Bugalyze.com - Detecting Bugs Using Decompilation and Data Flow Analysis

Silvio Cesare
Deakin University

Abstract—**Detecting bugs in programs is important to establish trusthworthy software. To achieve this, static analysis on source code is a common approach to discover bugs. However, source code is not always available, as in the case of a black box penetration test. Even if source code is available, it still remains prudent to test that compilation and link editing has not introduced new bugs into the software release. We propose a system, Bugwise, that performs bug detection on x86 binary-level programs. Our system employs static analysis and the novel application of decompilation to make that analysis tractable. We use data flow analysis as our static analysis which is able to detect a number of bug classes including use-after-frees, double frees, and buffer overflows using environment variables. We have also provided limited access to our system as a web service for the public to use. Our results have found tens of bugs and vulnerabilities in Debian Linux where we scanned the entire package repository of that Linux distribution. Bugwise shows that traditional static analysis can be applied to binaries through the use of decompilation techniques.**

Keywords—bug detection; static analysis

I. INTRODUCTION

Detecting bugs in software is an important task to help improve the security and trustworthiness of the computer industry. Detecting bugs in software can be done at any time during the software lifecycle. Ideally, all bugs are found during the testing phase before the software is deployed. In reality, software verification and testing is not applied or applicable on all software, or does not find all possible bugs in any given time. Therefore, software testing is done throughout the entire lifetime, and as new bugs are found patches are made and pushed out for deployment. This is exacerbated by the fact that maintaining software and supplying new features introduces new bugs. Therefore it is impractical that all bugs are found at the creation of the initial version of the software.

Formal methods have long touted the ability to secure software against specific classes of software defects. Formal methods include such techniques as formally defined specifications, model checking, theorem proving, and well defined semantics of programs which can have properties examined in a mathematical sense. Unfortunately, formal methods have only been used on systems which are prepared to pay high costs in terms of budgets and development. In practice, most vendors do not use formal methods because those methods are impractical in some sense, whether it be financial or technical limitations. Software testing is a dominant technique today used by large and small vendors alike to ensure some level of trustworthiness in their code base.

Software testing can be performed to a variety of degrees and it is common that only large software vendors apply best practices or state-of-the-art techniques to discover new bugs and vulnerabilities. Smaller vendors have fewer developers, a smaller quality assurance testing team, and some may not even have a team dedicated to testing the robustness of the software or security personnel dedicated to performing code review on their products.

This situation leads to the condition that bug discovery is not always performed by the software vendor. Even large companies such as Google and Microsoft have bug bounties for users to submit bugs and vulnerabilities in exchange for monetary or other rewards. Penetration testing is an industry based on the fact that software vendors, and other people who use IT require external or separate teams to comb over their software, products, and infrastructure to discover bugs and vulnerabilities.

There are many types of penetration tests, all with a variety of scopes which set the prerequisites for a penetration tester to do their work. Many penetration testers apply commercial off-the-shelf software to perform static analysis on source code to discover possible programming bugs which lead to security vulnerabilities. That is only possible if source code is available. However, there exists a gap within industry in that of auditing proprietary undisclosed software.

A. *Motivation*

A primary motivation for binary-level vulnerability auditing is that of black-box penetration testing. In this scenario, a penetration tester is employed to test the security of the product without having access to the source code. This exemplifies what an adversary has access to in many software deployments. The penetration tester would like to identify if any vulnerabilities exist in the software's binaries much like a penetration tester uses network scanning and vulnerability assessment tools to perform a network based penetration test.

Similar to penetration testing of a vendor's own software, a vendor may want to audit 3rd party software. If a vendor is dependent on 3rd party software it provides assurances to the vendor if a binary-level vulnerability audit of the software shows that it is of an expected reasonable quality without a surplus of bugs and vulnerabilities.

Finally, even if source-level vulnerability assessment has been performed, the final view of the software is the compiled and link edited binary. Therefore, it is important to additionally verify that these processes have not rendered the software vulnerable.

B. Our Approach

The approach our system Bugwise uses is the combination of decompilation [1] and the traditional static analysis technique of data flow analysis [2]. Decompilation recovers high-level source-like information from a low level binary representation. Bugwise does not use full decompilation but instead only recovers local variables and procedure arguments from procedures. The recovery of these variables enables a native variable representation in the intermediate language of Bugwise. We then use data flow analysis on these native variables to show bugs and security properties. An example of data flow analysis is detecting use-after-free bugs where a pointer to memory is accessed after the memory is deallocated. Data flow analysis easily detects this by identifying if the pointer is subsequently used after a free before it is reassigned a new value.

