Modeling and Visualizing Security Properties of Code using Dependence Graphs*

John Wilander
Dept. of Computer and Information Science
Linköpings universitet
johwi@ida.liu.se

ABSTRACT

In this paper we discuss the problem of modeling security properties, including what we call the *dual modeling problem*, and ranking of potential vulnerabilities. The discussion is based on the results of a brief survey of eight existing static analysis tools and our own experience. We propose dependence graphs decorated with type and range information as a generic way of modeling security properties of code. These models can be used to characterize both good and bad programming practice as shown by our examples. They can also be used to visually explain code properties to the programmer. Finally, they can be used for pattern matching in static security analysis of code.

1. INTRODUCTION

According to statistics from CERT Coordination Center, CERT/CC, in year 2004 more than ten new security vulnerabilities were reported per day in commercial and open source software [3]. In addition, the 2004 E-Crime Watch Survey respondents say that e-crime cost their organizations approximately $666 million in 2003 [8]. One way of countermeasuring these problems is using security tools to find the vulnerabilities already during software development.

In recent years a lot of research has been done in the field of static analysis for security testing. This research has resulted in several tools and prototypes based on various techniques, models and user involvement. Some of them are publicly available, some are not.

In November 2002 we published a comparative study of five tools publicly available at the time [30]. We used micro benchmarks and our study showed that tools performing lexical analysis produced a lot of false positives (52% to 71%), while syntactical and semantical analysis had problems with too many false negatives (70% to 73%). The latter mainly due to poor vulnerability databases, not the underlying techniques.

Since then many more tools have been developed. Although the research behind these tools and prototypes is often excellent and the empirical results are promising, it is not evident if and how the techniques can be combined to solve several security problems at once. They all focus on one or two categories of security properties each and make use of quite different system models, methods of analysis, and also require different amounts of user or programmer involvement. Further, to our knowledge there is no thorough study of the problems in modeling security properties that underlie static analysis.

*This work has been supported by the national computer graduate school in computer science (CUGS) commissioned by the Swedish government and the board of education.

1.1 Paper Overview

In Section 2 we present related work by doing a brief survey of eight existing static analysis tools performing syntactical and semantical static analysis to check security properties. A summary defines the problems we want to solve.

Graph models of security properties in code as a mean for visual communication with programmers is discussed in Section 3. Section 4 provides a definition and discussion of the *dual modeling problem* in the context of security properties in code. Criteria for severity ranking of security vulnerabilities are listed in Section 5.

In Section 6 we propose a generic modeling formalism for code security properties covering control-flow, data-flow, type and range information. Models of two security vulnerability types—integer flaws and double `free()` are explained in Section 7 and serve as examples of how the modeling formalism can be used.

Sections 8 and 9 discuss future work and provide our conclusions.

2. SURVEY OF STATIC ANALYSIS TOOLS

Static analysis tools try to prevent attacks by finding the security vulnerabilities in the source code so that the programmer can remove them. The two main drawbacks of this approach is that someone has to keep an updated database over programming flaws to test for, and since the tools only *detect* vulnerabilities the user has to know how to fix the problem. This paper tries to address these two drawbacks by proposing a way to model security properties of code that allows for both effective static analysis and visual communication with the programmer.

Several tools perform a deep analysis on a syntactical and semantical level. We have found eight such tools, all analyzing C code—Splint, BOON, CQual, Metal/xgcc, MOPS, IPSSA, Mjolnir, and Eau Claire. As some of these tools are still being developed and some are not even available as prototypes we do not know to what extent they are used in practice.

2.1 Splint

Secure Programming Lint or *Splint* was implemented by David Larochelle and David Evans [17].

Their approach is to use programmer provided semantic comments, so called *annotations*, to perform static analysis, making use of the program's parse tree. The annotations specify function constraints in the program—what a function *requires* and *ensures*.

Low level constraints are first *generated* at the subexpression level (i.e. they are not defined by annotations). Then statement constraints are generated by co-joining these subexpression constraints, assuming that two different subexpressions cannot change the same data. The generated constraints are then matched with the annotated constraints to determine if the latter hold. Splint only performs intraprocedural data-flow analysis, and the control-flow analysis is limited.

2.2 BOON

David Wagner et al presented *Buffer Overrun detectiON*, or *BOON*, aiming for detection of buffer overflow vulnerabilities [25]. In July 2002 a prototype was publicly released under the name . Under the assumption that most buffer overflows are in string buffers they model string variables (i.e. the string buffers) as abstract data types consisting of the allocated size and the number of bytes currently in use. Then all string functions are modeled in terms of their effects on these two properties. Analysis is carried out by solving integer range constraints.

