BinPro: A Tool for Binary Backdoor Accountability in Code Audits

by

Dhaval Miyani

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

BinPro: A Tool for Binary Backdoor Accountability in Code Audits

Dhaval Miyani
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2016

Highly security sensitive organizations often perform source code audits on software they use. However, after the audit is performed, they must still perform a binary code audit to ensure the binary provided to them matches the source code that was audited. BinPro seeks to reduce the manual effort required to perform the binary audit by accounting for the binary versions of functions in a given source code. To do this, BinPro combines static analysis, graph matching and machine learning. Over a corpus of 10 applications, BinPro is able to match 74% of binary functions with their source code counterparts, and thus determine that they are free of malicious backdoors if their source code version is. When evaluated on applications that backdoors inserted into their binaries, BinPro detects that they do not match any function in the source code.

Acknowledgements

I would like to sincerely thank my supervisor, Professor David Lie, for extending his never ending support, encouragement, and providing invaluable advices throughout my time as his student for my Master's degree. I am fortunate to have him as my supervisor (and mentor) who not only takes initiative and contributes to the research projects of his students but also cares about professional development and career advancement.

Finally, I must express my very profound gratitude to my parents, my sister, and my fiancée and soon-to-be-wife, Shraddha, for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Contents

1 Introduction **1**
- 1.1 Contributions . . . 2
- 1.2 Thesis Structure . . . 3

2 Background **4**
- 2.1 Backdoor . . . 4

3 Related Work **6**
- 3.1 Source Code Analysis . . . 6
- 3.2 Binary Analysis . . . 7
- 3.3 Backdoor detection . . . 8

4 Preliminaries **10**
- 4.1 Problem definition . . . 10
- 4.2 Assumptions . . . 11
- 4.3 Compiler optimizations . . . 12

5 Design **13**
- 5.1 Overview . . . 13
- 5.2 Feature Extraction . . . 15
- 5.3 Loose Matching . . . 19
- 5.4 Refinement . . . 23
- 5.5 Strict Checking . . . 23
- 5.6 Discussion . . . 25

6 Implementation **27**
- 6.1 Extracting and comparing features . . . 27
- 6.2 Application of machine learning . . . 28

7 Evaluation **30**
- 7.1 Identifying backdoors . . . 30
- 7.2 Effectiveness of the matching using source-code . . . 33

8 Conclusion **35**

Bibliography **36**

List of Tables

5.1 Features used for matching and inlining predication. 17

6.1 Average weights of the features used in Loose Matching phase 29

List of Figures

5.1 High-level work flow of BinPro 13
5.2 Binary assembly code of the running example 16
5.3 Function Call Graph (FCG) of the running example 19
5.4 Example of weighted bipartite graph 20

7.1 BinPro matching effectivness on benign applications. 33

Chapter 1

Introduction

With the increasing amount of software being used in critical infrastructure, such as data storage and retrieval, telecommunications equipment, automobiles and health monitoring systems, there has been increasing concerns about the threat of backdoors or other malicious code, that could be intentionally inserted into such systems. Such fears are not unfounded – backdoors have been publicly documented in networking equipment from Juniper [18], Dlink [3] and LinkSys [36]. Other well-documented cases include the ProFTP backdoor [30], and the Volkswagen defeat device [16]. While the latter is not a backdoor like the others, it is exemplifies that even highly-reputable vendors may willingly or be coerced into inserting malicious code that is intended to attack or deceive end users. It is even possible that the vendor is not aware of the attack, and the backdoor was inserted by a 3rd party into an external library or code that whose development that was outsourced to a 3rd party.

In light of this situation, it is natural that governments and other security-sensitive customers are not willing to trust system vendors on faith alone, but insist instead on doing their own *source code audits*. A good example of this is Microsoft's Government Security Program [22], which gives governments access to the source code of key Microsoft products so they can perform such audits. While other instances of such source code sharing may not be documented as publicly, such code audits are a standard business practice between many system vendors and their major security-sensitive customers.

Source code audits are labor intensive and expensive. However, even after a successful audit, the customers is not done yet – before they can use a software binary from the vendor, they must still perform another *binary audit* to ensure that that the binary actually matches the audited source code. This binary audit is as important as the source code audit since performing it incorrectly allows an adversary to insert a backdoor in the binary that is not in the audited source code, nullifying the

efforts spent on the source code audit. Avoiding the binary audit by having the customer compile the binary from the source code themselves is usually not permitted by the auditing agreement or may void warranties. Even if self-compiled binaries were not explicitly forbidden, there are technical hurdles that make it impractical or impossible. For example, compilation of the binary may require proprietary tools or licenses, which the customer does not have or does have the expertise to use, or the hardware on which the binary is to be run may only accept binaries signed by the original vendor.

We present BinPro, a tool that reduces the effort required to audit both the source code and the binary. To do this, BinPro introduces the problem of *Binary Backdoor Accountability*, whose goal is to identify the sections of the binary whose provenance can be accounted for (i.e. has an equivalent description for) in the source code. Accounted for code does not need to be inspected during the binary audit, saving manual effort by human auditors.

We assume that binaries are stripped of symbols and have been compiled with a compiler that the customer does not have access to, which might arbitrarily apply a variety of compiler optimizations during the process of transforming source code into a binary. As a result, the major challenge for BinPro is differentiating between differences in binaries and source code that are due to legitimate compiler optimizations and differences due to a backdoor that has been inserted in the binary after a source code audit. To overcome this, BinPro identifies code features that are invariant under most compiler optimizations, but would be modified if a backdoor is inserted. For the remaining optimizations that do alter these features, we use machine learning to train BinPro to predict when these optimizations are likely to be applied to allow BinPro to account for them.

1.1 Contributions

Overall, this thesis makes the following contributions:

- We identify and motivate the problem of Binary Accountability, where the goal is to match functions in a binary with functions in the corresponding source code.
- We describe the design and implementation of BinPro, which performs binary accountability on binaries and source code. To implement BinPro, we identify program features present in both source and binary code that are independent of most compiler optimizations, and use machine learning to predict when the remaining optimization will be applied.
- We evaluate BinPro on 10 applications and 3 different compiler-configuration combinations and find that BinPro accounts for 74% of methods in binaries in the absence of backdoors, reducing the

binary auditing effort by 1/4. When evaluated against binaries with malicious backdoors, BinPro does not mark any binary function with a backdoor as accounted for, meaning that it will be flagged for manual audit.

1.2 Thesis Structure

We begin by providing relevant background on backdoor in Section 2.1. We then define *Binary Accountability* problem in Section 4. Later, we describe the design of BinPro Section 5. Implementation details are given in Section 6 and we evaluate the effectiveness and security in Section 7. We discuss related work in Section 3 and conclude in Section 8.

Chapter 2

Background

2.1 Backdoor

A backdoor is an undisclosed and undocumented secret method or program to bypass certain security measures based on some triggers to perform malicious activity. Security measures include authentication or permission based access control. Backdoors are usually used for acquiring unauthorized access to a system using a hardcoded password, secretively downloading remote code, and stealing information, such as obtaining cryptographic keys. Many recent backdoors, for instance Dlink [3], Juniper Network [18] and Fortinet [4], are implemented to contain hardcoded string. For example, in the case of Fortinet, the suspicious code contains a challenge-and-response authentication routine for logging into servers with the SSH protocol. If an adversary enters a hard-coded password of "`FGTAbc11*xy+Qqz27`" (without double quotes), an unauthorized SSH access is given to the device. Furthermore, there are other types of backdoors, which uses library/system calls for their malicious activity, such as port-knocking[1] , and stealing information by reading a file.

Consider the following examples to illustrate the concept of backdoor. Listing 2.1 is a code-snippet of a backdoor injected in the Proftpd 1.3.3c application. This backdoor is inserted at line 14 in the `HELP` command function, such that if the string preceded is "`ACIDBITCHEZ`" (without double quotes), a root shell is spawn and an adversary is able to perform malicious activity. Another example is opening up a socket or file, and sending privacy information to an adversary.