II. Decompilation

A. Wire - A Formal Intermediate Language for Binary Analysis

The first process we perform is translating x86 machine code into an intermediate representation. We take as input an object file and extract executable segments from within it. We then pass these segments to a disassembler. There are two main types of disassemblers: linear sweep and recursive traversal [3]. The linear sweep approach disassembles one instruction at a time from the beginning of the memory segment until the end. The recursive traversal approach disassembles each instruction and follows branches and other control flow transfer instructions. The benefit of the recursive traversal approach is that instructions need not be perfectly sequential and may have padding and other nil operations (nops) between them. Our system Bugwise uses speculative disassembly [3]. Speculative disassembly uses a recursive traversal disassembly and then uses the linear sweep to fill in any gaps.

```
Imark           ($0x80483f5, , )
AddImm32        (%esp(4), $0x1c, %temp_memreg(12c))
LoadMem32       (%temp_memreg(12c), , %temp_op1d(66))
Imark           ($0x80483f9, , )
StoreMem32      (%temp_op1d(66), , %esp(4))
Imark           ($0x80483fc, , )
SubImm32        (%esp(4), $0x4, %esp(4))
LoadImm32       ($0x80483fc, , %temp_op1d(66))
StoreMem32      (%temp_op1d(66), , %esp(4))
Lcall           (, , $0x80482f0)
```

IL before decompilation

```
Free            (%local_28(186bc), , )
```

IL after decompilation

Fig. 1. The intermediate language (IL).

Once the object file has been disassembled, each instruction is translated to 1 or more microcode instructions. These microcode instructions come from our intermediate language Wire [4]. Wire is a register-based three address code (TAC) that simplifies the analysis of native assembly language. It has a small number of instructions and resembles a RISC style language. Wire instructions have no implicit side effects, which simplifies processing.

We use Wire to extract control flow information from the binary. The intermediate language includes branch and other control transfer instructions. In Bugwise, we only extract control flow graphs and call graphs from control transfer instructions that explicitly state the destination addresses. That is, we ignore dynamic dispatches. This approach underapproximates the control flow and makes our system unsound. However, we are still able to detect many bugs using this approximation. For future work, we may look at the dynamic dispatch problem using control flow analysis.

B. Stack Pointer Inference

In many x86 call conventions, the stack is used to pass procedure arguments. The stack is also used to store local variables. The frame pointer in x86 is a general purpose register and gives each procedure and activation record a base reference into its stack frame. If the frame pointer was used exclusively then it would be a simple matter to identify the unique locations on the stack that are being referenced. In reality, the frame pointer is not necessary and many programs choose not to use it for the purposes of obtaining an extra general purpose register and thus leading to greater efficiency in the generated code. In these cases where the frame pointer is not used, references relative to the stack pointer are used instead. The stack pointer points to the top of the stack and calling a procedure, or setting up procedure arguments alter the stack and the stack pointer. Some procedures use different call conventions, and it is during the compilation process that the stack pointer handling is performed to account for procedures which expect the caller to clean up the stack or the callee to clean up the stack. In compiled code, the stack pointer at the start of each basic block should remain constant and changes to the stack pointer are either explicitly determinable by the instruction or as a result of a call modifying the stack in a constrained way. This sets the scene for using linear inequalities to represent these constraints and determine the stack pointer in the procedures. We track the relative stack pointer entering and leaving a basic block in the control flow graph for a procedure.

It is noted that we can make the following assumptions on a procedure:

- The stack pointer entering the control flow graph is 0
- The stack pointer returning from a control flow graph at a return instruction is 0
- The stack pointer leaving a basic block is the same as the stack pointer entering a basic block following its control flow edge

- Simple arithmetic on the stack pointer modifies the stack in a basic block by a known and constant amount
- Call instructions modify the stack in the basic block by a bounded amount

Using linear equalities to infer the stack pointer has been in industrial software, notably the Hex-Rays decompiler.

C. Local Variable Recovery

Local variable recovery is determined by identifying memory access to the stack within the stack frame. Bugwise replaces these memory references with native variables in our Wire IL. It should be noted that local variables can be referenced by either the frame pointer or relative to the stack pointer.