BOON reports any detected vulnerabilities as belonging to one of three categories, namely "Almost certainly a buffer overflow", "Possibly a buffer overflow" and "Slight chance of a buffer overflow".

2.3 Cqual

The tool *Cqual* uses constraint-based type inference [11]. It traverses the program's abstract syntax tree and generates constraints that capture the relations between type qualifiers. A solution to the constraints gives a valid assignment of type qualifiers to the variables in the program. If the constraints have no solution, then there is a potential bug.

Umesh Shankar et al have used Cqual to find format string vulnerabilities [24]. They add a new C type qualifier called *tainted* to tag data that has originated from an untrustworthy source (Cqual requires the user to manually tag untrustworthy data sources). Then they set up typing rules so that tainted data will be propagated, keeping its tag. If tainted data is used as a format string the tester is warned.

The same tainted functionality was used by Chen et al to statically find implicit type cast errors constituting security vulnerabilities [5]. Johnson and Wagner are using Cqual to check for insecure pointer handling between kernel and user-space in Linux [15].

2.4 Metal and xgcc

Ashcraft and Engler have done security research in the area of meta-level compilation. With their compiler extension *xgcc* and extension language *Metal* they have statically analyzed code for input validation errors on integer variables [1]. C programs are modeled as control-flow graphs and are analyzed path by path.

By formulating rules in Metal they check that integer values coming from untrusted sources are bounds checked before they are used in any sensitive function. The security bugs found are unvalidated integers used in pointer arithmetic, and integer overflows. Memory management errors (`malloc()`/`free()`) were also found but not substantially analyzed.

Potential bugs found are ranked by properties such as local vs global scope, distance in lines of code, and non-aliased vs aliased variables.

2.5 MOPS

Chen and Wagner have designed a static analysis tool called MOPS which checks ordering constraints [4]. Some security bugs can be described in terms of temporal safety properties. MOPS specifically checks dropping of privileges and race conditions in file accesses. C programs are modeled as *pushdown automata*, and the security properties are modeled as *finite state automata*. Security models can be combined into complex security properties.

No data-flow, pointer, or aliasing analysis is done, which is justifiable since only temporal properties are checked.

2.6 IPSSA

Livshits and Lam have defined and used an extended intermediate form for finding buffer overflow and format string bugs [18]. Their program model builds on *static single assignment (SSA)* form—an intermediate code representation that separates values operated on from the locations they are stored in which is very useful in for instance optimization [19]. The extension, called *IPSSA*, provides interprocedural definition-use information with indirect memory accesses via pointers. It can then be used to perform static analysis that handles pointer and aliasing analysis. Security properties are modeled using a "small special-purpose language designed for the purpose". While technical details of this special-purpose language are lacking their empirical results are very promising, especially the low rate of false positives. The solution was chosen to be unsound for scalability reasons.

2.7 Mjolnir

Weber et al have presented a tool called *Mjolnir* which makes use of dependence graphs and constraint solving to find buffer overflows in C code [27]. They represent buffers with the same range variables used in BOON (see Section 2.2), build system dependence graphs, decorate them with range constraints based on the semantics of C string library functions, and finally solve the constraint sets.

To decorate the dependence graphs they traverse the program bottom-up and generate summary nodes containing the constraints of the current function and all its callees.

Weber et al do not clearly state how safety constraints are

Table 1: Overview of static analysis tools checking C code for various security properties. "Intra" and "Inter" refers to intra- or interprocedural analysis, "Alias" means data aliasing, "Ptr" means pointer analysis, "Type" means type and type conversion information, and "Annot" means code annotations.

	Control-flow		Data-flow					Annot
Tool	Intra	Inter	Intra	Inter	Alias	Ptr	Type	
Splint	x		x		x			x
BOON			x	x				
Cqual			x		x		x	x
MOPS	x	x						
Metal/xgcc	x	x	x	x				
IPSSA	x	x	x	x	x	x	x	
Mjolnir	x	x	x	x	x			
Eau Claire	x	x	x	x				

generated, but we assume they generate them only for statically allocated buffers. They provide both control-flow insensitive and control-flow sensitive constraint generation. Although global variables normally are handled in dependence graphs (see Section 6.1) they are not handled by Mjolnir. No pointer analysis is done.

2.8 Eau Claire

In spring 2002 Brian Chess presented his tool *Eau Claire* [6]. The tool translates C code into so called *guarded commands*, enhanced with exceptions, assertions, assume statements, and erroneous states. Vulnerabilities are modeled using the ESC/Modula2 specification language where you define what a function requires, modifies, and ensures. Eau Claire then augments guarded commands with the specifications. The outcome is a set of verification conditions which are processed by an automatic theorem prover to find potential violations.

