Augmenting Vulnerability Analysis of Binary Code

Sean Heelan
Persistence Labs
sean@persistencelabs.com [*]

Agustin Gianni
agustin.gianni@gmail.com [*]

ABSTRACT

Discovering and understanding security vulnerabilities in complex, binary code can be a difficult and time consuming problem. While there has been notable progress in the development of automatic solutions for vulnerability detection, manual analysis remains a necessary component of any binary auditing task. In this paper we present an approach based on run time data tracking that works to narrow down the attack surface of an application and prioritize code regions for manual analysis. By supporting arbitrary data sources and sinks we can track the spread of direct and indirect attacker influence throughout a program. Alerts are generated once this influence reaches potentially sensitive code and the results are post-processed, prioritized, and integrated into common reverse engineering tools. The data recorded is used to inform the decisions of users, rather than replace them. By avoiding the processing required for semantic analysis and automated reasoning our approach is sufficiently fast to integrate into the normal work flow of manual vulnerability detection.

1. MOTIVATION

Much of the commodity, commercial and industrial software in use today is only available in binary form. As a result, analysis of this software requires reverse engineering — a process that is primarily manual and usually labor and time intensive. Reverse engineering is a necessary step in vulnerability detection, malware analysis, crash triaging, exploit development, protocol recovery, interoperability engineering and other processes. Although the end goal differs, these tasks typically involve solving similar problems during the process of understanding a program in binary form. Due to the time and labor cost of reverse engineering, algorithms and techniques for assisting in the process are valuable.

In this paper we will focus on the goal of vulnerability detection, an area that has seen significant research interest over the past few years in response to the need for greater machine assistance. Progress has been made in the testing of file parsers [11, 12, 19], and system tools [7, 6] through techniques based on symbolic and concolic execution. However, many vulnerability types, software architectures and programs above a certain size are currently not handled by automatic methods. As such, a reverse engineer is required to manually assess the software. When one considers the task of assisting the reverse engineer in this process many new opportunities and research problems arise.

[*]Parts of this work were completed while the author was an employee of Immunity Inc.

2. INTRODUCTION

Over the past decade there has been extensive work in developing tools and theory based on formal program analysis with the goal of analyzing binary code. Among the most successful outcomes of this research have been those approaches based on symbolic/concolic execution. Combining binary instrumentation, software emulators and SMT solvers, automated solutions have been proposed for bug finding [11, 12], exploit generation [4, 1, 13], protocol reverse engineering [5], driver reconstruction [8], type recovery [21] and a variety of other tasks. These efforts have also resulted in the availability of several frameworks for binary analysis, including TEMU [22], BAP [3] and S2E [9].

The problems these tools tackle are faced daily by professionals involved in reverse engineering, vulnerability detection and exploit development. However, in general the solutions presented have not permeated into this industry. One possible reason is that the tools often do not integrate well with the work flow of the reverse engineer. Reverse engineering is generally a process of iterative, gradual understanding through forming hypotheses and looking for supporting evidence. Tools that have excessively long running times or do not support bidirectional information flow between the user and the tool are incompatible with such a process.

For example, frameworks that rely on full semantic emulation of instructions tend to have running times measured in hours for complex software like web browsers, document viewers and network servers. While such information may be necessary if one wants to automatically determine whether a program trace potentially contains a bug, we will later show how a more lightweight approach can provide a user with the required information to make this determination in a much smaller amount of time. As a result, more code can be covered while at the same time gaining an insight into the program that is not available if one relies on entirely automatic methods.

Another problem for many tools is that they are designed to be entirely automatic solutions and as a result often lack the flexibility to deal with corner cases not envisaged by their designers. While reverse engineering involves many repeated patterns of work there are inevitably corner cases and complications in the analysis of every application.

Automatic approaches to bug finding also tend to be quite limited in the types of properties they can reason about, and as a result the types of bugs they can find. Tools such as KLEE [6] and EXE [7] can easily model and reason about arithmetic constraints and thus are targeted at bugs directly resulting from integer overflows, underflows and incorrect bounds. On the other hand they know nothing about dangling pointers, race conditions, type confusion and other common bug classes.

While full automation is desirable in many situations, in this paper we discuss an approach that is user centric. Instead of aiming to replace a user, we present a combination of dynamic taint analysis and lightweight static analysis to inform their decisions and increase efficiency during manual program analysis. Our algorithms focus on data gathering, analysis and display while leaving the specifics of problem solving up to the user.

We will direct the system towards the problem of manual vulnerability discovery but the information generated can also be used to seed more automated systems. A common problem faced by automated systems is in deciding what parts of the program to explore and how to explore them. The output we generate can be used to guide these decisions and target testing tools towards more interesting parts of the attack surface. As an example, similar technology has been successfully applied to provide guidance during fuzzing [10].

In section 3 we describe a system with support for direct and indirect data tainting. The system supports arbitrary data origins and sinks and can generate alerts based on function and instruction level information. These alerts focus the user on potentially sensitive areas of the code that are influenced by attacker controlled input. The primary goal of this tool is to assist in attack surface discovery and to prioritize regions of the code for assessment. Later in section 4 an evaluation of this tool on several real world applications.

