Model Checking Dataflow for Malicious Input

Ansgar Fehnker*
NICTA
University of New South Wales
Sydney, Australia
ansgar.fehnker@nicta.com.au

Ralf Huuck*
NICTA
University of New South Wales
Sydney, Australia
ralf.huuck@nicta.com.au

Wolf Rödiger†
Department of Software Engineering
Augsburg University
Germany
wolf-steffen.roediger@student.uni-augsburg.de

ABSTRACT

Many embedded systems today are no longer isolated control units, but are fully fledged miniature desktops with their own kernel and sometimes operating system networked with the outside world. This opens up a whole new set of security issues previously not known to embedded systems. One example is potentially malicious input that exploits source code weaknesses leading to critical mission failures. In this paper we propose a new automated malicious input detection approach that works on a staged application of traditional tainted dataflow analysis and syntactic software model checking. The advantages of this approach are that tainted data can be tracked from its source to its application point, a precise path through the source code can be computed, speed and precision can be custom-tuned by automated refinement, and the approach is flexible to deal with real-life security threats. We illustrate our approach with a number of analysis examples taken from existing open source C/C++ projects.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking, Formal methods*; D.4 [**Operating Systems**]: Security and Protection

General Terms

Security, Verification

Keywords

Security, Model Checking, Static Analysis, Command Injection

*NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

†This work was carried out while visiting the Systems Software Research Group at NICTA, Sydney.

1. INTRODUCTION

The times where the typical embedded system was an isolated unit for highly specialized purposes are gone. In fact, it becomes more and more difficult to differentiate between what used to be desktop capabilities from embedded systems features. Entertainment systems in cars, game consoles, phones, and TVs, to name a few, often contain a full blown operating system and more application code than a standard desktop computer not so many years ago.

To illustrate this by one example: A latest TV set contains an operating system (often Linux variants), comes with software applications totaling around 4 million lines of source code, has Internet connection, and can be remotely controlled through phone apps. Software is by far the largest cost factor and this comes with all the advantages and disadvantages of modern software systems.

One of the threats to these types of embedded systems is security. While a TV set is rarely used for mission critical applications, other similar products such as phones, integrated GPS solutions and remotely diagnosable embedded devices are. Oftentimes, security considerations are not build in, but build on top of existing systems. Either by access control, (para-)virtualization or simple obscuration.

However, building large interconnected embedded systems will always require exchange of data with different components such as: Third party applications for communication and interaction, new hardware components in a plug-and-play fashion or end users who input and store data or configure systems to their preferences.

Not surprisingly, command line injects of tainted data are ranked among the *Top 25 Most Dangerous Software Errors* listed by the SANS/MITRE organization[1]. With increased interoperability there is an increased risk of unchecked input exploiting security holes. This is in particular true when embedded systems interact with third parties they were originally not designed for.

There are a number of testing and verification approaches designed to catch or prevent these types of security flaws including standard dynamic testing, run-time verification and symbolic execution, penetration testing and formal verification. Most of these techniques either give a low assurance as they can only test certain parts or scenarios or are highly

[1]`http://www.sans.org/top25-software-errors/`

expensive in terms of run-time and human resources needed. More lightweight verification techniques that can be used at software development time, i.e., during the implementation rather than after it, are often summarized under the term *static source code analysis* [14]. Static analysis techniques comprise a number of methods to automatically investigate source code by building control and data dependencies, approximating the semantics, and tracking information across functions.

In this paper we propose a new static tainted data analysis that combines two different techniques: Traditional dataflow analysis and model checking. Dataflow analysis [1] is used as a program approximation to determine for every program point which tainted data can reach that point. Typically, dataflow analysis is fast and scalable, but it operates on a fixed abstraction and the results are often a) prone to significant over-approximation and b) do not provide an example trace explaining the path from the origin to the sink of a vulnerability. Model checking [6, 15] on the other hand is a technique for an exhaustive exploration of a state space with respect to certain temporal requirements. While well suited for abstraction refinements and to produce counter-examples, model checking applies to fixed sized finite models only making it applicable to investigate control flow problems, but much less so for tracking data dependencies.

In this work we combine both approaches as follows: We first run a classical dataflow analysis over a program to detect potential sources and sinks for tainted data. Next, we build an abstraction of the control flow graph annotated with the relevant information of where the tainted data is produced, consumed and changed. This second model enables us to model check for existing paths that lead from a tainted source to a vulnerable sink and return these paths to the end user. Moreover, we can apply abstraction refinement techniques to the model for checking whether the resulting path is a spurious one. Hence, we obtain a number of advantages over existing solutions:

Speed. Traditional software model checking approaches are prone to suffer from scalability limitations if used with a fine grained abstraction as needed for a precise taint analysis. Introducing a first stage that performs the dataflow analysis on its own, we are able to obtain all the relevant data for the subsequent model checking. By incorporating the dataflow results into the model checking problem we are able to have a much coarser abstraction compared to traditional software model checking, leading to runtimes that are fractions of a second for many real life files.

Traceability. One of the disadvantages of traditional dataflow is that it propagates (sets of) information without any history about the exact paths along which it has been propagated. Model checking on the other hand can produce a counter-example path when a property is violated. Our staged solution will automatically produce an example trace from a tainted source to its potential vulnerable application. This is in particular helpful in professional software development when dealing with large complex code.