Shortcomings of Eau Claire's static analysis are the conservative approach to pointer dereferences (it assumes that any two pointers of the same type may reference the same location) and references into structures and unions. Type-based vulnerabilities are not targeted by Eau Claire [7].

2.9 Summary

Tables 1 and 2 summarize the properties and features of the tools above.

We conclude that several categories of security properties can be statically checked but there is need of a generic solution. The first step toward such a solution is to define a modeling formalism that both covers all necessary aspects and allows for static analysis.

Two other key issues are that such a solution has to allow for effective feedback to the programmers who have to fix the security problems, and it has to support intuitive modeling of new security properties for effective updates of the database. None of the tools presented above have any other kind of input or feedback than text.

We require that the modeling formalism can:

- visually communicate with programmers who model or fix security problems in code (Section 3);
- model several types of security properties (Section 4);
- rank the severity of potential flaws (Section 5); and
- take into account data-flow, control-flow, type and range information, and combinations thereof (Section 6).

3. THE NEED FOR VISUAL MODELS

As mentioned in Section 2 the two main drawbacks of static analysis tools are that someone has to keep an updated database over programming flaws to test for, and since the tools only *detect* vulnerabilities the user has to know how to fix the problem.

Current tools such as the ones briefly presented in Section 2 use textual models of security properties in their databases to give textual feedback to the user. For example Splint gives output in the following manner:

```
bounds.c:9: Possible out-of-bounds store:
  strcpy(str, tmp)
  Unable to resolve constraint:
  requires maxSet(str @ bounds.c:9) >=
  maxRead(getenv("MYENV") @ bounds.c:7)
   needed to satisfy precondition:
  requires maxSet(str @ bounds.c:9) >=
  maxRead(tmp @ bounds.c:9)
   derived from strcpy precondition: requires
  maxSet(<parameter 1>) >=
  maxRead(<parameter 2>)
```

Just as call graphs and and flow graphs can help programmers understand code in general (Grammatech's tool "CodeSurfer" is a perfect example [12]), visual models and graph representations of security properties can help to understand and fix security flaws. Especially when the flaws include interprocedural data- and control-flow dependencies.

4. THE DUAL MODELING PROBLEM

A common issue in security modeling is what we call the *dual modeling problem*—the problem of modeling malicious or benign things. When modeling security properties of code we need both kinds—models of bad programming practice, and models of good programming practice.

Table 2: Overview of static analysis tools checking C code for various security properties (cont.). The program models are control-flow graph (CFG), abstract syntax tree (AST), push-down automata (PDA), parse tree (PST), static single assignment (SSA), system dependence graph (SDG), guarded commands (GC). The property models are constraint based (CB), finite state automata (FSA), "Metal" (MET), ESC/Modula2 specification language (ESC), and other, special purpose modeling (OTH).

	Program model							Security property model				
Tool	CFG	AST	PDA	PST	SSA	SDG	GC	CSB	FSA	MET	ESC	OTH
Splint	x							x				
BOON				x				x				
Cqual		x						x				
MOPS			x						x			
Metal/xgcc	x									x		
IPSSA					x							x
Mjolnir						x		x				
Eau Claire							x				x	

In a seminal paper from 1977 Leslie Lamport describes a formalism closely related to the dual modeling problem—a property stating that nothing bad happens during execution is called a *safety property*, and a property stating that something good (eventually) happens during execution is called a *liveness property* [16].

Typical for a safety property is that we can detect a property violation between one execution step and another. During execution we can look ahead and see if the next execution step will take us into a bad state and in such a case raise an alarm or terminate execution. All run-time security measures such as intrusion detection systems and anti-virus applications detect safety properties—they either try to match with known bad behavior, or they monitor for deviations from good behavior.

In the case of a liveness property we can only detect property violations at termination since during execution, we never know whether the good thing will eventually happen or not. Fulfilling the liveness property could potentially be the last execution step before termination. Therefore we cannot rely on run-time monitoring to countermeasure security vulnerabilities that are violations of liveness properties. Static methods, on the contrary, can look into the "future" by following possible execution paths all the way to termination, and try check if a program satisfies a liveness property.

However, models of good or bad programming practice do not correspond directly to safety and liveness properties. Instead they can be a combination of safety and liveness as explained in Section 4.1 and 4.2.

A comprehensive discussion on this fundamental difference between safety and liveness security properties can be found in Schneider's paper "Enforceable Security Policies" [23].

4.1 Modeling Good Security Properties

Some security properties of code are typically described as "If you do A you must do B". These properties are best modeled as good programming practice—"do like this".