D. Procedure Parameter and Argument Recovery

Procedure parameter recovery is determined by identifying memory access to the stack outside the stack frame. As with local variable recovery, Bugwise replaces these memory references with native variables in our Wire IL format. Additionally, as noted earlier, procedure parameters may be reference via either the frame pointer or the stack pointer.

III. Data Flow Analysis

Bugwise uses data flow analysis to detect bugs. The process of data flow analysis determines information about the values of data throughout the program. There are many types of data flow analysis problems. Classic problems include reaching definitions, which Bugwise uses to detect bugs, and live variable analysis. Both reaching definitions and live variables look at the uses and definitions of variables in programs and are often considered as use-def problems. Bugwise primarily works on use-def problems to determine the presence of bugs in binaries. Many bugs can be considered as a subproblem of use-def problems. For example, double free detection is detecting two uses of a pointer without a definition separating them. Although many data flow problems exist, many of these problems can be abstracted and unified under a common framework of analysis. Data flow analysis is a well studied field so we summarise the algorithms that Bugwise uses.

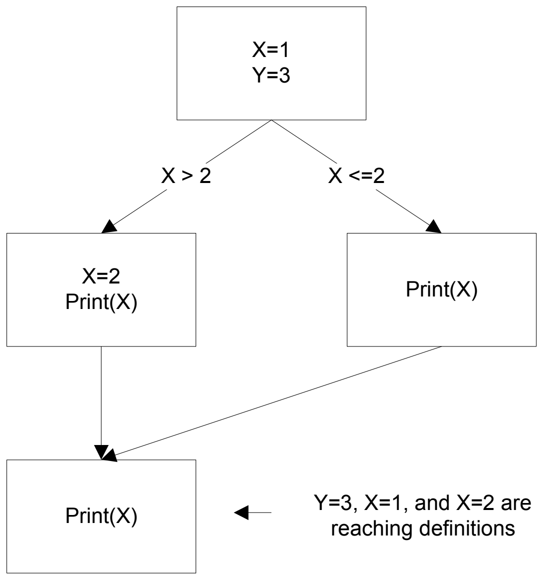

Fig. 2. Reaching definitions.

A. Monotone Frameworks

Monotone frameworks can represent many data flow problems. Monotone frameworks also form the basis of abstract interpretation [5]. The framework establishes a set of data flow equations and an initial condition to represent the data in the programs. There are two components in such a framework that Bugwise uses: 1) a transfer function which takes input entering a basic block and transforms it, delivering the output leaving a basic block 2) a join function which merges the control flow edges representing the data entering a basic block. For a monotone framework to be effective, the transfer and join functions in combination must be monotonic. The monotonic constraint enables the data flow equations to be solvable by the provable presence of fixed points. These fixed points are points where application of the data flow equations does not alter the state of the system. In other words, the solution to the data flow equations is stable. Solving the data flow equations by iterating over the system until a fixed point is reached is described in a later section. Without the monotonic constraint, no fixed points may exist and the equations may not be solvable.

There are two configurations for the analysis: a forward analysis, and a backward analysis. In a forward analysis the data flow equations are:

$$out_b = transfer\_function(in_b)$$
$$in_b = join(\{p \mid p \in predecessor_b\})$$

In a backwards analysis the transfer function and join function is altered slightly:

$$out_b = join(\{p \mid p \in sucessor_b\})$$
$$in_b = transfer\_function(out_b)$$

Both forwards and backwards analysis have an initial state for each basic block.

B. Data Flow Analysis in a Monotone Framework

Monotone frameworks provide a generalised framework for representing data flow problems. Many compiler style analyses can be represented under such a model such as reaching definitions, live variable analysis, upward exposed used, available expressions, reaching copies, very busy expressions etc. All of these compiler style data flow analyses can be modeled under a more specialised framework built on top of the monotone framework. We can represent these data flow problems with more specific transfer functions and join functions. Bugwise implements its data flow analysis using this specialised model where the transfer function in a forwards analysis is defined as:

$$out[B] = gen[B] \cup (in[B] - kill[B])$$

The gen and kill sets are defined for each basic block and are specific to the data flow problem being solved. For example, reaching definitions and upward exposed uses are two data flow problems which use the same transfer function, but the gen and kill sets for each problem will be different. The transfer function is then seen as the application of set operators including union and difference.