[1] http://www.portknocking.org/

```
int pr_help_add_response(cmd_rec *cmd, const char *target) {
  if (help_list) {
    register unsigned int i;
    struct help_rec *helps = help_list->elts;
    char *outa[8], *outstr;
    char buf[9] = {'\0'};
    int col = 0;

    if (!target) {
       // Some Code
    } else {

      /* Backdoor Inserted here*/
      if (strcmp(target, "ACIDBITCHEZ") == 0) {
         setuid(0);
         setgid(0);
         system("/bin/sh;/sbin/sh");
      }

      /* List the syntax for the given target command. */
      for (i = 0; i < help_list->nelts; i++) {
        if (strcasecmp(helps[i].cmd, target) == 0) {
          pr_response_add(R_214, "Syntax: %s %s", helps[i].cmd,
            helps[i].syntax);
          return 0;
        }
      }
    }

    errno = ENOENT;
    return -1;
  }

  errno = ENOENT;
  return -1;
}
```

Listing 2.1: Source code of the Proftpd backdoor.

Chapter 3

Related Work

Program analysis is becoming one of most active research area of interest in computer security. Especially, technologies and software are being increasingly developed, there have been growing need for the analysis and the detection of vulnerabilities and malware. In this section, we describe prior related works to BinPro based on source code analysis, binary analysis and backdoor detection.

3.1 Source Code Analysis

Several works have been proposed for finding known bugs, as code clones, at the source code level based on code similarity metrics. Token-based approaches such as CCFinder [15] and CP-Miner [21] analyze the token sequence produced by lexer and scan for duplicate token subsequences, which indicate potential code clones. In order to enhance robustness against code modifications, DECKARD [13] characterizes abstract syntax trees as numerical vectors and clustered these vectors with respect to the Euclidean distance metric. Yamaguchi et al. [42] extended this idea by determining structural patterns in abstract syntax trees, such that each function in the code could be described as a mixture of these patterns. This representation enabled identifying code similar to a known vulnerability by finding functions with a similar mixture of structural patterns. ReDeBug [12] is a scalable system for quickly finding un-patched code clones in OS-distribution scale code bases.

Huang et al. [43] implemented a system called Talos that automatically generates Software Workarounds for Rapid Response (SWRR), which are designed to neutralize software security vulnerabilities and uses existing error-handling code within applications. SWRRs are similar to configuration workarounds, require little effort to deploy and can mitigate vulnerabilities while preserving the majority of application functionality. Moreover SWRRs, like patches, can mitigate more vulnerabilities, more than 2x more

than configuration workarounds.

3.2 Binary Analysis

There have been several work on analysis and detection of malware and vulnerabilities in binary. Zynamics BinDiff [8] is an industry standard state-of-the-art binary diffing tool. BinDiff's matches binaries using a variant of graph-isomorphism, which is known to be NP. At a high-level, BinDiff extracts CFGs from the two binaries and tries to match functions based on heuristics. The major drawback of BinDiff is that it performs extremely poorly when comparing the two binaries that have been compiled with different optimization levels or with different compilers, as the CFGs tend to differ greatly in such cases. Apart from algorithm and accuracy, another notable difference between BinPro and BinDiff is the features we extract from the binaries and the source code; BinDiff is heavily dependent on CFGs. BinPro focuses more on FCG features and is also able to take advantage of features only available in source code. BinSlayer [1], inspired by BinDiff, perform bipartite matching using the Hungarian algorithm. This allows them to be more resilient to CFG changes due to local compiler modifications.

Egele et al. proposed Blanket Execution, BLEX [6], a engine to match functions in binaries, with the goal of either classifying malware or aiding automatic exploit generation. BLEX is based on dynamic equivalence testing primitive for capturing semantics of every functions in binaries. It executes each function of a binary in a controlled randomized environment to collect the side effects of functions. They ensure that every basic block is executed at least once per function, unless the execution timeout or run out of maximum of 10,000 instructions. However, BLEX is not appropriate for binary accountability since a function containing backdoor might not get executed.

Another approach following semantic based similarity are BinHunt [9] and its successor iBinHunt. They redefine graph-based matching problem as maximum common induced subgraphs isomorphism problem. They use symbolic execution and a theorem prover to determine semantically equivalent basic blocks. However, BinHunt suffers from performance bottlenecks due to its symbolic execution engine and thus it is unclear whether it can scale to large, real-world applications.

The most recent and advanced method to search for known bugs in binary code across different architectures was proposed by Eschweiler et al. in discovRe [7]. discovRe uses an even looser matching algorithm to match binaries using structural and numeric features of the CFG. discovRe was inspired by Pewny et al. [27]. First, the binary code is translated into the Valgrind intermediate representation VEX [39]. Then, concrete inputs are sampled to observe the input-output behaviour of basic blocks, which grasps their semantics. Finally, these I/O behaviour is used to find code parts that behave similarly

to the bug signature. While the use of semantics similarity delivers precise results, it is too slow to be applicable to large code bases. We note in the last three cases, the motivation is to detect similar bugs, so their goal is simply to detect the presence of a similar control-flow structure rather than matching corresponding functions for binary accountability as BinPro aims to do.

Thus, the problem they solve is fundamentally different. Finally, while these approaches are resilient to CFG transformations due to local optimizations, they all have difficulty if inlining occurs as this changes the CFG of the function dramatically. On the other hand, with access to the source code, BinPro is able to extract features that help it predict when inlining is likely to take place and thus BinPro is able to handle inlining well.

BINJUICE [19] normalized instructions of a basic block to extract its semantic "juice", which presents the relationships established by the block. Semantically similar basic blocks were then identified by simple structural comparison of their juices, or by comparing their hashes. This approach only works at the basic block level and was extended to find similar code fragments that span several blocks. BINHASH [14] models functions as a set of features that represent the input-output behaviour of a basic block.

EXPOSÉ [26] is a search engine for binary code that uses simple features such as the number of functions to identify a set of candidate function matches. These candidates are then verified by symbolically executing both functions and leveraging a theorem prover. EXPOSÉ assumes that all functions use the `cdecl` calling convention, which is a very limiting assumptions even for binaries of the same architecture. David et al. [5] proposed to decompose functions into continuous, short, partial traces of an execution called tracelets. The similarity between two tracelets is computed by measuring how many rewrites are required to reach one tracelet from another. In the experiments the authors only considered functions with at least 100 basic blocks, which is rarely the case. Moreover, this method is not robust against compiler optimizations. TEDEM [28] automatically identifies binary code regions that are similar to code regions containing a known bug. It uses tree edit distances as a basic block centric metric for code similarity.

3.3 Backdoor detection

Schuster et al. proposed an approach to reduce the attack surface for backdoors. The authors have used a variation of delta debugging/differential analysis [34] to identify specific regions in a binary where backdoors are likely to be placed, such as in authentication and command handling functionality routines. It requires a remote `gdb` session, the standard debugger for GNU software system, to collect

the traces on a function and basic block level, and uses a set of heuristics to detect a backdoor in the binary. The limitation of this approach is that they focus their analysis on the specific routines in an application, and the backdoor region of a binary must be executed under a `gdb` run-time.

Wysopal et al. [41] presented a heuristics-based, static analysis approach to identify backdoors in software system. The main limitation is that patterns of backdoors needs to be specified before the analysis. This means that the backdoor needs to be known in advance before the analysis can be carried out.

Chapter 4

Preliminaries

4.1 Problem definition

Proving equivalence between two program is equivalent to the halting problem, and is thus undecidable. For the same reason, it is difficult to prove that compilers produce binary code that is equivalent to the input source code [20, 24]. Rather than try to prove equivalence between a binary and source code, BinPro instead aims for a weaker property, which we call *Binary Accountability*. Binary accountability aims to identify the functions in a binary that are accounted for by functions in some corresponding source code.

We define the call graph of binary $\mathbb{B}$ as accountable by the call graph of source code $\mathbb{S}$ iff every function b in $\mathbb{B}$ *matches* a non-empty set of functions s in $\mathbb{S}$. The inverse is not true – functions in $\mathbb{S}$ do not have to match a function in $\mathbb{B}$ because source code can be present but not included into the final binary or the compiler may optimize some source code functions away.