An example is *input validation* of integers. When an integer can be affected by input from users, files, the network et cetera it has to be validated before affecting any memory pointer via type-casting, array references, pointer arithmetic, or the like. Otherwise the pointer may reference unintended memory areas leading to arbitrary behavior or even full compromise of the process.

While being a model of good programming practice correct input validation is both a liveness property (external input must eventually be validated assuming it will be used sometime), and a safety property (no sensitive use of external input without validation).

4.2 Modeling Bad Security Properties

Some security problems are typically described as "If you do A then you must not do B". Such properties are best modeled as bad programming practice—"do not do like this".

An example of such a problem is the double `free()` vulnerability. Freeing the same memory chunk twice or more may open up for heap corruption attacks.

Trying to model all possible benign ways of freeing memory is infeasible since that would be the same as building complete models of all well-behaved programs using `free()`. A model of a bug, however, covers all cases. The absence of multiple `free()` is a safety property.

5. RANKING OF POTENTIAL VULNERABILITIES

Engler and Musuvathi have clearly pointed out the problem of reporting huge amounts of potential bugs as the result of static analysis and model checking—"It's not enough to find a lot of bugs. (...) What users really want is to find the 5-10 bugs that really matter ..." [20]. Based on our knowledge and experience on static analysis we propose using the following information from the analysis to generate severity ranking:

- Pointer analysis is a hard problem to solve accurately and thus the risk for false positives increases with the amount of such analysis. Therefore we propose that the more pointer analysis involved in finding a flaw, the lower the ranking.

```
void func() {
  int sum=0, i=1;
  while(i<11) {
    sum=sum+i;
    i=i+1; }
  printf("%d\n",sum);
  print("%d\n",i); }
```

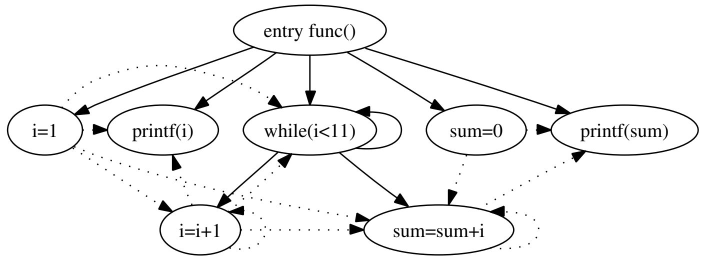

Figure 1: A small C function (left) with its corresponding program dependence graph (right). Solid arrows represent control-flow dependence, dotted arrows represent data-flow dependence. All dependencies are transitive (if $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$).

- Aliasing is another problem in static analysis. Because of potential inaccuracy in the analysis we therefore propose that the more aliasing involved in finding a flaw, the lower the ranking.

- Interprocedural control-flow may result in infeasible execution paths being analyzed. Again, because of potential inaccuracy in the analysis, flaws involving interprocedural analysis are ranked lower than intraprocedural ones.

- Flaws involving implicit events are ranked higher than explicit ones since implicity imposes a higher risk for unintended behavior. An example of this is implicit versus explicit type-casts.

5.1 Using the Dual Model for Ranking

In some cases we can make use of modeling both good and bad programming practice. If we have reached a concise description of a property in one distinct model, the dual of that model often explodes into several cases.

For instance, in the case of implicit type-casting and integer signedness vulnerabilities a model of good programming practice is to validate the integer and to have no implicit type-casts at any use points (this example is explained in detailed in Section 7.1).

The dual of this model contains several ways of violating the property. Various narrowing type-casts and missing validation points can be combined. The benefit of exploding the dual and creating all these models is that we can possibly rank them in terms of severity. Perhaps a certain violation is definitely a security vulnerability, whereas another violation only *might* be vulnerable.

6. A MORE GENERIC MODELING FORMALISM

To meet the requirements listed in Section 2.9 we propose decorated dependence graphs as a more generic formalism for visualizing and modeling security properties, and performing static analysis. We here present intraprocedural and interprocedural dependence graphs, decorated with range and type information. We end the section with a view on possible analysis techniques.

6.1 Program Dependence Graphs

Dependence graphs were first presented by Ottenstein and Ottenstein as an intraprocedural intermediate form—the *program dependence graph*, or *PDG* [21]. While originally generated for procedural languages such as C, algorithms generating dependence graphs for object oriented languages exist, e.g. Java [26]

A dependence graph is an intermediate representation of code where vertices represent statements and predicates (henceforth called *program points*), and edges represent control- and data-flow *dependence*. This means that only necessary temporal constraints are encoded in the graph—it does not include a complete control-flow graph.

A program point B is *control dependent* on another program point A, if A controls whether B is executed or not. Formally A is the first program point not *post-dominated* by B when traversing the control-flow graph backward from B. Informally we can say that program point A is a conditional and B is executed in only one of A's outgoing paths.