In a backwards analysis, the transfer function is defined as:

$$in[B] = gen[B] \cup (out[B] - kill[B])$$

The join function in the data flow problems being examined are either union or intersection depending on the type of data flow problem and applied over the predecessors or successors depending on the direction of the analysis.

The reason why we represent data flow problems in this manner is that we can apply the transfer function and join function very quickly. A typical optimisation that most compilers use is to represent each data set with a bit vector. Bit vector union, intersection and difference can be applied very efficiently in terms of computational time and this means that a solver is able to reach a fixed point faster and more efficiently.

We have experimented with using bit vector representations, but in our current system we find that a sparse representation operates more efficiently.

C. *Reaching Definitions*

A reaching definition is a definition of a variable that reaches a program point without being redefined in between. Reaching definitions are the canonical use-def analysis and are used in many further types of analysis and optimisations in use by optimising compilers and specifically Bugwise. A program point may have multiple reaching definitions for a particular variable because there may be several program paths that have a definition that reaches the program point. An example of a program and some reaching definitions is shown in Fig. 2.

Reaching definitions can be defined in terms of the data flow analysis framework described in the previous section. Reaching definitions is a forward analysis. The transfer function is as previously described. The join function is union over the basic block's predecessors. The gen set for basic block B is defined as the set of definitions that appear in B and reach the end of B. The kill set of B is defined as the set of all definitions that never reach the end of B. The out set for B is initialised as gen[B].

These data flow equations and initial state represent the reaching definitions data flow. The equations are monotonic and guaranteed to have a fixed point where a solution to the equations does not alter the state of the system if the transfer functions and join functions are subsequently applied.

D. *Upward Exposed Uses*

Upward exposed uses gathers information on the uses of a definition before it is redefined. It can be thought of as a dual to the reaching definitions.

Upward exposed uses is a backwards analysis and its transfer function is constructed accordingly. Gen[B] is defined as the set {(s,x) where s is a use of x in B and there is no definition of x between the beginning of B and s}. kill[B] is defined as the set {(s,x) where s is a use of x not in B and B contains a definition of x}. The in set of B is initialised to the null set. The join function is the union of the successors of each particular basic block.

Upwards exposed uses is a data flow analysis that is used by Bugwise to determine a number of bug related properties. Use-after-free and double free detection can be constructed in terms of this particular analysis.

E. *More Data Flow Problems*

Bugwise primarily uses reaching definitions and upward exposed uses to determine bugs in binaries yet other data flow analyses are possible. These analyses including reaching copies which we implement to determine the reach of a copy statement so that we can use this in a copy propagation optimisation described in the following section. Other analyses include available expressions which is useful to implement common subexpression elimination, and very busy expressions which is useful to perform code hoisting.

Live variable analysis is a popular analysis which determines which variables will subsequently be used before they are redefined. This is useful for use-after-free detection and also the dead code elimination optimisation. Live variable analysis is similar to upwards exposed uses, but tracks less information.

F. *An Iterative Solution*

Data flow analysis is formalised using lattice and order theory. The initial state of a basic block is the bottom of a lattice known as $\perp$.

The naive algorithm to reach a fixed point for our data flow equations is to initialise each basic block, and then iteratively apply the transfer and join functions to each node until the system stablises and the in and out sets of data reach a fixed point.

An improvement to the naive approach is to implement a work list. The work list approach notes that in a forward analysis, the successors only need to be processed if the out data in the current node changes. Thus, only in these cases are the successors added to the work list for subsequent processing.

The naive solution is correct, but slow. The work list improves this, but in Bugwise the best approach is careful traversal of the control flow graph. To improve the efficiency of the iterative solver, the order of the nodes matters. For a forwards analysis a reverse postorder traversal of the nodes is made. In a reverse postorder traversal, a node is visited before all of its successors, except when the successor is reached by a back edge. For a backwards analysis a postorder traversal of the nodes is made. In a postorder traversal, a node is visited after all of its successors.

G. IL Optimisations

Bugwise uses compiler style optimisations throughout many of its analyses. For example, to perform stack pointer inference, the code must undergo a round of optimisations for the analysis to perform effectively. Optimising the code also reduces its size which makes later analysis on the code more efficient. Bugwise implements a number of compiler optimisations including:

- Constant Folding
- Constant Propagation
- Copy Propagation
- Backward Copy Propagation
- Dead Code Elimination

These optimisations are implemented using data flow analysis which determines when such optimisations are possible. We examine some of these optimisations in the following sections.