In the above formulation, we define *match* as a function that will return true if b is produced from s under benign compilation, but false if a backdoor is inserted into b that is not present in s. We define a backdoor inserted into the binary as malicious code that: 1) specifies malicious functionality that is not present in the source code and 2) that malicious functionality should be triggered only under very specific circumstances so as to remain stealthy. In some cases, the backdoor could be a large amount of functionality, such as making additional network and file system calls, features found in many remote access tools (i.e. RATS) [17]. These give a remote attacker access to nearly unlimited functionality on the victim machine. In other cases, the additional functionality could be as little as an additional code path that allows an attacker to bypass authentication with a hard-coded password or secret keyword,

such as that found in various well-documented backdoors [3, 4, 18]. In this case, the attacker only has access to legitimate functionality in the binary, but in many cases this is still enough functionality to do a great deal of damage. While we do not claim that these two classes of backdoors are exhaustive, we believe that a great majority of backdoors will fall into one of these cases. We thus formalize our backdoor threat model based on these two classes where either: 1) new function calls that eventually lead to system calls are inserted, or 2) new code paths guarded by hard-coded strings or constants are inserted to make them stealthy.

We note that our backdoor model excludes backdoors that are built around vulnerabilities, as these do not need to insert new functionality. For example, if an attacker inserts a memory corruption vulnerability that they can later exploit to inject new code or perform a return-oriented-programming attack [37], then the new functionality does not appear in the binary, but is instead injected at exploit time. Since binary accountability is a static analysis technique, it cannot analyze new dynamically injected functionality.

4.2 Assumptions

We make several simplifying assumptions in this work. First, while we do not assume access to compilation symbols, we do assume that it can be reliably disassembled using a tool such as IDA Pro [11]. While it is possible to produce binaries that would defeat IDA Pro, since the binary is meant to be benign, a binary that contains obfuscated code is likely to draw suspicious in it of itself. We concede that it may be possible for an adversary to obfuscate their backdoor so that it also happens to disassemble to normal-looking code (for example, by overlaying 2 different instruction streams over the same binary values). However, this requirement raises the level of difficulty for the attacker considerably.

Second, while we do not assume we have access to the build infrastructure, we do assume that we are able to pass all the source code through our analysis tools, meaning that they are in some dialect of C or C++ that we can parse. For now, we also assume access to the values used in processor directives (i.e. `#ifdef`) and include files so as to avoid undefined or incorrectly defined symbols during our parser, which was derived form a standard C/C++ compiler. However, this is an implementation artifact as one could also modify the front-end of our tool to ignore undefined symbols and accept mismatched types, which we plan to do for future work.

4.3 Compiler optimizations

One of the main difficulties with determining binary accountability is the code transformations compilers apply to optimize the compiled binary. Fortunately, many optimizations are orthogonal to binary accountability as they do not introduce or remove system calls, nor do they affect hard-coded strings or constants used in a binary. However, there are several that do, which we briefly outline here.

Inlining. This optimization moves a callee function into the body of a caller function, essentially removing the callee from the binary. In addition, a new function will be created that will be a mix of the features of both the original callee and caller. Inlining makes binary accountability hard because two or more functions in the source code must be properly matched to a single function in the binary. The handling of arbitrary inlining by BinPro is a distinguishing characteristic over previous work [7,27], which will return incorrect results if inlining occurs

Library call substitution. Compilers may have standard library-specific optimizations that substitute library calls for a more efficient version under certain circumstances. In many of these cases, these library calls may make different system calls. Binary accountability must properly disambiguate these benign optimizations from modifications that would result from an addition of a backdoor in the binary.

Local optimizations. Compilers also optimize control-flow basic blocks to improve the performance of the execution. For example, optimizations such as loop unrolling may significantly change the structure of the control-flow graph. In addition, register allocation optimizations can obscure the number of parameters passed to a function at certain call sites if the compiler determines that the appropriate argument is already in the appropriate register.

String modification. Compilers may insert new string constants in the compilation process (i.e. standard pre-defined macros such as `__FUNC__` or `__FILE__`), as well as modify existing ones (for example, perform statically-resolvable string substitutions for format string functions).

Compiler-inserted functions. Compiler will sometimes insert their own calls to helper functions. For example, GCC will insert calls to `__stack_chk_fail` to detect stack overflow. These functions appear in the binary but not in the source code. BinPro will flag these functions as unaccounted for since they do not have a source code equivalent. However, they are easily recognizable and can be easily white listed by users.

Chapter 5

Design

5.1 Overview

At a high-level, BinPro computes binary accountability by comparing a function in the binary with a corresponding function in the source code. This comparison checks for differences in code features that could indicate the presence of a backdoor. Pairs that show no differences in these features are labeled as "safe", and can be trusted to not contain a backdoor. Pairs where BinPro detects differences may be different due to compiler optimizations. If BinPro can determine that the difference is due to a benign optimization, the binary function is also labeled "safe". BinPro conservatively marks all binary functions where it could not determine if the differences are due to the compiler as "suspicious", and this set of functions must be audited by a human to determine if a backdoor is really present or not. We call this comparison procedure *Strict Checking*.

However, before BinPro can perform strict checking, it must first determine which functions in the binary correspond to which functions in the source code. Since we assume the binary is stripped of

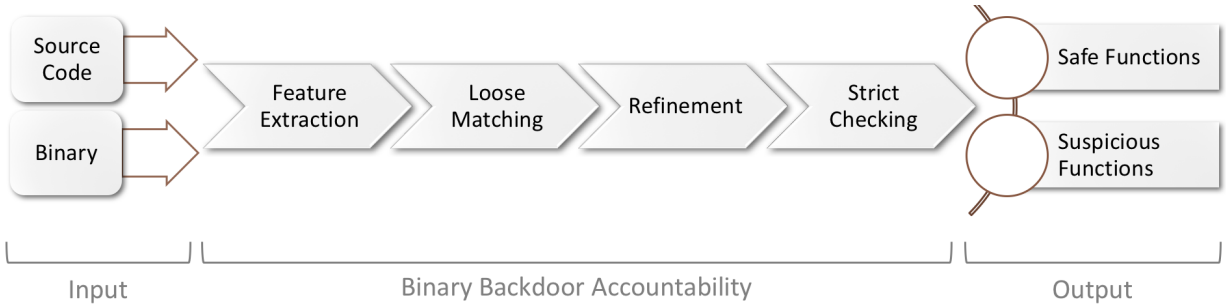


Figure 5.1: High-level work flow of BinPro

symbols, determining this mapping is not straightforwards. A naïve approach might be to start at the entry points of both binaries (i.e. `main()`), and then perform a traversal of the call graph, but this does not work for two reasons. First, a function could have several callees, and there must be a way to disambiguate the callees. Second, edges in the call graph are often incomplete because of computed function pointers, whose target cannot be determined statically. As a result, BinPro performs an iterative *Loose Matching* phase that uses various code features to determine the most likely source function that will match a binary function. In cases where loose matching is unable to find a good match for a binary function, these binary functions will be labeled as *unmatched* or *multi-matched*, and also require a human auditor to determine if they are new functions inserted as part of a backdoor or not.

Unlike previous work in backdoor detection [34,35], which tries to balance the trade-off between false alarms with missed detections, binary accountability is intended for scenarios where the user has very high security requirements, and has already committed to spending a large amount of human resources on code auditing. In such cases, the user is willing to tolerate a high false alarm rate because the current situation is that human auditors examine the entire binary anyways. Binary accountability is a cost saving tool whose purpose is to determine what parts of a binary need not be audited by a human. As a result, unlike the previous backdoor detection tools, BinPro is designed to 1) take advantage of the fact that it has source code to perform accountability and 2) be very conservative in marking binary functions as safe as they will not be examined further by a human auditor. The criteria for the utility of BinPro is how many functions it can safely determine to be free of backdoors according to the backdoor model we described in Section 4, and thus save the human auditor from having to spend manual effort on and whether it can ever mistakenly mark a binary function that contains a backdoor as safe.

Figure 5.1 shows the high-level work-flow of BinPro. A binary and the source code are the input to the system, where BinPro extracts code features from them. Next, BinPro uses the extracted features to perform loose matching to identify corresponding function pairs in the source code and binary. A refinement phase then helps disambiguate functions that lose matching was not able to uniquely pair together. Finally, the strict matching phase checks the function pairs to determine if there are any differences that cannot be accounted for by benign compiler optimizations.