Benefits of this approach include:

Generality Dynamic data-flow tracking is a well studied area [20] and usually works regardless of the software being analyzed. Instead of attempting to perform automated reasoning on top of this we instead focus on prioritizing the results of this analysis for human assessment. This allows the user to target the tool at software not generally handled by current automatic approaches such as network servers, web browsers and interpreters. It also assists the user when looking for bug classes that are not well handled by approaches based on symbolic execution e.g. use-after-free bugs.

Efficiency A key component of our approach is post-processing of the recorded data to allow the user to find the events they may be most interested in quickly. We take two approaches to this. The first is to use lightweight static analysis algorithms and a ranking function to prioritize particular results. The second is to extract relevant properties from recorded events and present them to the user in order to inform their decision making.

Speed We perform minimal analysis at run-time and so the overhead on the instrumented program is low. The time taken for post-processing of data is usually a few seconds. This is important because it allows the user to iteratively run the tool, view results, modify the configuration and re-run the tool without excessive wait times.

```
struct TaintInformation {
    size_t source_pc;
}

struct DirectTI : public TaintInformation {
    Descriptor      d;
    DescriptorInfo  di;
    Offset          o;
}

struct IndirectTI : public TaintInformation {
    vector <TaintInformation*> parents;
}

struct CompoundTI : public TaintInformation {
    vector<TaintInformation*> components;
}

struct TraceTI : public TaintInformation {
    BigInt sequence_num;
    vector<TaintInformation*> parents;
}

struct FlagUpdateTI : public TaintInformation {
    vector<TaintInformation*> parents;
}
```

Figure 1: Pseudo-C++ Taint Information Definitions. When we refer to instances of the `TaintInformation` class we include instances of all subclasses.

3. SYSTEM DESCRIPTION

`PINNACLE` consists of three high level components — a data tracking tool (`DT`) that monitors data as as it is processed by an application and generates alerts on events of note; a collection of static analysis scripts (`SA`) that process these alerts and the user interface (`UI`) that is integrated with `IDA` [14].

3.1 Data Tracking and Alerts

`DT` is implemented as a shared library on top of the PIN binary instrumentation framework [17]. We define `M` as the set of valid program memory addresses and `R` as the set of valid CPU register identifiers. For simplicity, in the remainder of this paper we will refer to the set of identifiers given by $M \cup R$ as `I`.

For each flag of the CPU's flags register we define a separate identifier in the set `R`, e.g. `CF` to identify the carry flag. We also define a separate identifier for each 8, 16 or 32 bit subregister. As an example, the various components of the `RAX` register can be referenced through `RAX`, `EAX`, `AX`, `AH` and `AL`.

A shadow data store `S` is maintained to keep metadata about each register or byte of program memory. We can consider `S` to work as a standard map from elements of `I` to instances of `TaintInformation`. Subclasses of the `TaintInformation` class can differ in terms of their attributes depending on the source of that taint information and the taint propagation mechanism in use. The details of this class will be elaborated on in section 3.1.1, while its definition is provided in Figure 1.

On starting a program, or attaching to one, the following holds:

$$\forall x \in I : S[x] = NULL$$

A memory location or register $x \in I$ is defined as *tainted* if $S[x] \neq NULL$.

DT is concerned with introducing tainted data, propagating it across memory locations and registers, and generating alerts when an instruction in combination with the metadata in S matches against a defined set of rules.

3.1.1 Introducing Tainted Data

DT uses the instrumentation mechanisms provided by PIN to update the shadow data store S. PIN allows one to instrument at varying levels of granularity but, for the purposes of introducing new tainted data, we instrument at the function level. With compiled code we need not limit ourselves to higher level function boundaries but can instead consider any arbitrary chunk of assembly code to be a function. However, practically speaking we will choose code that corresponds to a traditional higher level function as this is usually the level of granularity we require when considering the introduction of tainted data.

For any function f that we consider to introduce tainted data we define a pair of functions (f_{pre}, f_{post}) that work together to update S with new taint information. f_{pre} is run before the execution of f and can access the parameters of f. f_{post} is run after the execution of f and can access its return value and any other outputs. For each location $x \in I$ that f updates with attacker controlled data, (f_{pre}, f_{post}) will create a new metadata instance d that is a subclass of `TaintInformation` and update S via $S[x] = d$. This indicates that the memory location or register identified by x is tainted and metadata is provided by d.

During vulnerability discovery the reason for marking a location as tainted is typically to identify it as being influenced by an attacker and then to determine how that influence may effect the rest of the program's control and data flow. *Influence* is a matter of degrees however [18] and the level of control one may have over a value in memory or registers can vary. In DT we reflect this fact by allowing for data to be marked as either directly or indirectly tainted.

For each $d \in (d_0, \ldots, d_n)$ created by (f_{pre}, f_{post}) a subclass of `TaintInformation` must be selected. Two subclasses are available, `DirectTI` and `IndirectTI`, which can be seen in Figure 1. We will say that a location $x \in I$ is *directly tainted* if $S[x]$ is an instance of `DirectTI` and *indirectly tainted* if $S[x]$ is an instance of `IndirectTI`.

The function pair (f_{pre}, f_{post}) are manually created so a developer/user of DT can use their own judgement as to the type used for each $d \in (d_0, \ldots, d_n)$. As we do not perform any automatic input generation based on the taint information the correctness of the chosen type is not critical. However, the type of influence an attacker has is used in prioritizing results and thus it does have some importance.