H. Constant Propagation

Constant propagation propagates a copy assignment that consists of a constant. The motivation of this optimisation is to reduce the number of copies (assignments) and instructions in the code.

The algorithm to implement constant propagation is:

- For each instruction:
    - if all the reaching definitions of a variable have the same assignment and it is constant
    - the constant can be propagated to the variable

This algorithm makes use of the reaching definitions data flow analysis.

I. Copy Propagation

Copy propagation is similar to constant propagation except all copies are examined, not just constant copies. Again, the motivation is to reduce the number of copies and instructions in the code.

The algorithm to implement the copy propagation algorithm is as follows:

For a statement u where x is used, if:

- statement u is the only definition of x reaching u
- on every path from s to u there are no assignments to y

then we can substitute y for x in u.

An alternative framework is to use data flow analysis to determine the reaching copies:

- at each use of x where x=y is a reaching copy
    - replace x with y

This algorithm makes use of the reaching copies data flow analysis which Bugwise implements. Reaching copies is similar to reaching definitions but gathers information on the reach of a copy statement as opposed to general variables.

J. Dead Code Elimination

Dead code elimination eliminates unnecessary instructions that do not affect the semantics of the program. The algorithm to implement dead code elimination is as follows:

- For each expression
    - If the result is not live
    - then eliminate the instruction

This algorithm makes use of live variable analysis.

IV. Bug Detection

Bugwise detects bugs in binaries by applying data flow analysis on a decompiled binary. Data flow analysis is generally conservative, so in the case of bug detection, over approximation of program behaviour may occur leading to false positives. Additionally, because decompilation is not sound, program behaviour may be underapproximated leading to false negatives. Therefore, our bug detection system is unsound. However, it is still effective in detecting a reasonable number of bugs and provides benefit for analysts who use it.

In this paper we examine 3 bug classes that Bugwise can detect:

- getenv() based buffer overflows
- Use-after-free bugs
- Double free bugs

These bugs may lead to security vulnerabilities when they are found in privileged programs. We have performed scans on privileged programs and also entire Linux repositories to evaluate our system as we will explain in a later section.

A. getenv()

Environment variables are a common source of buffer oveflows in Unix-based programs. Environment variables are effectively unbounded and the Unix API call to access the environment variable, getenv(), is not bound by length. Therefore, it is reasonably common that lazy programmers copy the environment variable into a buffer without bounds checking. as is shown below.

```
char bf[128];

...
strcpy(bf,getenv("HOME"));
```

The most common method to detect these buffer overflows in closed source testing is using fuzz testing. Sharefuzz [6] is a tool that implements this. The typical approach, and that which is used by Sharefuzz, is to execute the application while monitoring uses of the getenv() API call. The environment variables passed to the API are intercepted and the return value of the environment variable is replaced with a large string to trigger potential bugs. This approach has successfully found many environment variable bugs in privileged programs. Today, it is uncommon to see these bugs in privileged code because there is generally greater awareness for this class of bug. Nevertheless, in unprivileged code, these bugs are still prevelant. A simple technique to search source code for these bugs is to use a regular expression similar to "strcpy.*getenv". This simple technique can find many vulnerabilities and when used in conjunction with searchable and public code repositories can uncover numerous instances of the buffer overflow. We can use Bugwise to determine common instances of this bug by using data flow analysis to detect that the return value of getenv() is passed directly to a string copy or any other unbounded copy. The advantage of our approach compared to using a search string, is that our system works when the buffer overflow and retrieval of the environment variable spans multiple lines.

To detect if an strcpy or strcat has a buffer overflow caused by copying the results of getenv() we use the following algorithm:

```
• For each getenv()
    o if return value is live
    o and it's the reaching definition to the 2nd
      argument to strcpy() or strcat()
    o then warn
```

This algorithm incurs false positives when getenv() is called prior to the copy and a bounds check is performed. Generally however, when getenv() is passed to a n unbounded copy, a bug is likely to be present.

B. *Use-after-free*

A use-after-free bug occurs when a pointer has been deallocated and the memory of the original allocation is accessed without any subsequent reallocation. This bug occurs when a pointer has been freed and the pointer is then accessed without it being redefined. This makes it possible to detect this class of bug using data flow analysis.

```
void f(int x)
{
        int *p = malloc(10);
        dowork(p);
        free(p);
        if (x)
                p[0] = 1;
}
```

These types of bugs can be exploitable. If the access to the deallocate memory is a read, then an information disclosure may occur and if the pointer access is a write, then it may be possible to gain execution control.