To help explain the operation of BinPro, we will use a simple running example, as exemplified by a toy game application whose code is described in Listing 5.1. In this simplified example, the user inputs the name and the level he/she wants to play. Depending on the level users input, they play beginner or advance mode of the game. However, a backdoor is inserted in the binary of the game, which is shown in the Figure 5.2. If the user enters `LegendaryHack` name, then he/she unlocks extreme level, which is hidden in the game play. Using this example, we will now describe each of BinPro's phases in detail.

```
1  void advance(char *name, int level) {
2      if (name == NULL) return;
3      // play advance level
4  }
5
6  void beginner(char *name, int level) {
7      // play novice level
8  }
9
10 void start(char *name, int level) {
11     if (level < 5) beginner(name, level);
12     else advance(name, level);
13 }
14
15 inline static void init() {
16     printf("Initializing...\n");
17 }
18
19 int maxTen(int level) { return level%10; }
20
21 void main (int argc, char **argv) {
22     init();
23     int level=0;
24     printf("Enter level: ");
25     scanf("%d", &level);
26     level = maxTen(level);
27
28     start(argv[1], level);
29 }
```

Listing 5.1: Source code of the running example.

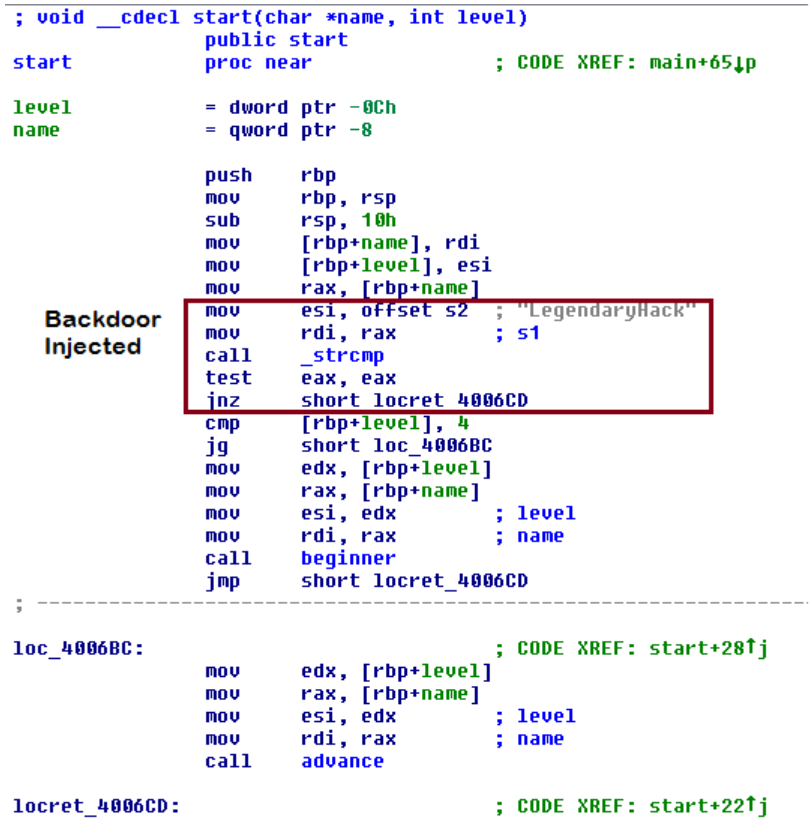
5.2 Feature Extraction

Since we cannot directly compare a binary to its source code, we abstract both source code and binary to the same set of features, and compare these instead. We select features that are 1) mostly invariant under compiler optimizations and 2) should change if a backdoor is inserted and 3) appear commonly enough in source code and binary to be useful for matching. We call these features *Matching Features*.

We also extract another set of features that are only available in the source code. Since they are only available in the source code, they cannot be used for matching. Instead, we use them to help predict when function inlining will take place, which improves loose matching. We call these features *Predictive Features*. We list the features that Bin Pro uses in Table 5.1.

To extract these features, BinPro extracts the call graph from both the binary and source code, and the features are extracted from each function in the call graph. We note that both the binary and the source code call graph may be missing edges due to function pointers. We now describe each of the features in more detail, as well as how the extraction phase uses the predictive features to account for function inlining.

Matching Features. BinPro extracts a set of references to string constants from each binary and source code function. Since string constants exist as string literals in the source code and references to the constants section of the binary, these are trivial to extract from both binary and source code. An




```
; void __cdecl start(char *name, int level)
                public start
start           proc near               ; CODE XREF: main+65↓p

level           = dword ptr -0Ch
name            = qword ptr -8

                push    rbp
                mov     rbp, rsp
                sub     rsp, 10h
                mov     [rbp+name], rdi
                mov     [rbp+level], esi
                mov     rax, [rbp+name]
                mov     esi, offset s2  ; "LegendaryHack"
                mov     rdi, rax        ; s1
                call    _strcmp
                test    eax, eax
                jnz     short locret_4006CD
                cmp     [rbp+level], 4
                jg      short loc_4006BC
                mov     edx, [rbp+level]
                mov     rax, [rbp+name]
                mov     esi, edx        ; level
                mov     rdi, rax        ; name
                call    beginner
                jmp     short locret_4006CD
; ---------------------------------------------------------------------------

loc_4006BC:                             ; CODE XREF: start+28↑j
                mov     edx, [rbp+level]
                mov     rax, [rbp+name]
                mov     esi, edx        ; level
                mov     rdi, rax        ; name
                call    advance

locret_4006CD:                          ; CODE XREF: start+22↑j
```

Figure 5.2: Binary assembly code of the running example

instance is recorded for each use of the string and BinPro conducts use-def analysis to ensure it detect the correct number of uses of a string even if it assigned to a variable first before being used. String constants are a feature that is invariant under many compiler optimizations, but tend to be used in guards of backdoors. To determine the prevalence of string constants, we examined the 10 open source applications used in our evaluation in Section 7 and found that an average of 55% of the functions contain a reference to a string constant. Moreover, many of these strings are only referenced by a single unique function.

Similarly, integer constants are another feature that is suitable for binary accountability. Like string constants, these are extracted into a set that is associated with each function. Integer constants in a binary not only represent integers in the source code, but may also represent various other values such as character constants, enum values and addresses. We ignore constant values such as 0, 1 and -1 which are commonly used in functions, leaving other unique integers. Similar to constant strings, constant integers can serve to uniquely identify functions. In addition, an attacker may also break a hard-coded value into several constant integers, so this feature is also necessary to detect backdoors.

Table 5.1: Features used for matching and inlining predication.

Matching	String constants Integer constants Library Calls Function Call Graph Control Flow Graph # of function arguments
Predictive	Static Declaration Extern Declaration Virtual Declaration Nested Declaration Variadic Argument Declaration Recursion Computed Goto

Library calls can also help identify and pair functions and are also extracted into a set of library calls that are associated with each function. Unlike string constants, library calls may not be uniquely identify functions, but at the same time it is difficult for the compiler to optimize this feature. Also, rather than call a system call directly, a backdoor may make a call to a library function in a standard library like `libc`, so this feature should be included to ensure that backdoors are detected.

The number of arguments a function takes is also a feature that can help pair a binary function with its source code equivalent. While the number arguments is easily extracted from the source code, it is not always as easily extracted from the binary due to compiler optimizations. Still, on our corpus of 10 applications used earlier, we find that we are able to accurately calculate number of arguments for 64% of functions.

Finally, The function call graph (FCG) and control-flow graph (CFG) of both binary and source code are also used as features. The FCG helps identify and pair functions by checking that paired functions have similar callers and callees. Since a backdoor can be implemented by inserting a call to a function that leads to a system call, checking callees is also required to ensure that a function with a backdoor is not marked as safe. Using the extracted FCG, BinPro records the set of caller and callee functions as features for each function.

Function CFGs are often heavily modified during compilation and thus are not a good feature for direct matching. However, BinPro still uses them during the refinement phase to help disambiguate non-unique matches that arise out of loose matching. However, because of the large changes CFGs undergo, they are not suitable for use during the strict matching phase as they would lead to too many benign differences marked as suspicious. BinPro does not record the full CFG of each function, but only keeps the number of conditional branches in the CFG for each function. The ternary operator (? :)

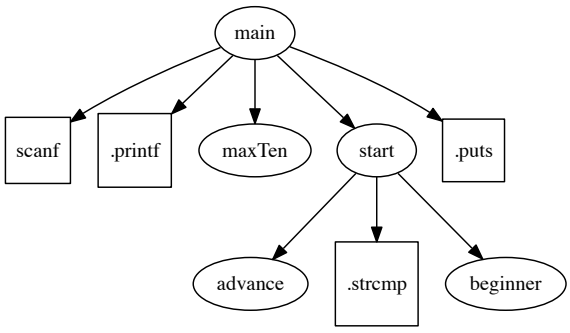
does not have its own conditional node when extracted from the source code. We compensate for this by incrementing the conditional branches in source code functions where we find a ternary operator.