`DirectTI` is used when we consider the data stored at the register or memory location identified by $x \in I$ to be directly controlled by the attacker. For example, when a value is read into a program unconstrained and unchanged from an attacker controlled source, such as a socket or file descriptor, we create a `DirectTI` instance d for each written

$(w_0, b_0) \leftarrow \{(r_0, b_0), (r_1, b_0)\}$
$(w_0, b_1) \leftarrow \{(r_0, b_1), (r_1, b_1), (r_0, b_0), (r_1, b_0)\}$
$(w_0, b_2) \leftarrow \{(r_0, b_2), (r_1, b_2), (r_0, b_1), (r_1, b_1)\}$
$(w_0, b_3) \leftarrow \{(r_0, b_3), (r_1, b_3), (r_0, b_2), (r_1, b_2)\}$
$flags = (ZF, CF, OF, PF, AF, SF)$

Figure 2: Syntactic specification for the *add reg32, mem32* instruction

memory address x and then set $S[x] = d$. Direct sources are typically functions such as *read*, *recv* and so on.

Alternatively, we may use `IndirectTI` when we consider the data stored at $x \in I$ to be indirectly under the control of an attacker. Indirect sources of control are more diverse and sometimes less obvious than their direct counterparts. Often indirect influence arises when an output from a function has a control flow dependency on a location $x \in I$ for which $S[x] \neq NULL$. A straightforward example of this form of indirect influence is the *strlen* function. If a program contains a call to *strlen(s)* where s is controlled by an attacker then the return value is certainly influenced, albeit indirectly. For such cases we can connect the new `IndirectTI` objects to the `TaintInformation` objects of the data which induced the control flow dependency.

For other sources of indirect influence there may be no clear data or control flow dependency between the updated location $x \in I$ and another location $y \in I$ for which $S[y] \neq NULL$. For instance, the return value of the `recv` function can be indirectly controlled by an attacker based on the number of bytes sent and also by closing the connection. For such cases there is no parent `TaintInformation` instance to connect the new `IndirectTI` instance to but as we are not aiming to automatically generate new inputs this is not problematic.

3.1.2 Propagating Taint Information

After the first execution of a (f_{pre}, f_{post}) pair that updates S such that $(\exists x \in I \mid S[x] \neq NULL)$ is true the taint propagation mechanisms of DT are enabled. From that point onwards every instruction executed is hooked to detect if it operates on bytes that are tainted and, if so, to propagate that taint information to any written registers, flags or memory locations. When this process starts a global tainted instruction counter is initialized to zero and incremented on each instruction that propagates tainted data. This is simply used later to assist in reconstructing traces.

For each instruction we define the set of outputs $U \subset I$ as the registers, flags or memory locations that are modified by the instruction. On the *x86/x64* architectures each instruction may have multiple outputs. Each output $o \in U$ is a function over a set of inputs $N \subset I$, where N may be the empty set $\emptyset$. If we define the set of tainted inputs T as $\{i \in N \mid S[i] \neq NULL\}$ then *taint propagation* occurs when, for a given output o, $T \neq \emptyset$. When these conditions are met we propagate the information from T to o in S.

Before discussing propagation we must first mention how multi-byte operations are processed to produce the required relationship between inputs and outputs. For each *x86/x64* instruction that we handle we provide a map M that describes the relationships between inputs and outputs in terms of the instruction's operands. This is a syntactic description of the relation from inputs to outputs with no semantic information. For example, for the *add reg32, mem32* instruction the map looks as described in Figure 2.

(w_n, b_m) stands for the m^{th} byte of the n^{th} write operand and (r_n, b_m) stands for the m^{th} byte of the n^{th} read operand. If we inspect the mapping for the second output byte we can see that it defines the output as a factor of both the second bytes of both inputs *and* the first bytes of both inputs. This is to account for potential carrys in the addition. For instructions like *shl reg32, cl*, where we do not know exactly which inputs may contribute the which output, we over-approximate. Typically this is done by saying each output $o \in U$ is a factor all elements of N. The map also defines the flags that may be updated by the instruction.

We begin propagation by converting each $(op_n, byte_m)$ pair to an identifier $i \in I$. For registers this is done by means of another map which, for any register operand and byte offset, returns the correct identifier used to reference that particular subregister, or byte. For memory addresses we can simply compute the identifier by adding the byte offset to the base address. By iterating over M we can then produce a set of tuples $R = (o \in U, N \subset I)$ defining the byte level relationships between inputs and outputs. This provides us with the required information to update $S[o]$ for each tuple.

Input Tracking.

`DT` allows for two different taint propagation modes. In the first mode, which we will call the *tracking mode*, each instruction simply spreads the taint status of inputs through to outputs without recording the instruction address, or any other identifying information, responsible for the taint propagation. In this mode, for any given location $(i \in I \mid S[i] \neq NULL)$ we can tell which program input bytes may have contributed to the value in location i. However, we cannot tell the instructions that were responsible for manipulating and moving that tainted data from the program input to location i.

For each tuple $(o \in U, N \subset I) \in R$ we begin by retrieving the corresponding `TaintInformation` instances for each input $i \in N$, producing a set of `TaintInformation` instances V. It is possible that an element of V is in fact a `CompoundTI` instance, representing the join of influence from multiple input bytes. If this is the case we recursively replace that `CompoundTI` instance with the members of its `components` attribute. This is done to ensure that `CompoundTI` instances that are created, used to influence a new location and then overwritten can be safely deleted, thus minimizing the amount of memory required. Once this has been performed for all elements of V the final set consists only of `DirectTI` and `IndirectTI` instances.