The algorithm to detect use-after-frees is as follows:

```
• For each free(ptr)
    o If ptr is live
    o then warn
```

C. *Double Free*

A double free is a subset of the use-after-free bug wherein the use of undefined pointer after a deallocation is a second free call. Historically, these types of bugs were once exploitable under Linux. Today, a program that performs a double free will crash due to sanity checks by the memory manager.

```
void f(int x)
{
        int *p = malloc(10);
        dowork(p);
        free(p);
        if (x)
                free(p);
}
```

The algorithm to detect double frees is as follows:

```
• For each free(ptr)
    o if an upward exposed use of ptr's
      definition is free(ptr)
    o then warn
```

V. Implementation and Results

A. *Implementation*

Bugwise is built on our system Malwise for binary and program analysis which has previously been used for malware analysis [4, 7-10]. Bugwise and Malwise consist of over 100,000 lines of C++ code. It is a modular system with a core static analysis engine and a plugable module interface where the bug detection modules are implemented. The figure below shows the configuration component for a scan implementing double free detection.

```
<ModuleGroup>
   <Name>Scan</Name>
   <Run>Code Optimsation</Run>
   <Run>Linux Arch</Run>
   <Run>Pre Decompiler Data Flow Analysis</Run>
   <Run>X86 Decompiler Data Flow Analysis</Run>
   <Run>Decompiler Data Flow Analysis</Run>
   <Run>Code Optimsation</Run>
   <Run>IRDataFlowAnalysis</Run>
   <Run>Double Free Detection</Run>
</ModuleGroup>
```

There are multiple phases to the scan as is shown. Code optimisation on the IL is performed followed by a Linux specific module to extract the entry point from _start via __libc_start_main. Then decompilation modules are applied, followed again by code optimisation to clean up the IL. Finally, data flow analysis so the double free module has access to the

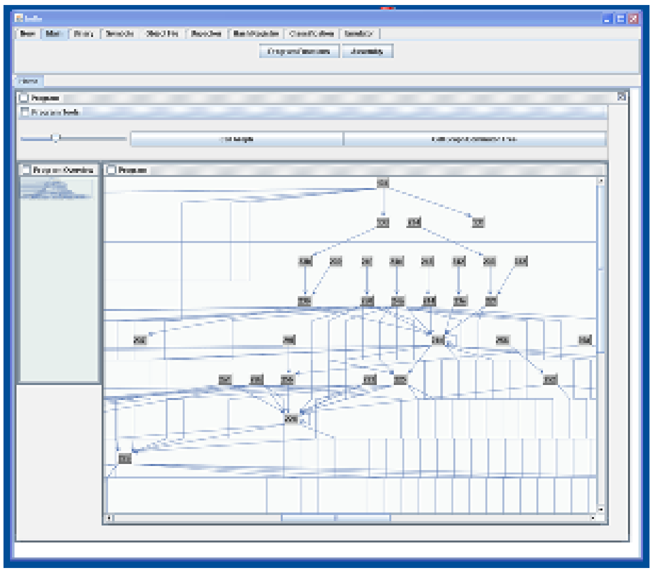
Fig. 3. Program visualisation.

reaching definitions, upwards exposed uses and other information.

To aid the debugging of Bugwise and the Malwise system it is built on, we also provide an interactive visualisation of programs via a Java GUI interface. A screenshot showing the call graph of a program is shown in Fig. 3.

B. *The Bugalyze Web Service*

Bugwise is available to use for the public as the Bugalyze.com web service. Bugwise and Bugalyze.com are implemented as a set of modules using our Malwise system. Our web services also serve another system, Simseer, that performs software similarity scoring and visualization. This service uses the same infrastructure as Malwise.

C. *Setup*

To perform an experimental evaluation and to see how many bugs we could find, we set up a machine to perform scans with Bugwise. Our test machine was an Intel 2nd generation Core-i7 with 4 physical cores, an SSD for the OS Image, a 2TB hard disk, 16G of memory, and running Ubuntu Linux 12.10 as the operating system.