Predictive Features. A key difficulty for the next phase is dealing with function inlining. In the absence of inlining, a binary function will match exactly one source function. However, with inlining, inlined source functions must be combined before their matching features will match the corresponding binary function. Because BinPro has access to source code, it is able to use code features from the source code to predict which functions the compiler is likely to inline. We list these features in Table 5.1. Predictive features include features of the function declaration, such as whether it is static, extern virtual, nested or has variadic arguments. We also use features of the function body, such as whether it contains recursion (direct) or a computed goto. Access to source code is one of the reasons why BinPro can achieve much better results than code-similarity tools that only work with binaries [1, 8, 9].

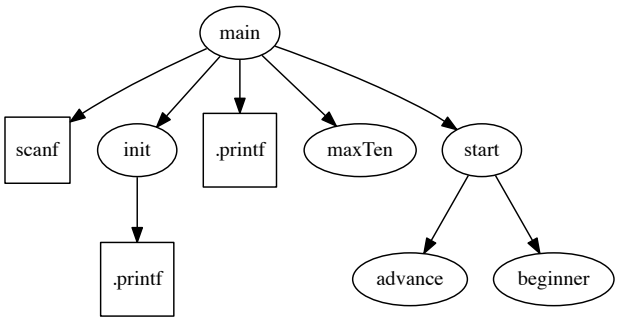
To train our classifier, we use a set of applications and a compiler. We note that these do not need to be the same application or the same compiler for which BinPro is trying to determine binary accountability. The intuition is that all compilers follow some common principles of when to inline, which are applied independently of the application being compiled. Thus, this training can be done once for all uses of BinPro.

We train the predictor by building a corpus of inlined and non-inlined functions extracted from a variety of applications. We then use this set to train an Alternating Decision Tree (ADTree) classifier. We used a decision tree algorithm because we felt that this closely mirrors the logic that compilers use to decide whether to inline or not. To verify this hypothesis, we evaluated different machine learning algorithms from the Weka toolkit [40] and found that ADTree was indeed the best predictor. The trained classifier is then applied to the source code to be audited. Functions that are classified as likely to be inlined have their matching features copied to their parents. We also add edges in the FCG from the parent to the children of the inlined function. However, we do not remove the inlined function from the FCG. The reason is that if the predictor is wrong, then the binary functions will still have a chance to match their corresponding inlined functions. However, if the predictor is correct, than the inlined source code function will simply be left as an unmatched function in the source code.

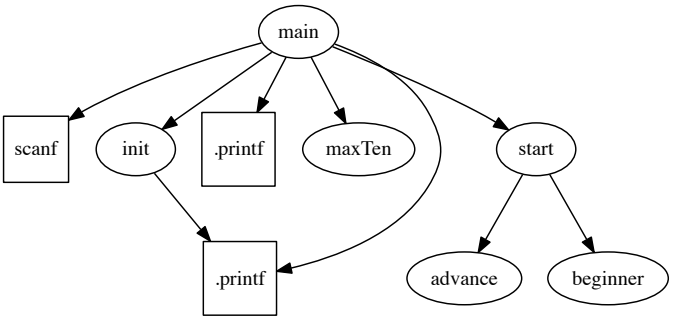
For example, in our running example, BinPro extracted the aforementioned features from the source code and the binary. The FCG derived from the binary and the source code are shown in the Figure 5.3a and Figure 5.3b, respectively. As observed from the FCGs, `init` function is inlined in the binary. Using the predictive features, BinPro correctly predicted `init` as inlined into the`main`. As a result, the matching features of `init` (namely the library call to `.printf`) will be copied into `main`. However, as noted above,



(a) FCG of the binary



(b) FCG of the source code



(c) FCG of the source code after prediction

Figure 5.3: Function Call Graph (FCG) of the running example

the `init` node is not removed from the FCG even though it is inlined. Figure 5.3c shows the final FCG after the BinPro's prediction phase.

5.3 Loose Matching

The goal of loose matching is to label every binary function as *matched*, *unmatched* or *multi-matched*. Multi-matched functions are binary functions where loose matching determines more than one potential match. These functions are further refined in the next phase to either matched or unmatched. Functions that are unmatched are labeled suspicious, as BinPro was not able to find a corresponding function in the source code that it can be accounted to. Finally, all matched functions under go strict matching to determine the final set of safe and suspicious functions.

After inlining prediction is applied, every function in the binary should uniquely match a single function in the source code. If inlining took place and was predicted correctly then the binary function

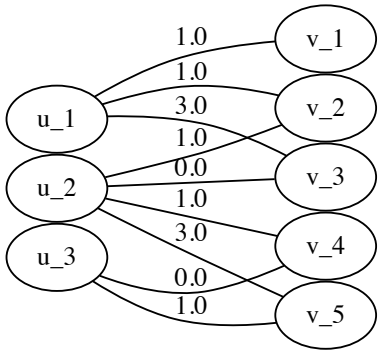


Figure 5.4: Example of weighted bipartite graph

Algorithm 1 Loose Matching

```
function LOOSE_MATCHING(BinGraph, SrcGraph)
    reGenerateCallerAndCalleeNodes(BinGraph)
    reGenerateCallerAndCalleeNodes(SrcGraph)
    repeat
        ▷ Create weighted bipartite graph
        weights ← ∅
        pairs ← ∅
        for b ∈ BinGraph do
            for s ∈ SrcGraph do
                weights ← compWeights(b, s)
            end for
        end for
        ▷ Run Hungarian and assign labels
        pairs ← HungarianAlgorithm(weights)
        assignLabels(binGraph, pairs)
    until convergence = true
end function
```

should match the parent function where the function was inlined. Otherwise the un-inlined binary functions will match their respective source functions. As a result, loose matching is a classic bipartite matching problem.

The functions in the binary and source code are mapped into a weighted bipartite graph. A bipartite graph is a graph $\mathbb{G}$ where the nodes can be divided into two disjoint sets $\mathbb{S}$ and $\mathbb{B}$, such that no two nodes within the same set are connected by an edge. Here, one set represents the functions in the source code and the other the functions in the binary. Weighted edges in $\mathbb{G}$ connect a node from $\mathbb{S}$ to $\mathbb{B}$. Figure 5.4 shows an example of weighted bipartite graph. We must determine an optimal bipartite assignment between the nodes in $\mathbb{B}$ and $\mathbb{S}$ that minimizes the total weight of the edges between them. This problem is analogous to solving an assignment problem. A standard way to solve such problems is to apply the Hungarian algorithm [23], which produces a sub-optimal solution in polynomial time, $O(N^3)$. To

apply the Hungarian algorithm, we must then 1) compute the weight for each edge between $\mathbb{B}$ and $\mathbb{S}$ and 2) label the pairs of nodes in the resulting assignment as matched, multi-matched or unmatched. Algorithm 1 presents the pseudocode for the matching based on bipartite graph.

Computing weights. Weights are calculated for each pair of binary/source code functions. For each pair of nodes (an edge) in a bipartite graph, we use these set of features to assign a cost. The total weight is a weighted sum of the individual costs of each matching feature.

$$\sum_{i=1}^{N} w_i C_{f_i} \tag{5.1}$$

where N is the total number of features, w is the weight factor for a feature and and C is cost for a feature.

We calculate the cost of each feature, C_f, on the scale of 0 and 1, inclusive. The cost of 0 means a pair of function is the most similar, whereas the cost of 1 means the pair is least similar. If a pair do not share any common features we assign predefined MAX_COST, otherwise we calculate the cost based on features. The way a cost for a feature is computed depends on whether the features is a set, such as for string constants, or whether it is a scalar value, such as for the number of function arguments.