If V contains two or more elements we create a `CompoundTI` instance and set its `components` attribute equal to a vector constructed from V. The end result is that every element of S is either a `DirectTI/IndirectTI` instance or a `CompoundTI` instance with a `components` attribute consisting of instances of these types.

Input Tracing.

In the second mode, which we will call the *tracing mode*, each instruction creates new `TraceTI` instances for each output, allowing us to track exactly which instructions were involved in the computation of any given byte of tainted data. Under this mode, for any given location $(i \in I \mid S[i] \neq NULL)$ we can construct a tree by recursively traversing the parents of each `TraceTI` instance. The leaves of this tree will be `DirectTI/IndirectTI` instances and from it we can present the user with a list of all instructions involved in the computation of i.

To propagate the taint information we first iterate over the tuples $(o \in U, N \subset I) \in R$ retrieving the corresponding `TaintInformation` instances for each input $i \in N$, producing a set of `TaintInformation` instances V. In this mode, an element of V may be a `DirectTI`, `IndirectTI` or `TraceTI` instance. The key difference is that we do not expand `TraceTI` instances like we did `CompoundTI` instances. Instead we create a new `TraceTI` instance and set its `parents` attribute equal to a vector constructed directly from V. This ensures that for any element of S we have available the information to construct the tree linking it back to input data. The other difference is that `TraceTI` objects store the global tainted instruction counter in the `sequence_num` attribute. This is integral in later reconstructing the order in which updates were made to values that are not derived from one another. It is also used to align the individual byte updates performed in a single multi-byte instruction when processing traces.

Value Constraints.

The above processes allow us to track or trace data between a taint source and a taint sink. However, it is also desirable that we would be able to report on the *constraints* imposed on this data. For example, if the current path through the program contains a `test` instruction that operates on tainted data, followed by a conditional jump, then this may impose a restriction on the set of values that data may have and still trigger the same path. Such information is useful in differentiating safe from unsafe arithmetic performed by a program.

In order to facilitate this, we allow a user to enable an optional mode that propagates taint information to the CPU flags register. For each instruction that updates CPU flags and operates on tainted data we create a `FlagUpdateTI` instance. The `parents` attribute of this instance is set to a vector of all `TaintInformation` instances across all operands of that instruction. For all flag identifiers $i \in I$ updated by the given instruction, S is then updated by setting $S[i]$ equal to the `FlagUpdateTI` instance.

On a conditional jump instruction we then retrieve the `FlagUpdateTI` instance for each flag used and merge their `parents` vectors together into a new set C. C is a set of all `TaintInformation` instances that may affect the outcome of the conditional jump. A tuple (C, a, n) is then appended to a list used to represent the path condition, where a is the current instruction address and n is the current value of the global tainted instruction counter. This list is periodically flushed to disk.

The Performance/Accuracy Tradeoff.

The primary downside to a syntactic, rather than semantic, approach to data flow specification is related to accuracy. Without access to the semantics of an instruction it is possible to conclude that a location in S is tainted when it is not. As an example, consider the instruction `xor eax, eax` and assume that the EAX register is tainted. If the handler for `XOR` shares the specification of other bitwise operators, like `AND` and `OR`, we would incorrectly conclude that the EAX register is still tainted after this instruction. Taking the `XOR` of a location with itself in order to clear it is a common operation however, and so we can specifically encode a handler for this situation to avoid an erroneous result.

In general, such a solution is not possible without bit-level taint tracking and semantic specifications for instructions. As a result, in some situations there will be elements of S that are incorrectly marked as tainted. The following code is a contrived example but demonstrates the problem:

```
0:  cmp al, 0xf0
1:  jne EXIT
2:  and al, 0x0f
```

Assume that the AL register is tainted at address 0. The task is to decide at address 2 whether this still holds. At 2 the only information available to DT is that $S[AL] \neq NULL$ as there is no facility for reasoning about the semantics of the path condition to detect the implied boundary on the value of the AL register. We must therefore conclude that it is possible for at least one of the lower four bits of AL to be set and thus that $S[AL] \neq NULL$ still holds after the execution of the instruction.

While this issue is worth noting, in our experience it is not a common problem that interferes with the users confidence in the data presented. In the domain of concolic/symbolic execution over-tainting can still occur [15], albeit for different reasons. Similar to our work however, this is rarely a significant problem.

3.1.3 Tainted Data Sinks

DT allows for configurable and on-demand *sinks* to execute a handler once they detect tainted data. The purpose of these handlers is usually to inform the user once attacker influenced data reaches code that may be of interest. We provide handlers for common sinks e.g. memory allocation or data copying functions, like `malloc` and `sprintf`, used with tainted parameters; instructions with a `REP` prefix and a tainted ECX register; conditional jumps based on tainted flags and so on. However, as one reverse engineers an application it is common to want to set application-specific handlers. We support this by allowing users to specify an $(address, shared\_library)$ pair and inserting hooks from their shared library at the address provided. This allows monitoring of custom memory management routines, third party libraries, and any other code without modification to the core of DT.