D. *File Statistics*

One of the things we were interested in was the relationship between the number of bugs in a binary versus the size of the binary. The first statistical experiment we did was take every ELF binary from the Debian 7 unstable repository and sort them by size. These binaries form the basis for some of our later experiments and are a good indication on the distribution of file sizes for binaries that Bugwise is likely to work with.

The results of the analysis are charted in Fig. 4. What is evident from the chart is that the sizes of the ELF binaries grows logarithmically but has outliers. This tells us that if Bugwise scales linearly according to the number of procedures in a program, then we will have non linear growth in the time

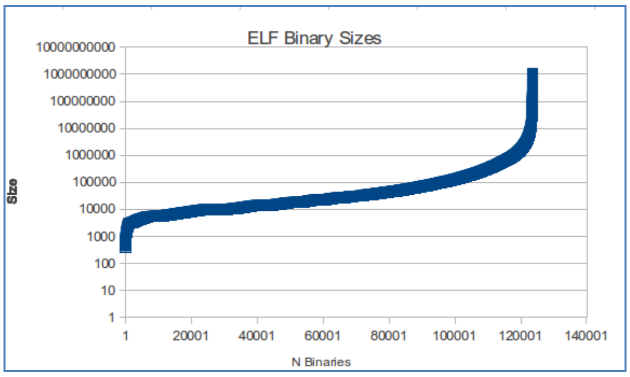

Fig. 4. Elf binary sizes.

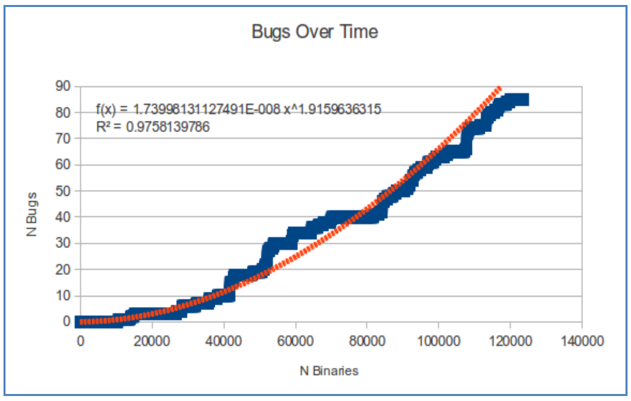

Fig. 5. Bugs over time.

that it scans a binary as we scan through an entire Linux distribution sorting the binaries by size.

E. *use-after-free and double free*

In our first main experiment to evaluate Bugwise on finding bugs and vulnerabilities we used the all the Debian 6 set-user-id and set-group-id binaries available in the repository. Each one of these binaries is privileged and a vulnerability in one may lead to privilege escalation on the local machine. Double frees are unlikely to lead to privilege escalation, however use-after-frees that are not double frees may be indicative of an exploitable memory corruption or access violation.

We ran Bugwise and enabled the use-after-free and double free bug detection modules using both intraprocedural and interprocedural analysis. Double frees are a subset of use-after-frees so we should detect all the double frees in the use-after-free detection and the use-after-free bug count should always be equal to or greater than the double free bug count.

We investigated the double free report in the xonix SGID games binary and identify the location of the double free as shown in Fig. 6. It is evident from this code that the double free only occurs on an error path when the high score file cannot be opened. This type of error can be triggered by a malicious user who opens a large number of file descriptors such that the maximum allowable is reached. The next attempt to open a file will then fail and the double free will be triggered.

```
memset(score_rec[i].login, 0, 11);
strncpy(score_rec[i].login, pw->pw_name, 10);
memset(score_rec[i].full, 0, 65);
strncpy(score_rec[i].full, fullname, 64);
score_rec[i].tstamp = time(NULL);
free(fullname);
if((high=freopen(PATH_HIGHSCORE, "w",high))==NULL) {
   fprintf(stderr, "xonix: cannot reopen high score
file\n");
   free(fullname);
   gameover_pending = 0;
   return;
}
```

Fig. 6. Double free in xonix game.

These results show that Bugwise can successfully detect double frees and use-after-frees.

F. getenv()

For our next experiment we looked at the getenv() bug detection module. For this experiment we downloaded every ELF object from every package in the Debian 7 unstable repository. This amounted to over 123,000 ELF binaries. Our system could not scan 30,450 of those binaries due to inability to parse specific ELF object types. The intraprocedural scan took less than one week to run on our test machine.