A program point B is *data dependent* on a program point A if some variable x is defined in A and later used in B without any new defines in-between. Data dependence can also be in form of definition order. Figure 1 shows a small C function with its corresponding program dependence graph.

6.2 System Dependence Graphs

The interprocedural version, called *system dependence graph*, or *SDG*, was presented by Horwitz *et al* [13]. To generate the SDG we need to encode data- and control-flow dependence between procedures which includes formal and actual parameters, formal and actual return values, and global variables.

A procedure call from procedure A to procedure B is modeled with a call vertex in A, an entry vertex in B, and an interprocedural control dependence edge between them. Parameters are handled with actual-in and actual-out vertices in A, formal-in and formal-out vertices in B, and interprocedural data dependence edges connecting them. Temporary variables are used for parameter passing by value-result. If a procedure uses a global variable, it is treated as a (hidden) input parameter, and is encoded as additional actual-in and formal-in vertices. For further information on summary edges for avoiding calling context problems see the original paper [13].

```
void func1(char *dest, char *src,
           int len) {
  if(len<MAX)
    memcpy(dest, scr, len); }
```

Figure 2: Implicit type-cast flaw (`len` casted to unsigned int in the call to `memcpy()`).

```
void func2(unsigned int size) {
  char *buf =
       (char *) malloc(size+1);
}
```

Figure 3: Integer overflow flaw (adding one to `size` may cause overflow).

6.3 Range Constraints in SDGs

Weber *et al* have used decorated SDGs to statically detect buffer overflow vulnerabilities [27]. The graph is augmented with range constraint information for string buffers. Each PDG contains a summary vertex with range constraints of the procedure and all its callees.

```
void copy(char *src) {
  char dst[10];
  strcpy(dst, src); }
```

Figure 4: The PDG for the code above would have a range constraint node summary node saying Len(`src`) $\subseteq$ Len(`dst`).

6.4 Type Information in SDGs

Several so called *narrowing integral type-casts* have constituted security vulnerabilities. Chen *et al* have studied this category of security bugs and summarized the insecure conversions [5].

We propose that the original SDGs be decorated with type information, specifically implicit type conversions. Type conversion information should belong to edges in the SDG since it is the data-flow between two program points that can include such a conversion, and a program point can be data-flow dependent on several others. See Figure 5 for examples of this decoration.

6.5 Static Analysis Using SDGs

Dependence graphs were designed to allow for deep analysis of code. They are the underlying structure for *program slicing* and *chopping* and are used for optimization [10].

A program slice is the parts of a program that can affect the value of a chosen program point, the *slicing criterion*. *Static slicing*, invented by Weiser [28], was defined as a reachability problem in PDGs by Ottenstein and Ottenstein [21]. Interprocedural slices can be computed in a similar way in SDGs.

The combination of two (or more) program points, potentially a point with (malicious) user input, and a point with a vulnerability, allows for program chopping—a technique presented by Reps *et al* [22]. When chopping we want to know how some source points affect some target points.

Slices and chops of programs can help with understanding the cause of a vulnerability since they show exactly what parts of the program affect the execution of the vulnerable program point. The richness of program information found in SDGs together with slicing, chopping, type inference and range analysis means it covers all the features of the tools surveyed in Section 2 and provides visual communication with the user via a graph representation of the original code.

7. MODELING SECURITY PROPERTIES

In this section we show how four security properties can be modeled in terms of decorated dependence graphs. We show the use of dual models both for benign and malicious properties, and ranking of potential flaws. Our proposed formalism is not limited to these properties; they simply serve as examples.

In the graphs all edges represent interprocedural transitive dependence—solid arrows for control-flow, and dotted arrows for data-flow.

7.1 Integer Flaws

Handling integers may seem harmless and straight forward. But several security vulnerabilities prove this a difficult area. The problems mostly arise when integers are used as memory offsets, in pointer arithmetic, and when the integer representation changes from signed to unsigned or vice versa. For proper input validation in such sensitive cases, two crucial steps need to be taken; (1) validate integral variables so that narrowing type-casts do not lead to unintended behavior, and (2) validate upper and lower bounds of user affected integral variables before they are used in memory references and calculations.

We are now able to encode the first correct code pattern in terms of our decorated dependence graphs (see Fig. 6). The nodes are program points where "ext input" means external input, "def" means a variable is defined, "val" means a variable is validated, and "use" means a variable is used. The input has to be validated before it is used which means that the use point has to be control dependent on the validation point. Modeling of validation points is abstracted away from these models. Using range constraints is a feasible way of doing this [1].