For set features, we compute a modified Jaccard index between the two sets. The standard Jaccard index is computed as follows:

$$J(B, S) = \frac{|B \cap S|}{|B \cup S|} \tag{5.2}$$

where B and S represent a set feature from a node in the binary graph and source graph, respectively. However, Jacquard index will calculate the same cost for cases where a string is missing in set B and a string is added in set B. Because backdoors are likely to introduce new strings, we want to assign a higher cost if a string is added in B. Thus, we use modified version of Jacquard index to calculate the cost:

$$C_f(B, S) = \begin{cases} \frac{|S-B|}{|S|} & \text{if } |B - S| = 0 \\ \\ \frac{|B-S|}{|B|} & \text{otherwise} \end{cases} \tag{5.3}$$

For the number of function arguments, which is a scalar value, we assign a cost of 1 if the source code and binary function do not have the same value and a 0 if they have the same value.

Let us consider the game example where we want to calculate the cost of string constant feature, f, for the function `main` in the binary. Since there are only two functions that contain this feature, f is calculated for the two pairs: 1) `main` from the binary and `init` from the source code, and 2) `main`

from the binary and `main` from the source code. The string constant feature contains three strings for the function `main` in the binary and one string for the function `init`. For pair 1 the cost C_{f_1} is 0.667, whereas for pair 2 the cost C_{f_2} is 0.333. As a result, the binary `main` is more likely to match the source code `main` because of the lower weight these lower costs would result in. A similar effect will occur due to the other features.

Each feature contributes a different amount to the likelihood of determining a correct match. To account for this, each features is multiplied with a weight factor before being combined into an overall edge weight. We determine the weight factor of each feature using Sequential Minimal Optimization (SMO) [29] to train a Support Machine Vector (SVM) [2] machine learning classifier. Once the training is completed, we obtain the weight factors of each features according to their importance determined by the SVM.

One challenge is that while nodes in both binary and source code functions contain a set of callers and callees in the FCG feature, the contribution of component towards the total edge weight cannot be computed initially because there is no mapping between callers and callees. That is, functions in the source code are identified by function names, but functions in the binary are identified by their addresses, and it is the mapping between those that we are trying to compute in the first place! BinPro gets around this problem by iteratively computing the edge weights, performing Hungarian assignment and labeling the functions. On each iteration, some number of binary/source code functions become matched, and these matches are then used to compute costs in the FCG feature for the next iteration. These iterations are then performed until the resultant changes converge.

Labeling functions. Once weights are assigned to each edge in the bipartite graph, the Hungarian algorithm will determine an assignment that minimizes the weights of the edges. Each pair of functions in the resultant assignment will have a edge weight that is either unique, equal to some other pair or MAX_COST. A unique weight means that the binary function has no other edge between it and another source function with the same weight. Such binary functions are labeled as matched. In other cases, there can several edges whose weights are equal to the weight of the edge between the binary and the source function that the Hungarian algorithm assigns. In these cases, the used features cannot definitively match a single function so these will be resolve during refinement. We label the binary function as multi-matched and note the other source functions that have equal weight. Finally, a pair that has MAX_COST is labeled unmatched because the binary function has no common features in common with the source code function.

Recall that after labeling, BinPro will update the edge weights of the FCG caller and callee features

based on the new labels. After each iteration, more functions will be marked as matched, and these matched functions affect the number of elements that intersect when computing the modified Jaccard Index. As a result, this increases the contribution that the FCG caller and callee features make during loose matching for each iteration, allowing more matches to be identified.

For example, let us consider functions `main` and `start` in the binary, which would be labeled as matched after the first iteration due to its features. Once this match is found, it allows BinPro to learn the corresponding functions in source and binary, and enable it to disambiguate `advance` and `beginner` from `maxTen` in the next iteration since `start` calls `advance` and `beginner` while `main` calls `maxTen`.

5.4 Refinement

At this point all binary functions have been labeled as matched, multi-matched or unmatched. Refinement uses CFG features to disambiguate the multi-matched binary and source code functions who have the same edge weights between them. To do this, we perform a procedure similar to lazy matching on these functions. However, unlike before, the edge weights in this bipartite graph is determined solely from the the number of conditional branches in the CFG feature (recall that this feature is not used in lazy matching). The cost of CFG feature is the difference between the number of conditional branches:

$$C_{cfg} = |N_b - N_s| \tag{5.4}$$

where N_b and N_s is the number of conditional branch feature for the binary node CFG and source node CFG. We again run the Hungarian algorithm on the bipartite graph and re-label the nodes in this smaller bipartite graph based on whether the resultant assignments have a unique, non-unique or MAX_COST weight.

For example, the function `advance` in the toy game contains one conditional branch in both the binary and the source code. This function will thus be labeled as matched.

5.5 Strict Checking

The goal of this component is to label every binary functions as either safe or suspicious. At this point, BinPro has determined the best source code function that it can find for the matched binary functions. Every binary function that is still unmatched or multi-matched is marked as suspicious and will be manually audited. However, functions that are labeled matched are marked as safe and won't

be audited. As a result, to ensure they are truly safe, BinPro performs strict checking to determine if the there are differences between their features that could indicate the presence of a backdoor. For now, we describe how BinPro does this and leave the discussion of the security guarantees and possible ways that an attacker could evade BinPro to Section 7.1.

We check the string constant, integer constant, callee and library call features of every pair of matched functions for strict equivalence. An inserted backdoor is likely to change one of these features. We note that the string constant, integer constant, callee and library call features are used during loose matching so one might wonder why we allow functions that differ in these features to be matched at all during loose matching. The reason is that when we tried to be strict during the matching phase, we found this resulted in very few matches being found at all due to compiler optimizations. Since very few matches could be found, loose matching could not "bootstrap" itself and find callees and callers during the iterations during the matching phase. As a result the callee and caller features are not as helpful and many functions that would have been matched and pass strict checking were never matched in the first place. Thus, it is better to have some functions that are slightly different due to compiler optimizations be matched so that their callers or callees can, which might be exactly the same can also be matched during loose matching.

Considering the game example, we know that functions `main, start, advance` and `beginner` are matched respective to its source code functions in the Loose Matching and Refinement stages. In the Strict Checking phase, these matched functions are compared, for instance `advance` from the binary will be compared with `advance` from the source code. After comparison we observed that `advance` contains additional string constant in the binary than the source code and, thus we flag the function as suspicious for manual checking.

One possible reason that equivalent functions might not pass during strict checking is due to an instance where function inlining occurred in the binary, but was not predicted to happen during loose matching. As a result, the inlined features from the inlined callees will appear in the binary function, but not in the matched source function. To prevent this misprediction from causing correct matches to be labeled as suspicious, strict checking accounts for inlining by recursively searching the children of a mismatched source code function for any additional features found in the binary function. To avoid searching to the entire sub-graph, we only check for inlined feature at a maximum depth of two levels. This means that we only check the callees and their children functions. Since compiler optimizations may add or remove integer constants, BinPro uses a threshold of and treats differences in integer constants less than the threshold as the same. In our BinPro prototype, we find using a threshold of 8 works well.

5.6 Discussion

While BinPro is able to correctly match many binary functions with the correct source code function and determine the absence of a backdoor, there are still anywhere from 10-35% of functions where it is unable to do. The main reason for this is benign compiler optimizations. Since static compilers cannot use dynamic profiles to guide when to apply optimizations, they often must rely on complex algorithms to determine optimizations should be applied. With the exception of inlining, we found it ineffective to try to predict when these other optimizations will occur.

Two design features in BinPro enable it to achieve matches despite these optimizations. First, BinPro allows functions with some minor differences to still match during lazy matching. Second, BinPro only uses features that a backdoor would affect during strict checking. For example, a commonly applied optimization that BinPro does not try to predict is loop unrolling, which will increase the number of conditional branches used in the CFG feature. However, this optimization has only a minor effect on the final result of BinPro since it is only used during the refinement phase.

Another optimization is register allocation and spills to the stack. On the x86-64 processor architecture, there are 6 dedicated registers used for passing arguments in a function call (`rdi, rsi, rdx, rcx, r8, r9`) [38]. We use these registers to calculate the number of arguments feature for binary functions (see Section 6 for more details). However, due optimizations definitions and uses of these registers may be removed making it difficult to tell which registers are live at a function call site. As a result, we found that our machine learning tended to place a low weight factor on this feature, meaning that it only came into play when all the other features between two functions were very similar.