Depending on the type of sink, the number of hooks required and their interactions may differ slightly but in general they operate as follows. For a given sink k we define a function pair (k_{check}, k_{alert}), where k_{check} inspects a set of program locations given by $L \subset I$ and k_{alert} logs sufficient information to allow prioritization of results by static analysis and/or, importing of the results into IDA for inspection by the user. Let $T = \{l \mid l \in L, S[l] \neq NULL\}$, then k_{alert} is executed if $T \neq \emptyset$. k_{alert} will typically log the address of the sink along with the information on all members of T. If the *tracking* propagation mode is in use then this information will be limited to the tainted inputs that influenced the members of T. However, if the *tracing* propagation mode is in use k_{alert} will have access to this information as well as a trace of every instruction that modified those input bytes between their introduction and the sink k. For each element of T the sequence of parents back to the taint source will be logged, including instruction addresses and the `sequence_num` attribute.

3.2 Post-Processing and User Interface

DT produces a variety of outputs containing information useful in the processes of vulnerability identification and reverse engineering. The following categories of information are made available:

A Program Trace A user can choose to log each instruction the first time it processes tainted data or, alternatively, a log of each instruction every time it processes tainted data.

Function Alerts Each (k_{check}, k_{alert}) pair as described above may result in a log of inputs, or inputs and then modifying instructions, every time that function is called with tainted parameters.

Instruction Alerts Certain instructions can also be hooked with a (k_{check}, k_{alert}) pair and produce the same style output as function hooks. Of particular interest are the execution of `REP` prefixed instructions with a tainted ECX register and conditional jumps that are influenced by tainted flags.

The Path Constraints If this feature was enabled then a log of the branch points conditional on user input, as well as the bytes they were influenced by will be available.

Applications that perform significant processing on attacker controlled inputs can result in large amounts of data being logged across these categories. Table 2 shows the number of unique instructions acting on tainted data with per tested application. These figures range from approximately 5000 instructions to upwards of 25,000. When working on smaller targets it may be sufficient to simply display this information in a table to the user. For larger targets however, such as those discussed in section 4, it is necessary to process, rank and then display this information in such a way as to focus the user on the most relevant results.

The processing and ranking functionality that makes up SA consists of several Python scripts that run on top of IDA. As input they take the results of DT and as output they integrate information on tainted data flow into the disassembly view provided by IDA, as well as extending it with new tables of relevant information. The analysis component SA and user interface components UI are quite closely linked so we will explain their features in tandem.

3.2.1 Attack Surface Identification

The first stage of reverse engineering driven vulnerability detection requires one to find the attack surface of a program. We will loosely define the attack surface as code with data-flow or control-flow that can be directly or indirectly influenced by an attacker. This stage is critical as typically one wants to find the most security flaws in the least amount of time and the ability to prioritize different parts of the code for review is crucial to this process.

Often applications come with their functionality spread through a number of shared libraries, each with potentially hundreds or even thousands of individual functions. For example, one of the applications discussed in section 4 contains 43 DLLs, several of which have more than a thousand functions. PINNACLE makes it easy to quickly discover which of these process attacker controlled input and which do not.

Previously, a popular approach to this problem has centered around a technique called *differential debugging* or *differential reverse engineering* [16, 23, 2]. Under this approach one attempts to discover the code responsible for particular features of a piece of software by recording the basic blocks executed when that functionality is not used and then again when it is used. By taking the difference between these two sets of basic blocks one usually ends up with a set of basic blocks that are related to the feature of interest. Differential debugging is a simple but relatively effective solution to the problem. However, using data tainting information we can get much more fine grained information. Instead of concluding that a collection of basic blocks may be somehow related to the feature in question we can conclude the precise instructions responsible for processing every byte of user input.

We break the process of attack surface identification down into two stages, firstly at the library level and then at the function level.

Library Processing.

Using the program trace we group the recorded instructions by DLL and count them. Following this we also group and count each function and instruction alert. The results are then displayed in a table to the user who can easily see those DLLs that perform the most work on attacker controlled inputs and also those that generate the most alerts for events of note.

Often applications will encapsulate functionality related to IO within a particular library and then tainted data will flow from this library through a number of other libraries before reaching a taint sink. If the tracing mode is enabled then on an alert we can create a graph visualizing this information flow for the user.

Attack Surface by Function.

Using the information provided by the *program trace* and the analysis API of `IDA` we update the disassembly view of `IDA` to highlight every instruction that has processed tainted data. A count per function of each alert generated by that function, its cyclomatic complexity, the number of instructions in the function and the number of tainted instructions is then placed in a sortable table and displayed. Any functions that introduce tainted data via a taint source are also highlighted.

The information displayed allows a user to get a feel for which functions perform the most processing and are potentially the most interesting locations to begin auditing. Currently we do not attempt to perform any automated ranking of functions but allow the user to select from the above metrics and sort the results.

3.2.2 Alert Prioritisation and Display

For each alert generated by a k_{alert} function the output will depend on the propagation mode in use. Under the tracking mode the output will be the information from a set of `DirectTI` and/or `IndirectTI` instances that allow us to identify a set of byte inputs from a file or network stream. Under the tracing mode the output is much more verbose and allows us to link the bytes of interest at the point of the k_{alert} execution to a set of input bytes through each instruction that may have transformed them along the way. The resulting information is a set of input bytes and a sequence of instruction addresses $A = < a_0, \ldots, a_n >$. Processing the produced log into this set of address is not discussed here due to space considerations. The process is relatively trivial and simply a matter of iterating over the produced log and building the set of instruction addresses, ordered by the tainted instruction counter.