Precision. Dataflow analysis typically has a fixed abstraction taking only limited program semantics into account. While it is good for speed it is less so for precision. The new combined model checking approach, however, enables us to make use of the advances in the area of abstraction refinement. In fact, we are able to automatically subject any given example trace to a refinement step where we efficiently determine the feasibility of that path in the actual program and not only its abstraction. This is valuable for avoiding false positives and keeping the overall noise level low.

Flexibility. The approach we present is configurable to deal with flexible requirements and different taint sources as well as vulnerability sinks. We will present a framework where the end user can configure and define the functions and parameters as well as value ranges which can trigger a vulnerability. Apart from known library functions that can expose a vulnerability in a certain context it is possible to mark third party API functions or parts thereof as introducing or consuming tainted data. This is particularly helpful in the context of embedded systems, where a certain vulnerability is only exposed in a certain context or combined with particular third party code.

In the present work we focused on merging static analysis and model checking for malicious input analysis. There are a number of pure static analysis approaches explicitly tracking potential vulnerabilities. This includes Livshits et al. using tainted object propagation [13], Wassermann and Su using a grammar-based solution to regular language containment [18] and Chess' implementation in the commercial Fortify tool [4]. On the model checking side large scale investigations have been performed with tools such as MOPS [17], SLAM [2] and SATABS [5]. Common to the model checking tools is that the security checks considered are not data driven, but rather control flow problems that can easily be expressed in terms of state machines. On the upside, unlike the former approaches they are able to create counter-examples and use inbuilt refinement techniques. Our approach is an alternative bridging the two worlds.

The remainder of this paper is organized as follows: In Section 2 we provide some background on dataflow analysis and model checking. In particular, we summarize our earlier approach on syntactic software model checking. In the subsequent Section 3 we present our novel framework of a staged analysis to detect tainted data and its potentially malicious use. We explain the underlying algorithms, the transformation steps from dataflow results to a model checking problem, and we highlight the advantages with a running example. Section 4 provides some more details on our experience with applying the presented framework to large scale real life code. This is followed by some conclusions and potential directions for future work in Section 5.

2. BACKGROUND

In this work we build on two techniques from the static analysis and verification community: Dataflow analysis and model checking. We first give a brief introduction to dataflow analysis for tracking potentially malicious user data in programs. Moreover, we give an introduction to *syntactic model*

checking, an approach that uses a standard model checking techniques to solve static analysis problems.

2.1 Dataflow and Taint Analysis

Dataflow analysis is a general term for a techniques of computing for each program point a set of potential information reaching that point. The information can be values of variables, variable names, expressions or other items of interest. Moreover, among other dimensions the analysis is typically distinguished to be either flow-sensitive or flow-insensitive as well as either context-sensitive or context-insensitive. This means, whether the analysis takes the control flow of the program into account and/or the calling context. For the purpose of this work we introduce a flow-sensitive, but context-insensitive analysis. We refer the reader to the future work section for directions on the latter.

The information computed for each program point is represented as a lattice L of program facts and each program point k has the ability to modify its local information based on the program point's semantic approximation. This modification is called a *transfer function* $f_k : L \to L$. In a flow-sensitive setting the information from each program point will be propagated along the *control flow graph (CFG)* of the program. We represent the CFG as a directed graph $G = (N, E)$ where N is a finite set of nodes representing the program points and $E \subseteq N \times N$ is the transition relation between these nodes. Note, the CFG is an abstraction of the original control flow as all branches are non-deterministically interpreted and there is no additional information on conditions or other program constraints. We will address these in later parts of this work.

To propagate information along the paths of the CFG we define two sets for each CFG node $k \in N$: IN_k and OUT_k which represent the information going into node k and leaving node k. Moreover, we define the flow of information between the nodes as follows:

$$\mathrm{IN}_k = \bigcup_{i \in pred(k)} \mathrm{OUT}_i \tag{1}$$

This means, input information of node k is the union of the information available at the exit of its predecessors pred(k). Note, in the general case merging information is called the *meet* operator and is not required to be the set union, but could be for instance the intersection depending on whether the analysis is to achieve an over- or under-approximation of the program's actual information.

Finally, the effect that each node has on facts of information is described by their local transfer function that are represented by the sets GEN_k and KILL_k. Here, GEN represents the set of new information that is *generated,* while KILL represents the set of old information that gets deleted. As a result the exiting information at each node k is described as:

$$\mathrm{OUT}_k = \mathrm{GEN}_k \cup (\mathrm{IN}_k \setminus \mathrm{KILL}_k) \tag{2}$$

This means, the output is the input information without the information getting deleted at node k, but added to it is the newly generated information. For every node k there will be one equation describing IN and one describing OUT. The overall equation set is solved by computing the least fixed point solution to it. Since we have a finite set of equations, our lattice will be of finite height, and all transfer functions are monotonous, such a fixed point is guaranteed to exist and can be computed algorithmically. In this paper we use a standard worklist algorithm to do so.

For applying the above to track potentially malicious input, we identify program points where unchecked user data can enter to program. We mark variables containing this data as *tainted.* The transfer functions for nodes where tainted data is entered or propagated will generate a set of tainted variable names. Fresh assignments to tainted variables "break" the propagation and will be represented by transfer functions that kill the respective variable names from the tracked set. This means, the lattice of information will be based on the variable names at each location, which are potentially tainted. Solving the dataflow equations propagates to each node the set of tainted variables. A security issue is then identified by finding a function that is potentially vulnerable to unchecked user input and uses the computed tainted data.