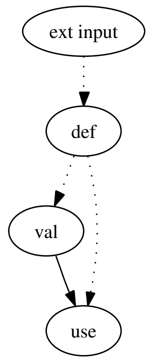

Figure 6: Correct code pattern for integer input validation.

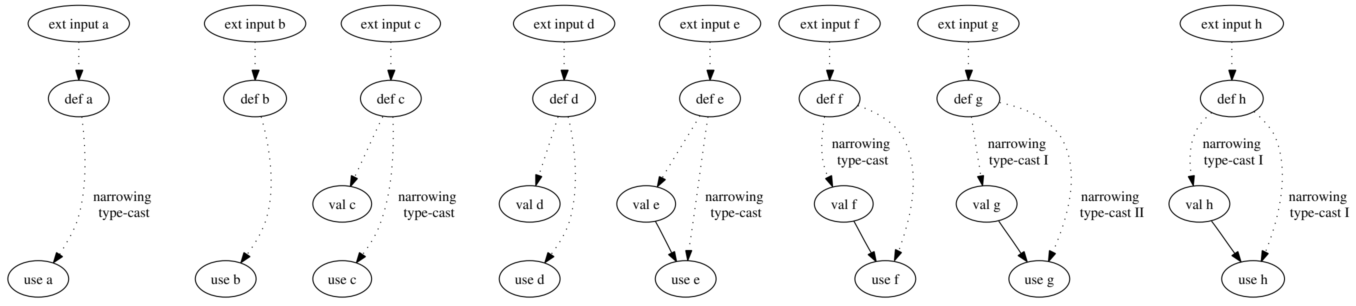

Figure 5: Eight incorrect graph patterns for integer validation. The nodes are program points representing external input (ext input), definition of a variable (def), validation of the variable (val), and use of the variable (use). "narrowing type-cast I" and "narrowing type-cast II" means two *different* type-casts. The proposed severity ranking from left to right is explained in Section 7.2 below.

Deviations from this good programming practice, i.e. integer security bugs, have been studied by Blexim [2], Howard [14], and Ashcraft and Engler [1] and we here briefly present the bug types they have identified:

7.1.1 Integer Signedness Errors.

Integer signedness errors can arise both due to implicit type-casting and insufficient validation. In Fig. 2 the signed integer `len` can be negative and as such pass the (inadequate) validation point. When calling `memcpy()` an implicit narrowing type-cast to `size_t` (unsigned integer) occurs which will convert a negative integer to a huge positive integer, possibly overflowing the destination buffer `dest`.

7.1.2 Integer Overflow/Underflow.

When an unsigned integer has reached the maximum value it can represent, an increment to that integer will make it wrap around and become zero. Decrementing an unsigned integer below zero will result in the maximum value.

In Fig. 3 the intent is to allocate the requested memory plus space for a null terminator. If `size` was the maximum unsigned integer possible, adding one will make it wrap around and call `malloc()` with zero as argument. The return value in such a case is either a null pointer or a non-null pointer that must not be used. Dereferencing such a non-null pointer may allow for heap corruption.

7.1.3 Integer Input Validation.

When an integer can be affected by input from users, files, network et cetera it has to be validated before affecting any memory pointer via type-casting, array references, pointer arithmetic, or the like. Otherwise the pointer may reference unintended memory areas leading to arbitrary behavior or even full compromise of the process.

7.2 Modeling Integer Flaws

To allow for severity ranking we can encode the dual to the correct code pattern, ending up with a collection of incorrect code patterns, i.e. models of bad programming practice (see Fig. 5). Using the ranking rule for implicity (see Section 5) we rank the incorrect code patterns in descending order as follows:

1. Missing validation and narrowing type-cast
2. Missing validation but no narrowing type-cast
3. Use not control dependent on validation and narrowing type-cast
4. Use not control dependent on validation but no narrowing type-cast
5. Narrowing type-cast on either validation or use (two graphs in Fig. 5)
6. Different narrowing type-casts on validation and use
7. Same narrowing type-casts on validation and use

7.3 The Double Free Flaw

Often "normal" bugs turn out to be tools for attackers. This is the case of *double free.* To allocate heap memory, the program calls `malloc()` and gets a pointer to the allocated memory as return value. When the program is done using the memory it has to be released, which is done with a call to `free()`.

To keep track of which parts of heap memory are allocated and which are free, the operating system has to store information. For scalability reasons this information is stored together with each allocated chunk of memory; it is stored "in-band". When memory is freed the in-band information is used to relink the memory chunk with the list of free memory.

Normally, attempting to free the same memory twice or more will lead to undefined behavior, often a *segmentation fault.* But if an attacker can change the memory in between two calls to `free()` he or she can inject false in-band information and potentially compromise the process.

