bug Finding in Systems Software*. PhD thesis, Stanford University, 2003.

[5] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, June 2001.

[6] N. Dor, M. Rodeh, and S. Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *8th International Symposium on Static Analysis (SAS)*, pages 194–212, July 2001.

[7] A. Edwards, T. Jaeger, and X. Zhang. Runtime verification of authorization hook placement for the linux security modules framework. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 225–234. ACM Press, 2002.

[8] D. Engler, D. Yu Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.

[9] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, January/February 2002.

[10] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–232, 2000.

[11] C. Flanagan, K. Rustan, and M. Leino. Houdini, an annotation assistant for ESC/Java. In *Symposium of Formal Methods Europe*, pages 500–517, Mar. 2001.

[12] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, June 2002.

[13] D. Freedman, R. Pisani, and R. Purves. *Statistics*. W.W. Norton, third edition edition, 1998.

[14] T. Kremenek and D. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *10th Annual International Static Analysis Symposium*, 2003.

[15] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium, Washington, D. C.*, Aug. 2001.

[16] K. R. M. Leino, G. Nelson, and J. Saxe. ESC/Java user's manual. Technical note 2000-002, Compaq Systems Research Center, Oct. 2001.

[17] A. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 129–142, Oct. 1997.

[18] J. Pincus. Personal communication. Developing a buffer overflow checker in PREfast (a version of of PREfix)., Oct. 2001.

[19] S. M. Ross. *Probability Models*. Academic Press, London, UK, sixth edition, 1997.

[20] J. Viega, J. Bloch, T. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *Annual Computer Security Applications Conference*, 2000.

[21] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, 2001.

[22] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *The 2000 Network and Distributed Systems Security Conference. San Diego, CA*, Feb. 2000.

[23] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, pages 33–48, Aug. 2002.