Finally, we need to stress that the standard dataflow analysis has a number of limitations: First of all, there can be some significant over-approximation every time the union operator is applied to merge paths. Moreover, standard dataflow analysis does not provide an example trace about how a violation as above has occurred. It simply states there is one at a specific location. Most importantly, there is no good framework for refinement for dataflow analysis. One can move to a more detailed lattice of facts requiring a completely new set of transfer functions or modifying the criteria for applying the union operator, but it does not match the advances in the recent years made in the area of Boolean abstractions and counter-example guided abstractions. Much of these have been addressed in the complementary area of model checking we are describing next.

2.2 Syntactic Model Checking

Model checking is an automated verification technique to analyze large models represented as labeled graphs for temporal relationships between these annotations. More formally, we check whether a model M satisfies a property ϕ denoted as $M \models \phi$.

The model is often given as a *Kripke* structure, which can be expressed as $M = (N, E, \mu)$ where N is a set of nodes, $E \subseteq N \times N$ a transition relation on these nodes and $\mu : N \to 2^{AP}$ is a labeling function mapping nodes to sets of atomic propositions AP. The property ϕ to check on the Kripke structure can be expressed in various forms of (temporal) logic. For the purpose of this work we focus on *Computation Tree Logic* (CTL) [3] that allows to reason about the occurrences of the atomic propositions as well as the branching behavior of the structure.

We briefly summarize: CTL allows path quantifiers **A** and **E**, and the temporal operators **G**, **F**, **X**, and **U**. The (state)

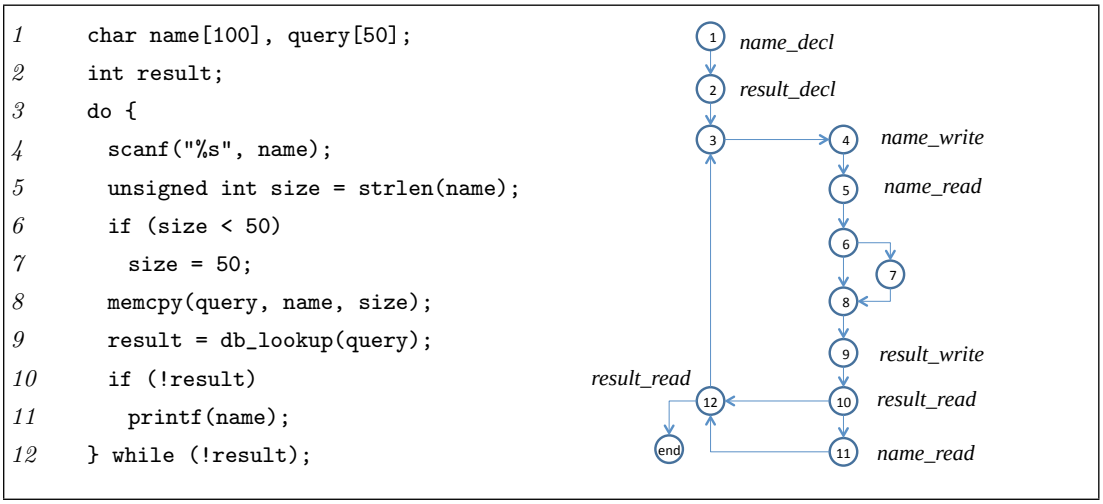


Figure 1: Example program and labeled CFG for uninitialised variable check.

formula $\mathbf{A}\phi$ means that ϕ has to hold on all paths, while $\mathbf{E}\phi$ means that ϕ has to hold on some path. The (path) formulae $\mathbf{G}\phi, \mathbf{F}\phi$ and $\mathbf{X}\phi$ mean that ϕ holds globally in all states, in some state, or in the next state of a path, respectively. The *until* $\phi\mathbf{U}\psi$ means that until a state occurs along the path that satisfies ψ, property ϕ has to hold. The *weak until* $\phi\mathbf{W}\psi$ means that ϕ is satisfied until ψ is satisfied. The latter includes that ψ never holds, in which case ϕ holds forever. In CTL a temporal operator is always immediately preceded by a path quantifier.

Model checking is used for various verification tasks including hardware, high-level system description, and software. In this work we use model checking to solve static program analysis problems. In contrast to typical equation solving approaches to static analysis we use an automata based approach [12, 8, 16]. While the details of our approach are described in [9] the basic idea is to map a C/C++ program to its corresponding control flow graph (CFG), and to label the CFG with occurrences of syntactic constructs of interest. The CFG together with the labels can be seen as a Kripke structure that can easily be mapped to the input language of a common model checking tool.

A simple example of this approach is shown in Fig. 1. The program reads user input until it finds a successful database lookup. To check, e.g., whether variables are written before they are used, we identify syntactically locations in the program that declare variables (and are not initializations at the same time), locations that write variables, and locations that read variables. For instance, for the variable `result` of Fig. 1 (a) we automatically label corresponding nodes with `result_decl`, `result_write` and `result_read`, as shown in Fig. 1 (b). These labels are computed based on a library of predefined patterns. The patterns themselves are expressed in a tree query language and are evaluated at compile time based on the *abstract syntax tree* (AST) representation of the parsed code. Similarly, we label the occurrences of variable `name`.

To check whether after a declaration of `result` the variable is written before it is read is expressed in CTL as:

$$AG\ (\texttt{result\_decl} \Rightarrow A(\neg\texttt{result\_read}\ W\ \texttt{result\_write}))$$

where $AG\phi$ stands for "for all paths and in all states ϕ holds" and $A\phi W\psi$ for "ψ holds until ϕ holds". A counter-example is a path that declares variable `result` and reads it before writing. The same property can be defined for variable `name`. In the example both variables satisfy the specification, i.e., it is guaranteed that they are not used uninitialized.