This is an example of a model of a bad security property (see Fig. 7). We show in Fig. 8 and 9 why the double free has to be modeled as a bad security property. The bad model *contains* the good one. Thus we cannot say a piece of code is secure simply because we have pattern matched a good use of `free()`; we also have to look for bad use of `free()`.

```
char *buf = (char *) malloc(SIZE);
...
free(buf);
...
free(buf);
```

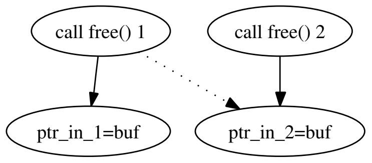

Figure 7: Incorrect code pattern for `free()` and the corresponding dependence graph. If there had been a new call to `malloc()` in-between the two calls to `free()` there would not have been a data dependency edge between the first call to `free()` and the second pointer to `buf` in the graph.

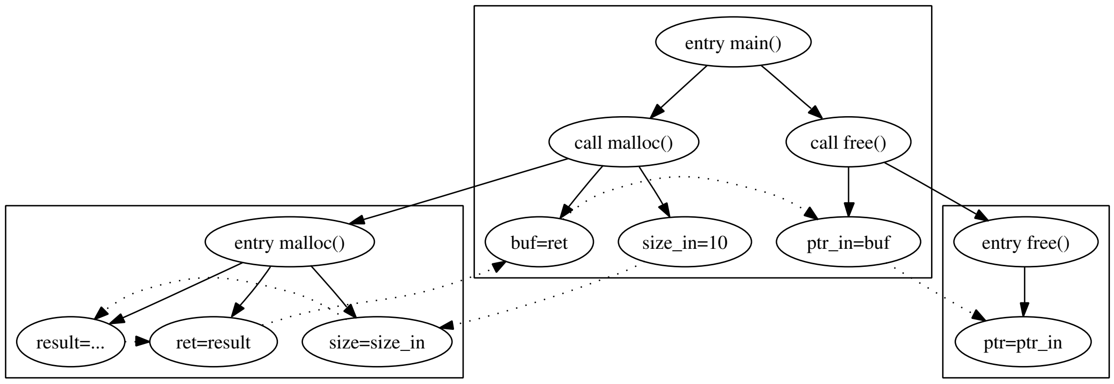

Figure 8: Correct graph pattern for `malloc()` and `free()`.

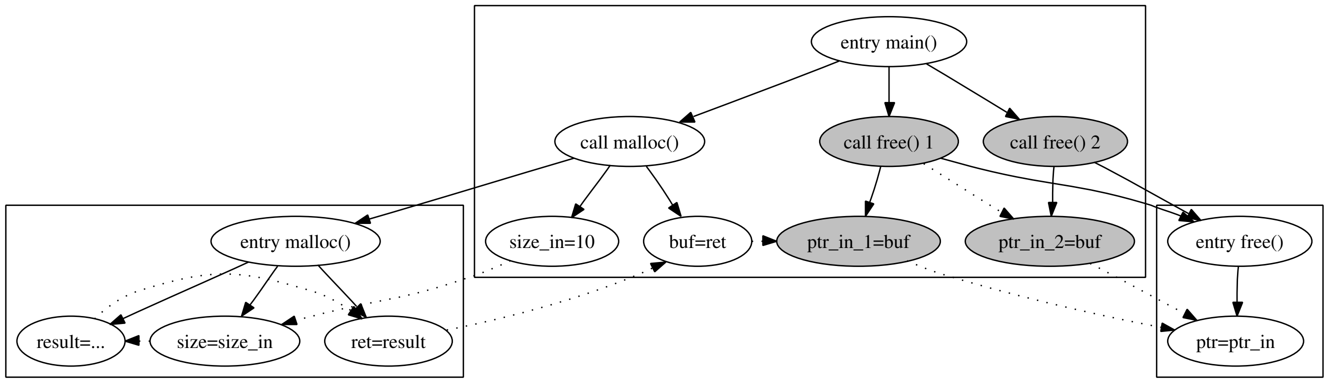

Figure 9: Incorrect graph pattern for `malloc()` and `free()`, where `free()` is called twice. Notice how the grey nodes in the `main()` box match the incorrect code pattern for `free()` which was shown in Fig. 7.

7.4 Modeling External Input

Knowing which data sources not to trust is not obvious. Still, many bugs become security vulnerabilities because the user can affect data input. The solution is system and API specific. *Environment variables* are considered untrustworthy sources [29], and Ashcraft and Engler add another three categories—System calls, routines that copy data from user space, and network data [1]. In modeling security properties these sources of so called *tainted* data will all be considered as nodes of external input and analyzed via transitive data dependencies.