For both the tracing and tracking mode we may also have access to the path condition. This can also be processed into a set of addresses $< c_0, \ldots, c_n >$ that identify tainted conditional branches. Assuming we are operating under the tracking mode the addresses of the conditional branches can be integrated into A using the associated tainted instruction counter values.

Due to the volume of alerts that may be generated it is necessary to have some form of prioritisation mechanism. We do this by performing a simple analysis of every instruction in A. If we detect an arithmetic operator like `add` the *has_arith* flag is set, if we detect a bitwise operator like `and` the *has_bitwise* flag is set and finally, if there are any conditional jumps in A the *has_jcc* flag is set. The attributes are ranked as follows:

$$\texttt{has\_arith} > \texttt{has\_bitwise} > \texttt{has\_jcc}$$

The ranking of a trace is given by the least of the attributes assigned to it. When two traces have the same ranking, the one with the shorter sequence of addresses is ranked higher.

Naturally, the *has_arith* and *has_bitwise* attributes only apply if we are operating under the tracing mode. In the tracking mode we have no choice but to simply rank traces without conditional jumps higher than those with them.

The list of alerts are then displayed to the user via a table in IDA, where the columns contain the name of the alert followed by the above calculated attributes. Once an alert is selected another table is generated listing every instruction in A. The disassembly view is also updated to color each of the instructions listed in A.

4. USAGE AND EVALUATION

For the purposes of this paper we have evaluated `PINNACLE` on the targets listed in table 1. The targets are all complex, commercial software and available only in binary form [1]. The purpose of the applications are diverse, including one of the most frequently encountered image editing suites (*ImageSuite*); a multi-threaded game server that typically has 50,000 online players at any one time(*GameServer*); the desktop control center for a common brand of network printer (*PrinterControl*) and a popular industrial design suite (*InDesign*). Between them their functionality is scattered across several hundred DLLs and several hundred thousand functions.

Reverse engineering any of these targets for the purposes of vulnerability detection can be a time consuming process but in this section we will demonstrate how `PINNACLE` enables a user to quickly discover the program's attack surface and then audit that attack surface in a prioritized manner. Using `PINNACLE` we were able to discover security vulnerabilities in all targets in a short amount of time.

[1]The targets have been anonymised as the security flaws found are unpatched

4.1 Performance

The primary output of PINNACLE is intended for human consumption, and thus an evaluation of the quality of this information is subjective. An idea of the usefulness of this data is provided in section 4.2 where we go through a number of serious flaws found with PINNACLE. However, we can provide metrics on the run-time impact of PINNACLE during the instrumentation phase as well as the size of the identified attack surface relative to the overall quantity of code executed. It is important to measure the run-time impact of the tool as this is the most time consuming part of the analysis. As mentioned in the introduction, excessively long run-times may not integrate well into a user's work flow. By measuring the difference between unique executed instructions and the subset of these that process tainted data we can get an idea of whether the wait for this information is worthwhile, versus an alternative approach such as differential debugging.

Table 1 shows the run-time impact of PINNACLE in comparison to a basic block coverage tool, used for differential debugging. The experiments were performed on a 32-bit dual-core Intel 2.20Ghz processor with 3GB of RAM running Windows 7 [2]. The basic block coverage tool is taken as a base and and the results of other methods are a factor of that. The actual program run time in seconds is also provided. The tasks performed were to scan a document (*PrinterControl*), load an image from disk (*ImageSuite*), connect a client to the game (*GameServer*) and open a design specification (*IndDesign*).

We can see that, as expected, PINNACLE has a higher run-time impact than simple basic block coverage tracking. However, the run-time impact is not sufficiently large as to prevent the integration of the tool into a reverse engineer's work flow. Interestingly, in some cases the impact of running PINNACLE in *tracing* mode is not significantly higher than running in *tracking* mode. In these cases we can get the benefits associated with the extra information provided, without run-times that might be inconvenient to the user. The two exceptions were with *GameServer* and *IndDesign*.

While running the *GameServer* under the *tracing* mode the client displayed a connection timeout error after approximately 80 seconds. By this point the client and server had already exchanged a number of requests and responses so we were still able to gather information on a useful portion of the connection protocol. With *IndDesign* under tracing mode the system eventually ran into an out-of-memory (OOM) condition while loading the input. The *IndDesign* application performs extensive decompression on the input. As a result of this, the application performs significantly more processing of tainted data, in comparison to the others. This leads to an OOM condition in PINNACLE due to the large chains of objects necessary to maintain information on each instruction operating on tainted data. Despite this, we were able to find a number of security vulnerabilities using the information from the tracking mode alone. However, both this OOM and the timeout on *GameServer* show that there is still work to be done on optimizing the instrumentation phase of our approach.

[2] PINNACLE also works on Linux and x86-64

One of the most useful applications of PINNACLE is in attack surface identification. In table 2 we can see a comparison of the number of loaded DLLs, including system DLLS, versus the number of DLLS that actually process tainted data. We can clearly see that taint tracking allows us to significantly cut the number of DLLs we may consider interesting from an attackers perspective. A large number of DLLs may still remain if we only consider those that process tainted data, however, using the data flow graph mentioned in section 3.2.1, along with basic metrics on the number of tainted instructions per DLL, a user is provided with useful guidelines on where significant processing is performed.