The approach to combine CTL expressions with patterns on the syntax has the advantage that the models become very compact. Both the CTL language, and the pattern language are very expressive, and can be used to encode many properties. Moreover, the model checking itself has the ability to automatically generate a counter-example trace if a property is not satisfied. This is quite useful to explain to the end user, why a check fails.

However, a limitation of the model checking approach is that the syntactic patterns are often insufficient to reason about data. This includes aliasing, buffer sizes, and information about tainted input. A common approach would be to encode these properties as part of the finite state model, i.e. to introduce state variables that model data, memory, buffers, or whether data is tainted or not. A disadvantage of these rich semantic models is that the state space easily becomes prohibitively large. With the approach as described above the state space grows in practice about linearly with the number of lines. Introducing a richer semantic model would negate this advantage.

Dataflow techniques are efficient means to get a good approximation of the program state. In this work we use these techniques to generate small models, suitable for model checking. This adds the capability to reason about paths to dataflow techniques. In addition it makes it possible to subject these paths to a closer analysis with a richer semantic finite state model only once a potential vulnerability has been detected.

3. A MULTI-STAGED TAINT ANALYSIS

We call input data *tainted* if it is provided by the end user or an unknown third party software. The goal is to check if the tainted data can propagate through a program such that it reaches a function or statement that is vulnerable to it. These types of vulnerabilities are the most common security problems such as command injections, buffer overflows, format string vulnerabilities, and array out of bounds accesses.

In this section we present our multi-staged approach by first defining and running dataflow taint analysis and subsequently using that result for syntactic model checking to obtain example traces and reduce potential false positives. We start of with a representative example program we use throughout this paper.

3.1 Running Example

We use the slightly contrived program depicted in Fig. 1 to demonstrate our approach. This program uses `scanf` in line 4 to ask the user for a string which is placed into the 100 byte sized `name` array. Next, the actual length of the input string is calculated and checked to be of a valid size. Apparently, the programmer of our example did a simple typing mistake and used the wrong comparison operator in line 6. He intended to ensure that `size` is between 0 and 50 but instead enforces that `size` is at least 50—we will come back to this later. In line 8 `memcpy` copies `size` bytes of the input string from `name` to the global char array `query`. Afterwards an embedded database is queried with the function `db_lookup`. If this returns no result, `name` is printed and the user is again asked for input until finally the query is successful.

This program has two flaws that in combination are the cause for severe vulnerabilities. Firstly, the user provides the value of the variable `name`. Since `size` is derived from `name` the user's control extends also to its value. Secondly, these user controlled variables are used as arguments to the vulnerable functions `memcpy` and `printf`.

This combination of user input and unchecked use leads to the following vulnerabilities:

- A format string bug, which an attacker can exploit by providing dangerous format tokens in the input string. This enables the attacker to read memory values in line 11 and even execute their own code with the privileges of the vulnerable program.
- A buffer overflow in line 8 if the user input is longer than the size of the `query` buffer. The overflow will overwrite the adjacent memory and is described in more detail in Section 3.3.2.

This specific program is a paradigm for a larger class of problems, namely those programs where user or third party data is entered and the data will be used without proper checking in later parts of the software. In the following we will present the details of our multi-level approach to deal with these types of issues. The approach has several stages grouped into two main analysis techniques:

1. Dataflow Analysis
   (a) Finding tainted sources
   (b) Propagating taints
   (c) Locating tainted sinks
2. Model Checking
   (a) Generating the model using results from 1.
   (b) Defining vulnerabilities as CTL properties
   (c) Presenting counter-examples

In the following we will present each of the steps in detail.

3.2 Dataflow Analysis

The first stage of our analysis determines where user input is entered, where it can potentially flow to in the program and which vulnerable statements might potentially be reached. This requires three steps: finding user input, propagating this tainted input along the control flow and locating vulnerable functions. Note, that these steps alone cannot return a path from the input source to the vulnerability sink. This will be addressed in Section 3.3.

3.2.1 Finding Tainted Sources

Program functions that are known to return user input or known to potentially return user input are called *tainted sources*. We identified around 50 of those functions from the standard C library. This, however, can be extended by the end user to third party functions by editing a configuration list. Once we know which functions return potentially malicious data, we can identify and track the variables that are potentially under attacker control.

As an example for a tainted source and how we represent it in our configuration list, consider the `scanf` function used in Fig. 1, which reads user input from `stdin`:

```
4 scanf("%s", name);
```

The function `scanf` parses the input into tokens according to the format specifiers contained in the first parameter. The resulting tokens are stored into the locations provided by the additional parameters. In our example the format string contains only the specifier "%s" which causes `scanf` to read subsequent characters until it encounters a whitespace. The resulting character array is copied to the location of `name`.

After a call to `scanf` all parameters starting with the second as well as its return value are tainted. This behavior is represented in our configuration list of user input functions by storing the name of the function, the function parameters that get tainted, and whether the return value is tainted:

$$\begin{aligned} scanf &\in \text{InputFunctions} \\ \text{taintedParameters}(scanf) &= \{2, \ldots, n\} \\ \text{returnTainted}(scanf) &= \text{true} \end{aligned}$$

In a connected embedded context the end users can modify the configuration list and add their own functions.