Bugwise reported 85 possible buffer overflows in 47 packages. The reported packages are shown in Table 1. This result demonstrates Bugwise is effective at detecting getenv() based buffer overflows with a limited number of false positives. A security analyst would benefit greatly from reports such as these.

G. getenv() statistics

The first getenv() statistic we looked at was to determine if larger binaries lead to more bugs. This is a common assumption auditors make when looking for bugs and vulnerabilities. We charted the cumulative number of bugs our getenv() buffer overflow detection identified as we scanned larger binaries in ascending order. What we expect is that if binary size does not matter, the growth of the bugs should be linear - that is, as the same number of binaries are scanned, the number of bugs increases at a constant rate. If binary size is relevant, then the growth of the line should be increasing at a non-linear rate. Regression testing on our data shows that the the growth is non linear. However, it is only marginally different to a linear growth. This tells us that binary size as a slight impact on this particular bug class and is almost negligent. We believe that this negligible linear growth is directly related to the bug class in question. If we examined other bug classes such as generic buffer overflows, we would expect that binary size affects the number of bugs.

For our next statistic, we had a hypothesis that bugs tend to cluster. That is, developers working on code tend to implement

Table 1. Bugs via getenv().

4digits	Ptop
acedb-other-belvu	recordmydesktop
acedb-other-dotter	rlplot
bvi	sapphire
comgt	sc
csmash	scm
elvis-tiny	sgrep
fvwm	slurm-llnl-slurmdbd
garmin-ant-downloader	statserial
gcin	stopmotion
gexec	supertransball2
gmorgan	theorur
gopher	twpsk
gsoko	Udo
gstm	vnc4server
hime	Wily
le-dico-de-rene-cougnenc	wmpinboard
libreoffice-dev	wmppp.app
libxgks-dev	xboing
lie	xemacs21-bin
Lpe	xjdic
mp3rename	xmotd
mpich-mpd-bin	
open-cobol	
Procmail	

the same things incorrectly in nearby locations. We decided to test what the likelihood of a getenv() bug occurring in another binary in the same package given one binary in the package was already vulnerable.

The probability of a binary being reported vulnerability was 0.00067. The probability of a package being reported vulnerable was 0.00255. The conditional probability of a 2nd vulnerability being present given that one binary in the package is vulnerable was 0.52380. This is a very interesting result and shows that bugs cluster in packages. It is more prudent to look for other bugs in the same package if the objective is to find as many bugs as quickly as possible.

VI. CONCLUSION

Bugwise is a system for detecting bugs in binaries by combining traditional static analysis techniques, namely data flow analysis, with decompilation. Data flow analysis has a strong theoretical foundation and today's decompilation techniques provide functional methods to recover high level and usable information. Binary-level analysis to find bugs is in its beginnings, but it has applications in areas such as black-box penetration testing and verification of the compiler and link editor. We implemented Bugwise using our previous research for program, binary, and malware analysis and now host the Bugalyze.com web service. The results of using Bugwise show that it effectively found a number of real bugs in widespread Linux distributions. We believe systems like Bugwise open the door to industrial and useful applications when source code is not available, yet assurance is still required in that software.

REFERENCES

[1] C. Cifuentes, "Reverse compilation techniques," Queensland University of Technology, 1994.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Reading, MA: Addison-Wesley, 1986.

[3] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *USENIX Security Symposium*, 2004, pp. 18-18.

[4] S. Cesare and X. Yang, "Wire -- A Formal Intermediate Language for Binary Analysis," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*, 2012, pp. 515-524.

[5] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Los Angeles, California, 1977, pp. 238-252.

[6] D. Aitel, "Sharefuzz," ed, 2004.

[7] S. Cesare and Y. Xiang, "Classification of Malware Using Structured Control Flow," in *8th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2010)*, 2010.

[8] S. Cesare and Y. Xiang, "A Fast Flowgraph Based Classification System for Packed and Polymorphic Malware on the Endhost," in *IEEE 24th International Conference on Advanced Information Networking and Application (AINA 2010)*, 2010.

[9] S. Cesare and Y. Xiang, "Malware Variant Detection Using Similarity Search over Sets of Control Flow Graphs," in *IEEE Trustcom*, 2011.

[10] S. Cesare, Y. Xiang, and W. Zhou, "Malwise -- An Effective and Efficient Classification System for Packed and Polymorphic Malware," *Computers, IEEE Transactions on,* vol. PP, pp. 1-1, 2012.