Table 2 also provides metrics on the number of unique instruction addresses executed versus the number of those that process tainted data. PINNACLE cuts the attack surface down to a fraction of those instructions actually executed e.g. by a factor of 120 in the case of the *PrinterControl* application. This saves significant amounts of time for a reverse engineer and is a more accurate solution in comparison to differential debugging.

The ability to focus quickly on code that is influenced by an attacker, in combination with direct targeting towards potential vulnerabilities, means that a user can be more efficient than otherwise. We have tested this hypothesis by real world use of PINNACLE.

4.2 Vulnerability Discovery

4.2.1 *Remote Overflow in* PrinterControl

In the application *PrinterControl* we found several vulnerabilities that can lead to remote compromise of the device attempting to interact with the network printer. One such vulnerability is found in the initialization stages of the application. When *PrinterControl* starts it sends a broadcast request over the network to identify an available printer/scanner. Any device on the local network can see and respond to these requests. By starting *PrinterControl* under PINNACLE we were able to detect a serious vulnerability resulting from insecure parsing of responses to this initial broadcast.

Several function alerts related to memory allocation using a tainted size are generated during this initial phase. The majority are in a single DLL, which we will refer to as *Parse.dll*. Using the data flow graph available, we can see *Parse.dll* uses a second DLL *Net.dll* to read data from a network socket through an exported function *Net.dll!Read*. In *Parse.dll* there are 67 call sites to the allocation function used, which PINNACLE narrows down to 10 as being dependent on attacker provided input. In all cases the allocation size is first read from data received over the network, then that same size is provided to *Net.dll!Read* to fill the allocated buffer. Using the taint information from PINNACLE we can quickly inspect these call sites and conclude that all uses of allocated buffer and attacker influenced size are safe.

However, if we instead inspect *Parse.dll* by looking at call sites to *Net.dll!Read* where the size is tainted the results are more interesting. Under this view we can see a single call to *Net.dll!Read* where the size to be read is taken from a previous network packet and the destination buffer is the stack. PINNACLE can provide us with each check and modification performed on the size value, allowing us to confirm that any non-zero size can be provided.

Pseudo-code for the vulnerability can be seen in Figure 3. The flaw itself is trivial to spot once the code has been iden-

Application	BBCov	PINNACLE Track	PINNACLE Trace
PrinterControl	x1 (58s)	x4 (210s)	x5 (290s)
ImageSuite	x1 (71s)	x3 (240s)	x13 (945s)
GameServer	x1 (12s)	x6 (68s)	Timeout
IndDesign	x1 (615s)	x5 (3000s)	OOM

Table 1: Runtime impact of various approaches to attack surface identification and vulnerability discovery

Application	# Loaded DLLs	# Tainted DLLs	# Unique Exec. Instrs.	# Unique Tainted Instrs.
PrinterControl	104	30	712134	5908
ImageSuite	165	33	603922	25263
GameServer	59	18	242671	12575
IndDesign	289	55	2811259	94598

Table 2: Actual loaded DLLs and executed instructions versus those influenced by attacker input

```
int res;
size_t next_read;
char buf[12];

...

res = read_data(&buf, 12);
if (!res)
    return;

...

next_read = get_size(buf + 6);
res = read_data(&buf, next_read);

...
```

Figure 3: *PrinterControl* remote overflow vulnerability

tified and the work performed by the called functions is figured out. PINNACLE makes both of these processes easier by providing direction on where to look and data flow information that can give intuition as to the sources and sinks for tainted data. We also found that PINNACLE made it easier to verify the non-vulnerable uses of *Net.dll!Read*. By highlighting the relevant code in IDA less time needed to be spent on each case and thus we could move on to more interesting cases.

4.2.2 Heap Overflow in IndDesign

The *IndDesign* application uses a compressed file format that requires extensive processing to decompress and parse. Using PINNACLE we were able to quickly locate the decompression algorithms and then discover a heap overflow in a related function. The impact of this bug is that opening an attacker provided design can lead to arbitrary code execution.

The file format used consists of multiple segments and a table that provides a size and checksum for each segment. We identified the functions responsible for the integrity checks and decompression by searching the tainted instructions for conditionals based on the checksums located in this table. As PINNACLE provides the index into the file of each tainted byte processed by each instruction, this search can be automated using a small script. Immediately after the integrity checks the decompression takes place.

```
int process_segment(char *data, size_t data_sz)
{
    size_t decompressed_sz = *(size_t*) data;
    if (decompressed_sz > 0x1000)
        return -1;

    char *decompressed_data = malloc(decompressed_sz);
    decompress(data, data_sz, decompressed_data,
               decompressed_sz);
}

void decompress(char *compressed_data,
    size_t compressed_sz,
    char *decompressed_data,
    size_t decompressed_sz)
{
    char *cur_byte = compressed_data;
    size_t copy_counter = 0;

    while (*cur_byte++ == CONSTANT1)
        copy_counter += CONSTANT2;

    while (copy_counter > 0) {
        expand_data(cur_byte, &decompressed_data);
        cur_byte++;
        copy_counter--;
    }
}
```

Figure 4: *IndDesign* heap overflow vulnerability

```
...

recv_len = recvfrom(s, buf, len, flags, from,
            fromlen);

...

computed = compute_checksum(buf, recv_len);
sent = (*(char*)(buf + recv_len - 2)) << 8 |
            (*(char*)(buf + recv_len - 1))

if (computed != sent)
    checksum_err();
...
```

Figure 5: ***GameServer*** **unchecked return value usage**

An alert on a tainted argument to the `malloc` function directs our attention towards the `process_segment` function shown in Figure 4. By inspecting the data flow information we can see that `decompressed_sz` is a 32-bit field, taken from the attacker provided file. Its range is restricted to values between 0 and 0x1000 and a buffer is allocated to store the decompressed data based on this information.