3.2.2 Propagating Taints

At this point of the analysis we know which variables are directly controlled by the user. However, this is insufficient for any non-trivial security analysis which can easily be observed in our running example:

```
4 scanf("%s", name);
5 unsigned int size = strlen(name);
```

The variable `size` depends on the value of `name` which is supplied by the user. As a result the user has also control over `size`. This simple example demonstrates the need of a dataflow analysis, which propagates the taint information through assignments.

We use a classical dataflow analysis to determine which variables may be influenced by user input. The information we propagate are the sources of user input. Tainted sources are represented by the name of the variable and the line number where the user input originally came from. We use this information later to generate a path starting at that input function and ending at a vulnerable function. In our example program user input is assigned to the variable `name` in line 4. This tainted source is therefore represented as (*name*, 4). The variable `name` is subsequently used in an assignment to `size`. The generated taint information is represented as (*size*, 4) because the user input originates from the `scanf` function in line 4.

Defining the Flow Equations. As described in Section 2.1 the dataflow equation system is set up by defining for each node k the sets of incoming information, equation (1), and the set of outgoing information, equation (2). This requires in particular to define for each node k the rules when to generate new facts (GEN_k) and delete old facts (KILL_k).

Taint information is generated when a variable is either tainted as defined in the previous section or when a tainted variable is used in an assignment, i.e., the tainting property is passed on as we saw in the example. This means:

$$\text{GEN}_k = \text{GEN}_k^{input} \cup \text{GEN}_k^{assign}$$

A variable is tainted by user input functions if the variable is either assigned to the return value of that function or the variable is used as an output parameter:

$$\begin{aligned}
&\text{GEN}_k^{input}([x \coloneqq f(x_1, \ldots, x_n)]) = \\
&\qquad \{(x, k) \mid f \in \text{InputFunctions} \\
&\qquad \wedge\, \text{returnTainted}(f)\} \\
&\text{GEN}_k^{input}([f(x_1, \ldots, x_i, \ldots, x_n)]) = \\
&\qquad \{(x_i, k) \mid f \in \text{InputFunctions} \\
&\qquad \wedge\, i \in \text{taintedParameters}(f)\} \\
&\text{GEN}_k^{input}(\text{other stmt}) = \{\}
\end{aligned}$$

Moreover, we define that an assignment taints a variable if there exists already a tainted variable in the righ-hand side of that assignment. Using the notion $use(t)$ for denoting all variables occurring in the expression t we define:

$$\begin{aligned}
&\text{GEN}_k^{assign}([x \coloneqq t]) = \\
&\qquad \left\{(x, k') \mid \exists x' \in use(t) \wedge (x', k') \in \text{IN}_k\right\} \\
&\text{GEN}_k^{assign}(\text{other stmt}) = \{\}
\end{aligned}$$

Having covered all the cases we consider as taint generating, we next define those cases that can delete the tainted status. We define a variable is "killed" if it gets newly assigned to. Other statements do not kill taint information. Note, however, that by definition of equation (2) we later add the newly generated information, if in addition the function taints variables. This means, if a variable gets killed it might be added again depending on the status of the variables on the right-hand side. We define kills as:

$$\begin{aligned}
&\text{KILL}_k([x \coloneqq t]) = \left\{(x, k') \mid \forall k' \in N\right\} \\
&\text{KILL}_k(\text{other stmt}) = \{\}
\end{aligned}$$

The set of flow equations over all nodes is iteratively solved until the least fixed point reached. Since our lattice of facts is the product of a finite number of locations and variable names and since all our flow functions are monotonic, it is guaranteed that we will reach a fixed point. As mentioned earlier we use a simple worklist algorithm in our implementation to solve those equations. The resulting fixed point for our example is as follows:

k	IN_k	OUT_k
1	—	—
2	—	—
3	—	—
4	(name, 4), (size, 4)	(name, 4), (size, 4)
5	(name, 4), (size, 4)	(name, 4), (size, 4)
6	(name, 4), (size, 4)	(name, 4), (size, 4)
7	(name, 4), (size, 4)	(name, 4)
8	(name, 4), (size, 4)	(name, 4), (size, 4)
...	...	...
12	(name, 4), (size, 4)	(name, 4), (size, 4)

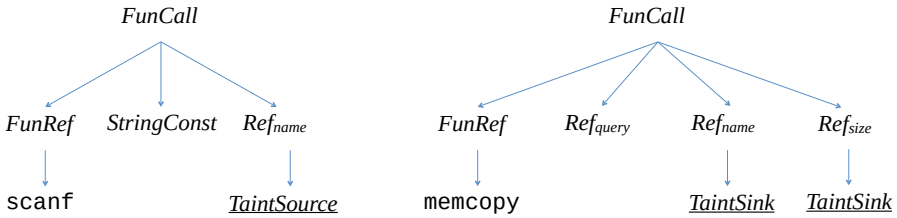
Both `name` and `size` are tainted by input originating from line 4. `size` is killed by an assignment in line 7.

3.2.3 Locating Tainted Sinks

The next step is to identify if any user input is used as a parameter to a vulnerable function.

Similar to the list of user input functions we also maintain a collection of vulnerable functions. This includes the `printf` family of functions, memory copy and allocation functions, string manipulation functions as well as the array access operator. Each entry specifies a *tainted sink*. At the moment we consider about 40 different functions.

The configuration list contains the name and vulnerable pa-



(a) Tainted source annotation (b) Tainted sink annotation

Figure 2: Annotations to the abstract syntax tree.

rameters for each function. The `memcpy(dest, src, size)` function which copies a number of bytes from a source to a destination buffer is represented as follows:

$$\begin{aligned} memcpy &\in \text{VulnerableFunctions} \\ \text{vulnerableParameters}(memcpy) &= \{3\} \\ \text{rangeParameter}(memcpy) &= 1 \end{aligned}$$

This means, the function named `memcpy` is subject to a vulnerability when user input is passed as the third parameter and not checked to be in the bounds of the first parameter.