8. FUTURE WORK

Finding the modeling formalism is the first step toward a single tool able to check for several security properties. We are right now implementing a prototype tool called *GraphMatch* that uses dependence graphs to check security properties [9]. The prototype currently finds interprocedural input validation flaws. Apart from modeling other security properties and checking them with real-life code, we plan to investigate scalability and accuracy issues of the analysis, and also evaluate dependency graphs as a visual aid in secure programming. Empirical studies will be made to evaluate the heuristic ranking of potential vulnerabilities.

9. CONCLUSIONS

We have shown that there is a need for a generic formalism both for description of security properties and for static checking of these properties. In addition we believe that visual support is needed to effectively communicate with programmers. System dependence graphs decorated with range constraints and type conversion information can serve that purpose. Dependence graphs are well-known in the static analysis and compiler communities and are able to model the diversity of security properties, covering both safety and liveness properties of code, as shown by our examples.

10. ACKNOWLEDGMENTS

We would like to sincerely thank the previewers of this paper, especially David Byers.

11. REFERENCES

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

[2] Blexim. Basic integer overflows. Phrack Magazine 60 `http://www.phrack.org/phrack/60/p60-0x0a`, 2002.

[3] CERT Coordination Center. CERT/CC statistics 1988-2004. `http://www.cert.org/stats/cert_stats.html`, January 2005.

[4] 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, Washington DC, USA, 2002.

[5] W. Chen, B. Rudiak-Gould, and B. Schwartz. Automatic detection of implicit type cast errors in C. Paper in graduate course, `http://www.cs.berkeley.edu/~wychen/papers/261.ps`, 2002.

[6] B. V. Chess. Improving computer security using extended static checking. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, Oakland, California, USA, 2002.

[7] B. V. Chess. Personal communication, 2004.

[8] CSO magazine, U.S. Secret Service, and CERT Coordination Center. 2004 e-crime watch survey. `http://www.csoonline.com/releases/052004129_release.html`, May 2004.

[9] P. Fak. Modeling and pattern matching security properties with dependence graphs. Master's thesis, Linkopings universitet, August 2005.

[10] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[11] J. S. Foster, R. Johnson, J. Kodumal, T. Terauchi, U. Shankar, K. Talwar, D. Wagner, A. Aiken, M. Elsman, and C. Harrelson. Cqual: A tool for adding type qualifiers to C. `http://www.cs.umd.edu/~jfoster/cqual/`, 2003.

[12] Grammatech Inc. Codesurfer. `http://www.grammatech.com/products/codesurfer/`.

[13] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1), 1990.

[14] M. Howard. Reviewing code for integer manipulation vulnerabilities. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode%/html/secure04102003.asp`, April 2003.

[15] R. Johnson and D. Wagner. Checking linux kernel user-space pointer handling with cqual. Work-in-progress report at IEEE Symposium on Security and Privacy, May 2003.

[16] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[17] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium*, Washington DC, USA, August 2001.

[18] V. B. Livshits and M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of the 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Helsinki, Finland, 2003.

[19] S. S. Muchnick. *Compiler Design & Implementation.* Morgan Kaufmann, 1997.

[20] M. Musuvathi and D. Engler. Some lessons from using static analysis and software model checking for bug finding. In *Proceedings of the Second Workshop on Software Model Checking*, Boulder, Colorado, USA, 2003.

[21] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177—184, Pittsburg, Pennsylvania, 1984.

[22] T. Reps and G. Rosay. Precise interprocedural chopping. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 41–52, Washington DC, USA, 1995.

[23] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[24] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Automated detection of format-string vulnerabilities using type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, `http://www.cs.berkeley.edu/~ushankar/`, August 2001.

[25] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed System Security Symposium*, pages 3–17, Catamaran Resort Hotel, San Diego, California, February 2000.

[26] N. Walkinshaw, M. Wood, and M. Roper. The java system depencence graph. In *Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation*, Amsterdam, The Netherlands, 2003.

[27] M. Weber, V. Shah, and C. Ren. A case study in detecting software security vulnerabilities using constraint optimization. In *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation*, Florence, Italy, 2001.

[28] M. Weiser. Program slicing. In *Proceedings of the Fifth International Conference on Software Engineering*, pages 439–449, San Diego, California, USA, 1981.

[29] D. A. Wheeler. Secure programming for Linux and Unix HOWTO v3.010. `http://www.dwheeler.com/secure-programs/`, March 2003.

[30] J. Wilander and M. Kamkar. A comparative study of publicly available tools for static intrusion prevention. In *Proceedings of the 7th Nordic Workshop on Secure IT Systems*, Karlstad, Sweden, November 2002.