By following the `decompress` call in `IDA` we can see, via the taint information, that the first `while` loop is bounded based on comparisons with attacker provided data. By controlling the input bytes to equal `CONSTANT1` we can manipulate `copy_counter` to hold values far larger than 0x1000. The loop calling `expand_data` is then bounded using the `copy_counter` variable instead of `decompressed_sz`. In this case it is the fact that the copy loop is *n*ot bounded by the same attacker controlled value used to allocate the destination buffer that is conspicuous.

By inspecting `expand_data` we can immediately tell, based on the data flow information, that it either copies attacker controlled bytes directly to the output buffer or decompresses them into a longer sequence. The end result of this is that we can manipulate the decompression algorithm to overflow the `decompressed_data` heap buffer with a controllable amount of data. Even with out the full tracing information available `PINNACLE` makes the process of tracking down such issues and understanding the surrounding code much easier than doing so using just a debugger, disassembler and coverage information.

4.2.3 *Unchecked Use of* recvfrom *Return Value in* GameServer

The *GameServer* application serves as a useful demonstration of a bug that is quickly uncovered when indirect tainting is used. The main processing loop of *GameServer* begins by reading data from a socket, then computes a checksum of this data and compares it against a value stored in the last two bytes of the received data.

Figure 5 shows pseudo-code for this initial function. Upon execution of this function the data read into `buf` will be marked as directly tainted while the value in `recv_len` will be marked as indirectly tainted. On the computation of the value of the `sent` variable `PINNACLE` creates an alert, if memory read/write based alerts are enabled. When the alerts are processed and loaded into `IDA` this will be assigned a high priority as a value indirectly controlled by an attacker is used, unchecked, in a memory dereference. If an attacker provides data of length 0 or 1 to the server the computation of `sent` will access 2 or 1 bytes *before* the `buf` buffer. This flaw will not result in malicious code execution but under certain circumstances could lead to a denial-of-service condition if `buf` is located within a byte of a page boundary and the previous page is unmapped.

While the flaw itself is not serious this example provides motivation for ensuring we are thorough in noting and tracking indirect sources of attacker influence.

5. CONCLUSION

Understanding software, especially software in binary form, can be both time consuming and difficult. However, it is also a necessary requirement in order to solve a diverse set of problems. In this paper we have presented techniques for attack surface discovery and vulnerability detection that are designed to increase efficiency in the work flow of a reverse engineer. Our results demonstrate the usefulness of attempting to solve *just enough* of a problem automatically, while relying on a human for creativity and decisions in the use of that data.

Much remains to be done in discovering the most effective ways to present users with the results of automated analysis and, likewise, on how best to extract their domain specific knowledge and integrate it with automatically generated models. We believe that in tandem with the development of modeling techniques it is also important to address these issues of usability and work flow integration.

Alongside the presented use of the data gathered, we also envisage it being beneficial in providing guidance to automated dynamic and static testing algorithms. Path and code region prioritisation is a challenge under most analysis approaches and it makes sense to prioritize code that is known to be subject to attacker influence. Furthermore, this data may allow one to begin analysis deeper in the code than necessarily starting from generic points of influence, such as file or socket access.

6. REFERENCES

[1] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. Aeg: Automatic exploit generation. In *Network and Distributed System Security Symposium*, pages 283–300, Feb. 2011.

[2] D. Blazakis. Differential reversing (or some better name). `http://dion.t-rexin.org/notes/2009/09/29/differential-reversing/`, 2009.

[3] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. Bap: a binary analysis platform. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 463–469, Berlin, Heidelberg, 2011. Springer-Verlag.

[4] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 143–157, Washington, DC, USA, 2008. IEEE Computer Society.

[5] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and communications*

security, CCS '07, pages 317–329, New York, NY, USA, 2007. ACM.

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

[7] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2), 2008.

[8] V. Chipounov and G. Candea. Reverse engineering of binary device drivers with revnic. In *EuroSys*, pages 167–180, 2010.

[9] V. Chipounov, V. Kuznetsov, and G. Candea. The s2e platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2, 2012.

[10] D. Duran, M. Miller, and D. Weston. Security defect metrics for targeted fuzzing. In *CanSecWest*, Vancouver, Canada, Mar. 2011.

[11] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

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

[13] S. Heelan. Automatic generation of control flow hijacking exploits for software vulnerabilities. Msc. dissertation, University of Oxford, September 2009.

[14] Hex-Rays. IDA. `http://www.hex-rays.com/products/ida/index.shtml`, 2005.

[15] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2011.

[16] J. Koret. MyNav, A Python plugin for IDA Pro. `http://joxeankoret.com/blog/2010/05/02/mynav-a-python-plugin-for-ida-pro/`, May 2010.

[17] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[18] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis. Path-exploration lifting: Hi-fi tests for lo-fi emulators. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, UK, Mar. 2012.

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

[20] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pages 65–74, New York, NY, USA, 2007. ACM.

[21] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS*, 2011.

[22] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, Dec. 2008.

[23] Zynamics. BinNavi. `http://www.zynamics.com/binnavi.html`, 2010.