Handing Over to Model Checking. In this first stage we computed the potential sources, sinks, generating and killing locations. We will use some of this information in the subsequent model checking process to compute and validate the actual paths through the program. To do so, we annotate the parse tree of the program with the tainted sources, tainted sinks and killing taint locations. We shall see in an subsequent section why this is sufficient. With respect to the running example, the annotations of the abstract syntax tree for the occurrence of `scanf` is depicted in Fig. 2a and for `memcpy` in Fig. 2b. We see both of them are function calls with different parameters, where some of the parameters a marked as tainted sources and sinks. In a similar fashion other statements are marked as kills.

3.3 Model Checking for Tainted Data

Our dataflow analysis computes an over-approximation of the statements that can be affected by user input. To reduce the number of false positives and provide the user with a feasible path we use model checking. We want to find valid paths leading from a *tainted source* to a *tainted sink* without encountering a *taint kill* on the way. Such a path represents an exploitable vulnerability. The resulting trace through the source code is presented to the user.

3.3.1 Generating the Model

In the previous analysis step the AST has been annotated with tainted sources, kills and sinks. An implicit property of the dataflow analysis is that between every source and kill relating to the same variable all other nodes in the CFG are also tainted for that variable. This in particular means, if we can find a path from a source to a sink without any corresponding kill in between, this will be a path along which all nodes are tainted. We consider this as a *valid* path.

Translating the existing information into a Kripke structure for model checking, i.e., for finding a path as above, is straightforward: We directly translate the CFG into the transition structure of the Kripke model and we label the CFG nodes with their corresponding tainting labels from the AST. The resulting labeled CFG of the example program is shown in Fig. 3. We see that line 4 is a tainted source, line 8 and line 11 are sinks and than the assignment in line 7 kills the taint for `size`.

3.3.2 Defining Vulnerabilities as CTL Properties

As mentioned above, we like to find a path from a source to a sink without an intermediate kill. Moreover, we like the model checker to generate us the path as a counter-example. This means, we check for the negation of the above, i.e., that there is no path from a source to a matching sink without some intermediate kill. Should this be violated, the model checker will automatically generate the path we originally were looking for. More formally, this is expressed as:

$$AG\ (\text{taint\_source} \Rightarrow AX\ (A\ (\neg\ \text{taint\_sink})\ W\ \text{taint\_kill}))$$

This means, on all paths every node is either not labeled with *taint_source* or for all following nodes holds that there is never a node labeled *taint_sink* before a node labeled *taint_kill*. Of course, the actual CTL formula is slightly more complex since the *taint_kill* has to correspond to the variable names reaching the *taint_sink* and also the *taint_source* has to match the *taint_sink*. As a result we will have one CTL formula for every occurrence of a sink.

3.3.3 Presenting Counter-Examples

When the model checker determines that the CTL formula is not valid for the given Kripke structure it will report a counter-example. Because of the high level of abstraction where branching is interpreted as non-deterministic choice, this path might still be spurious. We address this in the next section.

As for the example program two paths are reported to the user: The first path starts at the `scanf` function in line 4 and ends with the use of `name` within the `printf` function in line 11. The second path also starts at the `scanf` function, takes the false branch—by this skipping line 7—and ends with the use of `size` as the third argument for the `memcpy` function in line 8. The output generated by our tool is as follows:

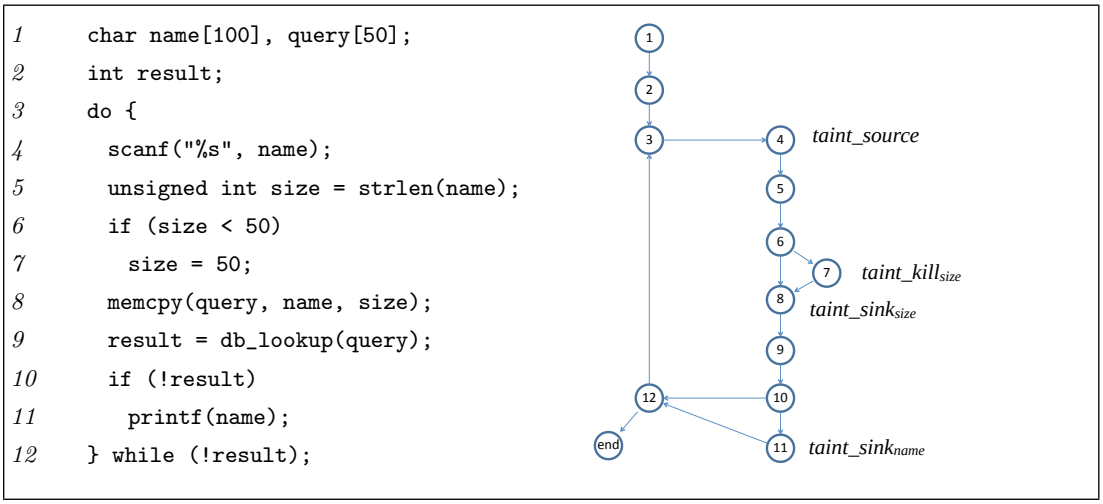


Figure 3: Example program and labeled CFG for the taint analysis.

```
wess11.c:7: warning: User controlled
    variable 'size' used as parameter
1:    main -
2:    main -
3:    main -
4: *  main - reference to variable 'name'
5:    main -
6: *  main - take the False branch
8: *  main - reference to variable 'size'
```

Both paths highlight severe vulnerabilities that enable an attacker to compromise the system. The first issue is a format string bug that can be exploited by providing dangerous format tokens to the input string. This includes "%x", which reads the next four bytes from the stack and "%n", which writes four bytes to an arbitrary memory location. The attacker can use these two format tokens to read memory contents or even execute his own code with the privileges of the vulnerable program.