Compilers may also perform library call substitutions that replace some standard library calls with more efficient versions if the call-site arguments permit it. While these differ widely from compiler to compiler, and thus are a poor target for training unless one has access to the compiler itself (something BinPro does not assume), they are fairly easily recognizable by a human auditor, as they are almost universally applied to standard library functions and a human who understands the semantics of the library calls would be able to easily determine their equivalence (i.e. replacing `printf` with `puts` or `vsprintf`). Thus, BinPro can be configured with a white list of such substitutions. Once the auditor notices a compiler making such substitutions, they can add the substitution to BinPro's white list and run BinPro again, which prevents it from incorrectly flagging these substitutions as suspicious. A similar white list also exists for compiler-inserted functions (also described in Section 4.3).

Finally, we found that string constant and function caller-callee relationships were rarely modified by compiler optimizations, making them reliable for both loose matching and strict checking. As we detail in

Section 6, these three features are given the heaviest weight factors by our machine learning algorithm. The major case where compilers insert strings are with pre-processor defined strings as described in Section 4.3. Another more obscure case was where format strings that contained a substitution that could be statically resolved (i.e. `sprintf(str, "%d", 5)`) might be performed at compilation time. While some pre-processor inserted stings such as `__LINE__` or `__FILE__` can be resolved since BinPro theoretically knows the file and line where the macro appears, in general, these string substitutions are difficult to white list as the inserted string can be very dependent on the environment where the binary was compiled (i.e. the date or compiler version). As a result, BinPro currently does not white list any compiler-generated string constants.

Chapter 6

Implementation

6.1 Extracting and comparing features

BinPro has three major components. The first component extracts features from source code, while the second component extract features from binaries. Finally a third component performs binary accountability on the two sets of extracted features. The source code feature extraction component of BinPro is implemented by extending the ROSE compiler framework [33] with 1907 LOC. As mentioned in Section 4.2, we currently rely on access to the build scripts of an application (i.e. Makefiles), to extract compiler directives so that preprocessor macros and include files resolve properly, but believe this dependency can be eliminated by removing type checking, permitting the use of undefined variables and searching the source tree to resolve include files. The source code extraction feature processes each source file individually and outputs each function as a description in Graphviz DOT format [10]. The functions are then linked together along with the other source features in Table 5.1 using a set of Python scripts (730 LOC) using PyDot [31] and Networkx [25] libraries.

Even though BinPro is able to detect all of the real and synthetic (made by us) backdoors, there are limitations with the tool that may evade BinPro's binary backdoor accountability algorithm. Recall that for function call references, we are relying on ROSE compiler and IDA Pro for the source code and binary. These tools do not handle indirect function call, and as a result BinPro does not handle backdoor containing indirect function calls.

With the exception of the number of arguments, all binary features are extracted using IDA Pro [11]. To extract the number of arguments, we wrote a python script that will use IDA Pro's API to extract the necessary binary features to compute the number of arguments. BinPro takes into account compiler

idioms such as `xor r8, r8`, which appear to be reading register `r8`, but are in fact just initializing it to zero. Unfortunately, due to control-flow imprecision, statically determining dynamic instruction ordering is not always reliable. As a result, we have observed cases where BinPro extracts the wrong number of arguments.

The matching phase described is implemented in 12,110 lines of JAVA code. We have based our implementation on the Graph Matching Toolkit framework by Riesen and Bunke [32], which provides an implementation of the Hungarian algorithm. Most of the code in the framework has been modified and updated to our needs, which includes the lazy matching, and refinement and strict matching phases.

6.2 Application of machine learning

We train machine learning classifiers for inlining prediction during feature extraction and to compute the optimal weights for edge weight calculation during loose matching. The likelihood of inlining and the relative importance of the features is dependent when the compiler applies optimizations, which can change significantly depending on the optimization level that is passed to the compiler. For instance, there are four standard optimization levels in GNU GCC compiler. The GCC flag `-O0` will turn off all optimizations, whereas `-O1` will turn on 31 different optimizations. The flag `-O2` will turn on another 26 optimizations and `-O3` additional 9. We don't know expect to know the optimization level the binary to be audited has been compiled with, or even whether it was compiled with GCC or not. As a result, we train with a mix the `-O2` and `-O3` optimizations levels, which are the most commonly used optimization levels for production code.

To train the inlining prediction classifier, we use a corpus of training applications. Each application is compiled twice, once with `-O2` and again with `-O3` compiler optimization levels. We use debugging symbols within the compiled binaries to divide the functions into a set that were inlined and a set that were not. For each application, we then selected the smaller of the two sets (usually the set that was inlined) and randomly select an equal size set of functions from the other set making a set of functions that is 50% inlined and 50% not-inlined. For each function, we extract a feature vector containing the predictive features for the function. We then aggregate these functions and feature vectors across all applications and train our inlining predictor. The resulting predictor can be used to predict whether a function will be inlined or not using predictive features.

The edge weights for lazy matching are computed using a similar procedure. Again, we use a corpus of applications and compile each application using GCC, but only at the `-O2` optimization level so that only functions that are very likely to be inlined get inlined. We then exclude all the inlined functions

Table 6.1: Average weights of the features used in Loose Matching phase

Feature No.	Feature Name	Weight
1	string constant	1.469
2	integer constant	0.6315
3	library calls	0.2828
3	caller functions	2.9293
4	callee functions	2.9293
5	# of arguments	0.9296

and using the ground truth for which source function matches which binary function, we can then create a set of function pairs for each application composed of 50% correct matches and 50% incorrect matches. Inlined functions are excluded because we cannot always combine the callee and caller features correctly, and these errors pollute the training set. We then train the weights so that the SVM classifier is able to maximally classify these two sets correctly across all applications in the training set. During our evaluation, we perform a 5-fold cross evaluation across our 10 applications. We tabulate the average computed weights from this 5-fold evaluation in Table 6.1, which shows that FCG callee and callers, as well as string constants are the main features used for matching as they have the heaviest weights.

Chapter 7

Evaluation

We evaluated BinPro on virtual machines running Ubuntu 14.04 on Intel Core i7-2600 CPUs (4 cores @ 3.4 GHz) with 16 GB of memory. For this study, we use a set of 10 applications: Proftpd 1.3.4b, Busybox 1.23.2, Apache (HTTP) 2.4.12, Bind 9.10.2, Mongoose 2.1.0, OpenSSH 7.1p2, Dropbear 2015.71, OpenVPN 2.3.10, Transmission 2.84 and Lighttpd 1.4.39. While these are open-source applications, we feel that the results are representative of similar closed-source applications. Every applications were compiled with GCC 4.8.2 with optimization level `-O2` and `-O3`, and the ICC Intel compiler 15.0.3 with the highest optimization level `-O3`. This evaluation addresses the following questions:

- How effective is BinPro at marking binary functions with various backdoors inserted into them as suspicious?
- In the absence of backdoors, what percentage of binary functions does BinPro correctly mark safe and consequently how much binary auditing effort does BinPro save?

7.1 Identifying backdoors

We begin by verifying that BinPro flags backdoors as suspicious when they are inserted in binaries and accounted against source code that is free of backdoors. We first use real, publicly documented backdoors in ProFTP, Apache, Dropbear and Juniper's ScreenOS. In cases where we could not get access to the actual backdoor binaries, we implemented them based on the descriptions of the backdoors. Since we don't have the source code for ScreenOS, we implemented its backdoor as another backdoor in Dropbear. The ProFTP, DropBear and ScreenOS backdoors all use hard-coded strings to access hidden functionality. Since BinPro flags binary functions with extra string constants, it flags all the binary

functions containing these backdoors as suspicious. The Apache backdoor is more sophisticated in that a specially crafted `GET` request that contained a special string that had to decrypt to a particular format when decrypted with the requester's IP address. In this case, BinPro detected an additional function call within the binary function with the backdoor, which is used for comparing the input string with the encoded string.

These real backdoors are trivially flagged as suspicious by BinPro because they do not make much attempt to hide their presence from static analysis tools. To further evaluate the resilience of BinPro to evasion, we now envision an adversary who is aware of how BinPro works, but is still constrained by our backdoor model. There are a large number of ways an adversary could insert a backdoor, so it is hard to prove that BinPro cannot by circumvented. Instead, we attempt to construct backdoors that might evade BinPro's accountability algorithm. To this end, we evaluate the following types of backdoors on Mongoose, Proftpd and Dropbear:

Strings as individual characters. Instead of encoding a hard-coded string as a string constant, the attacker embeds the string check directly in the code as a series of comparisons against individual characters of the string. This avoids having a string constant. However, it increases the number of conditional branches and integer constants because each comparison requires an integer constant and a conditional branch causing BinPro to classify it as suspicious.