The other vulnerability is a potential buffer overflow. If the user input in line 4 is longer than the size of the `query` buffer it will overflow and `memcpy` will overwrites the adjacent memory. Buffer overflows have a long history and are ranked third most dangerous in the current CWE/SANS list.

3.4 Improvements

We used several improvements to reduce the number of false positives and to increase the precision of our analysis. One is an additional interval abstract interpretation that is performed before the other analysis steps, and the other improvement an integration into an existing abstraction refinement framework. We will briefly outline both improvements.

3.4.1 Value Range Validation

As a first step in our improved analysis we run an interval abstraction interpretation [7]. As a result we get potential value ranges for all integer variables at every relevant node in the AST. We make use of this information to see if certain vulnerabilities are in fact false positives, because the developer appropriately checked the tainted data before using it.

The developer of our original example simply used the wrong comparison operator when checking `size` in line 6. Consider the following modification to our example:

```
6 if (size > 50)
7   size = 50;
```

By correcting the original mistake `size` is always less than or equal to 50. Our interval abstract interpretation will be able to also derive this fact. Hence, while the user still has control over the actual value of `size`, there is nonetheless no longer a security issue in the `memcopy` call because `size` is checked to be smaller than the size of `query`. A buffer overflow is successfully prevented by user input validation.

3.4.2 Abstraction Refinement

Classical dataflow analysis works with a fixed model of abstraction defined by the transfer functions onto the abstract domain. Refinement as such can be partially achieved by refining the abstract domain, but it does not change the underlying model of flow equations between nodes of the control flow graph. There is no refinement of the CFG incorporating additional program semantics including information about loop bounds and conditionals. On the other hand, the model checking community has long worked on program refinement to tune the trade-off between a precise model and a speedy analysis.

In particular, we like to refine our analysis in such a way that we can automatically identify spurious vulnerabilities resulting from unfeasible code paths. Right now, model checking on the level of the CFG abstraction can create counter-examples including such infeasible paths. A standard way to add more precision is *counter-example guided abstraction refinement* (CEGAR), as used in [11, 5]. In earlier work [10] we developed a different approach computing a *precise least solution* of an interval equation system, which

is computationally faster, at the expense of some precision. The main idea is to subject counter-examples to an interval abstract interpretation and check for the feasibility of that path. If the path is infeasible the model is refined with observer automata reflecting the minimal cause for it. The analysis is re-run until a bug disappears or no more infeasible counter-example occurs. We implemented the approach in our Goanna tool we use for the experiments in Section 4.

Independent of the exact abstraction refinement approach used, program refinement works very well with model checking and is something that is not straightforward to achieve with the classical dataflow framework.

4. EXAMPLES

We implemented the approach in our industrial strength static analysis tool for C/C++ called *Goanna*[2]. The tool itself is a closed source project, but available for commercial as well as academic use. The core technology is centered around syntactic model checking, abstract interpretation and abstraction refinement. Some details can be found in [9].

For experimenting with our proposed security analysis we investigated several open source C/C++ projects. Some metrics are depicted in Fig. 4. The runtimes are typically 3 to 4 times slower than compilation, but this includes the abstract interpretation process, the dataflow analysis and the model checking phase. For the purpose of this work we choose three interesting issues to show the validity of our approach.

4.1 wu-ftpd 2.6.0

Our analysis reports a use of a tainted variable in a memory allocation inside the `fb_realpath()` function, which resides in the `realpath.c` file.

```
158 loop:
186 p = resolved;
210 size_t len = strlen(p);
211 char *tmp = calloc(len + 1,
    sizeof(char));
219 n = readlink(p, resolved, MAXPATHLEN);
235 goto loop;
```

In line 219 the content of the symbolic link referred to by `p` is stored into `resolved`. The content and length of symbolic links could be controlled by the user. In the next loop iteration, `resolved` is used in an assignment to `p` in line 186. `len` is then derived from the size of `p` in line 210 and subsequently used as the size for an allocation. If an integer overflow occurs in the allocation on line 211 an attacker can cause a buffer overflow possibly compromising the system.

It is not likely that this issue presents an exploitable vulnerability, because the size of `len` will not be larger than `MAXPATHLEN`. Still, it exhibits a complicated source code structure, which depends on gotos and where taint follows complicated paths. Without counter-examples the cause would be difficult to track down.

[2] `http://www.redlizards.com`

Program	Source	Reach	Kill	Sink	Issues
bacnet-stack 0.5.9	108	754	12	0	0
git 1.7.6-rc1	90	803	36	16	2
mongoose 3.0	143	413	10	6	0
NanoStack 1.1.0	24	375	6	2	0
nfs-utils 0.1.6	59	600	18	7	0
redis 2.9.0	59	589	8	11	2
sqlite 3.7.7.1	5	134	6	13	2
sendmail 8.14.5	35	533	27	30	2
wu-ftpd 2.6.0	130	1146	27	18	2

Figure 4: Analysis results for open source projects.

4.2 redis 2.9.0

In the `rdb.c` source file inside the `rdbLoadDoubleValue()` function a tainted variable is used to access an array.

```
660 char buf[128];
661 unsigned char len;
663 if (fread(&len,1,1,fp) == 0)
      return -1;
664 switch(len) {
665 case 255: *val = R_NegInf; return 0;
666 case 254: *val = R_PosInf; return 0;
667 case 253: *val = R_Nan; return 0;
668 default:
669   if (fread(buf,len,1,fp) == 0)
        return -1;
670   buf[len] = '\0';
671   sscanf(buf, "%lg", val);
672   return 0;
673 }
```

In line 663 a byte is read from a file and stored in `len`. In line 669 `len` number of bytes are read and stored in `buf`. `len` is subsequently used to access `buf` in line 670.

If `len` is between 129 and 252 a buffer overflow occurs in line 669 and the array access in line 670 would be out of bounds. Depending on the other properties of this program this could exhibit a vulnerability.

4.3 sqlite 3.7.7.1

In the `shell.c` source code file inside the `find_home_dir()` function a tainted variable is used as the size for a memory allocation.

```
2531 if (!home_dir) {
2532   home_dir = getenv("HOME");
2533 }
2554 if( home_dir ){
2555   int n = strlen30(home_dir) + 1;
2556   char *z = malloc( n );
2559 }
```

In line 2532 the contents of the `HOME` environment variable are stored into `home_dir`. In line 2555 `len` is derived from `home_dir`. Afterwards `n` is used as the number of bytes allocated with the `malloc` function. An attacker could possibly choose the length of the `HOME` variable so that an integer overflow occurs in line 2555 and malloc allocates 0 bytes. This leads to a buffer overflow when the program writes to z and believes to have allocated enough memory.

However, this warning is a false positive. The non-library function `strlen30` ensures that its result will fit into 30 bits. Our analysis does not know about this specific property of `strlen30`. Still the warning has the benefit that the programmer thinks about the potential problem. Afterwards he can easily mute this warning.

5. CONCLUSIONS

Summary. In this work we presented a new stage dataflow and model checking approach for malicious code detection. Unlike each individual approach we obtain a solution that is fast, precise, traceable and flexible. Dataflow analysis provides efficient means to detect and track tainted data and is much better suited than model checking to deal with data dependent properties. However, unlike standard dataflow analysis our staged solution is able to generate counterexample traces indicating the flow of information leading to a security vulnerability. Moreover, by incorporating model checking techniques we are able to make use of existing abstraction refinement techniques enhancing the precision of the analysis in an efficient manner.

Future Work. There are a number of directions to extend the current work. Our main goal is to extend the current analysis, which is intra-procedural to an inter-procedural whole program analysis. The reasons are clear: Most sophisticated security bugs span several functions and are often due to the fact that several developers implement different parts of the code base without a unified security view. The current version of Goanna already supports a summary-based inter-procedural analysis for important facts such as potential null pointers, ranges of variables and allocated memory. We expect it to be a straightforward process to include the staged taint analysis into the same framework.

Another area of interest is to include a flow-sensitive alias analysis into the proposed framework. Capturing pointer alias information as part of the analysis would help to detect more sophisticated security issues and would also add precision to the current checks. One of the challenges with a precise alias analysis is to keep the performance overhead low. Again, we envision a combined staged solution similar to the proposed method to obtain the best of both worlds.

6. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *In: IFM. (2004*, pages 1–20. Springer, 2004.

[3] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. In *POPL '81*, pages 164–176. ACM, 1981.

[4] B. Chess and J. West. *Secure programming with static analysis*. Addison-Wesley Professional, 2007.

[5] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Proc. TACAS 2005*, LNCS 3440, pages 570–574, 2005.

[6] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons for branching time temporal logic. In *Logics of Programs Workshop, New York, May 1981*, volume 131 of *LNCS*, pages 52–71. Springer Verlag, 1982.

[7] P. Cousot. Semantic foundations of program analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., New Jersey, 1981.

[8] D. Dams and K. Namjoshi. Orion: High-precision methods for static error analysis of C and C++ programs. Bell Labs Tech. Mem. ITD-04-45263Z, Lucent Technologies, 2004.

[9] A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch. Model checking software at compile time. In *Proc. TASE 2007*. IEEE Computer Society, 2007.

[10] A. Fehnker, R. Huuck, and S. Seefried. Counterexample guided path reduction for static program analysis. In *Concurrency, Compositionality, and Correctness*, volume 5930 of *LNCS*, pages 322–341. Springer, 2010.

[11] T. Henzinger, R. Jhala, R. Majumdar, and G. SUTRE. Software verification with BLAST. In *Proc. SPIN2003*, LNCS 2648, pages 235–239, 2003.

[12] G. Holzmann. Static source code checking for user-defined properties. In *Proc. IDPT 2002*, Pasadena, CA, USA, June 2002.

[13] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[14] F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.

[15] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. Intl. Symposium on Programming, April 6–8*, pages 337–350. Springer Verlag, 1982.

[16] D. A. Schmidt and B. Steffen. Program analysis as model checking of abstract interpretations. In *Proc. SAS '98*, pages 351–380. Springer-Verlag, 1998.

[17] B. Schwarz, H. Chen, D. Wagner, J. Lin, W. Tu, G. Morrison, and J. West. Model checking an entire linux distribution for security violations. In *ACSAC '05*, pages 13–22. IEEE Computer Society.

[18] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. *SIGPLAN Not.*, 42:32–41, June 2007.