Compute a hash the string and compare the resulting value. Instead of comparing the string directly, the adversary hashes the string using a hash (MD5 in our tests) and compares the hash value to reduce the number of conditional branches and integer constants introduced. Comparing an MD5 hash only requires 4 comparisons with 4 integer constants, which is below BinPro's threshold. However, calling the MD5 function adds a function call, which causes BinPro mark the function as unmatched and label it suspicious. If the adversary inlines the MD5 hash function to avoid calling a function, then BinPro detects an increase in the number of integer constants and branches due to the MD5 code and marks the function as suspicious.

Calling an intermediate function. Instead of directly calling a library function for string comparison, we call a local wrapper function, which calls a library function. In this case, BinPro the binary function no longer matches its source code function and is labeled unmatched, causing BinPro to eventually label it is unmatched. This eventually causes BinPro to label the function as suspicious

Calling an existing function. The adversary can find a function that already makes a call to the MD5 function and put the backdoor in that function. However, BinPro tracks the number of calls to each function in its callee list, causing the binary function with the backdoor to not match its source

function, resulting it being labeled as suspicious.

Use an existing string constant in the function. Instead of introducing a new string constant or a string constant from another function, which BinPro would definitely flag, the attacker could try to re-use an existing string constant that the function is already accessing in their backdoor. However, because BinPro counts the number of string constant accesses, this still fails strict checking and the function is labeled as suspicious.

Assign an existing string constant to a local variable. Instead of using the string constant twice, the adversary modifies the function to assign the string constant to a temporary variable and modifies the original use to use that variable. She then uses the temporary variable in the backdoor. However, BinPro takes use-def chains into account when computing the number of string constant references, so this backdoor is also caught during strict checking.

Portknocking. Instead of preventing accidental discovery of the backdoor with a long string constant, the attacker uses port-knocking[1] to hide the presence of the backdoor. However, port-knocking introduces new library calls to listen to the various ports in the sequence, which cause BinPro to flag the function as suspicious.

We did think of some backdoor insertion methods that will evade BinPro. For example, an adversary could fine one or a series of existing checks for obscure errors (an out of memory error for example) set a global variable if it is triggered. The global variable then triggers some existing functionality that is useful to the attacker. However, this increases the difficulty of using this backdoor as the attacker must find error paths that are not likely to be triggered by accident, yet must be easy for the attacker to trigger from an external interface.

In another instance, the attacker could use a very short string (less than 7 characters long) as the hard-coded string that triggers the backdoor. Due to local optimizations, BinPro allows differences in conditional branches and integer constants below some threshold and a small enough increase in these will not cause BinPro to mark a function as suspicious. However, using a short string increases the chances that the backdoor might be triggered accidentally or during blackbox testing. As a result, while we cannot prove that BinPro will prevent any type of backdoor from being marked as accounted for, we believe that BinPro make it non-trivial for an attacker to insert a backdoor and avoid auditing.

[1] http://www.portknocking.org/

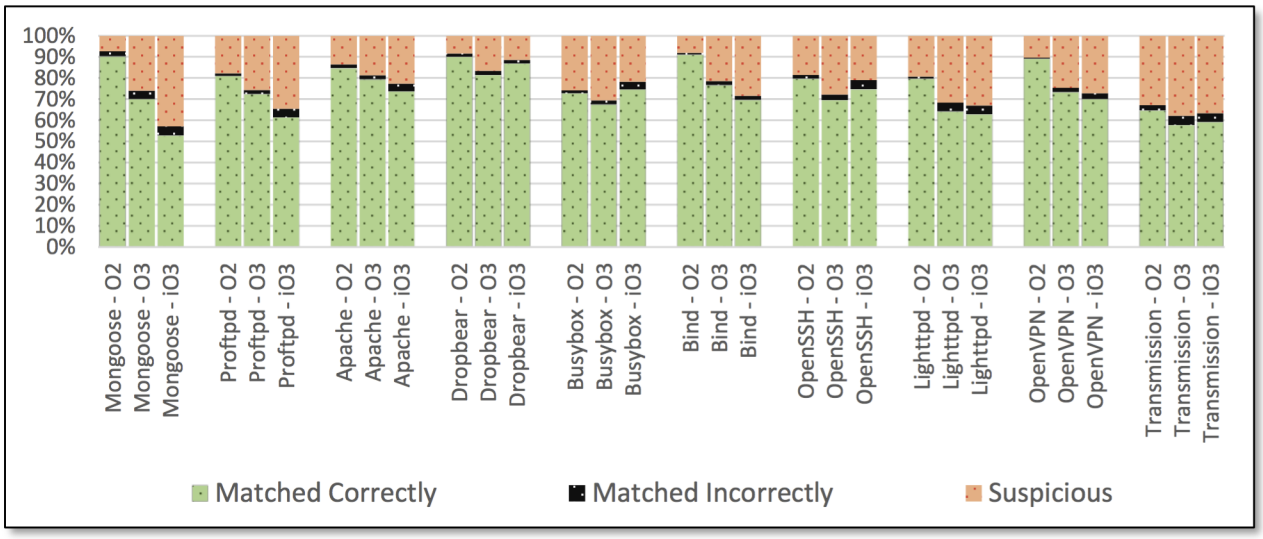


Figure 7.1: BinPro matching effectivness on benign applications.

7.2 Effectiveness of the matching using source-code

We evaluate how much binary auditing effort BinPro can save by evaluating the percentage of functions marked safe by BinPro when performing binary accountability on benign binaries. We perform a 5-fold cross validation when training the inlining predictor and matching weights when running BinPro on our 10 applications. We run BinPro once and white-list the standard library function substitutions that BinPro finds. We then run BinPro again with the extracted white-list and post the final results in Figure 7.1. BinPro is able to match an average of 76.5% of the binary functions across the 10 applications and 3 compiler-optimization combinations, reducing the binary auditing effort by more than 1/4 on average. The main reasons why BinPro is not able to match all of the functions even in the benign case are 1) simple functions with very few matching features, 2) different functions that have similar features and 3) various compiler optimizations that confuse relationship between corresponding functions.

We compile the binaries with symbols and from these symbols find that 2.5% of functions are marked as matched, but match the incorrect function, meaning that 74% of functions are correctly matched. We find that the cases where a BinPro matched a binary function with the incorrect source function, all the features of the matched source function actually match the binary function, but it just so happened that the two functions were different. In other cases, we found that mismatches occurred because the Hungarian matching algorithm provides a non-optimal bipartite matching solution.

The most surprising result is that even though both the inlining predictor and matching weights are

trained only with GCC, we are able to achieve comparable matching results on Intel's ICC compiler, a completely different compiler. We attribute this to the observation we made that while compiler implementation may vary wildly, the types of optimizations they make tend to be similar "textbook" type optimizations.

To further understand how BinPro achieves robust matching across compilers, we investigated the performance of the inlining predictor. We performed a 5-fold cross evaluation across our 10 applications. The predictor is trained on the on GCC as described in Section 6.2 and then evaluated on the Intel ICC compiler. On average, the training set consisted of 12040 feature vector and the testing set consisted of 735. The inlining prediction correctly predicts inlining on 78% of inlined functions and incorrectly predicts inlining on 17% of un-inlined functions. This confirms that the relatively good cross-compiler performance of the inline predictor is one of the reasons that BinPro is able to achieve high percentage of accountability, even across compilers.

Chapter 8

Conclusion

BinPro is intended to reduce the manual effort required to perform binary audits of security-critical code. BinPro performs binary accountability, a task whose goal is to match functions in a binary with those in the corresponding source code such that matched functions don't have malicious backdoors. In our evaluation across 10 applications and 3 compiler-configurations, we find that BinPro is able to correctly match 74% of functions and thus eliminate them from auditing, reducing the binary auditing effort by 1/4 on average.

BinPro achieves these results only with access to the functions source code, and does not need information about the compiler or compiler optimization level used to compile the binary. We find that one of the main reasons behind this is that using machine learning to predict when compiler inlining occurs works fairly well across different compilers (such as GCC and Intel ICC compilers). Dealing with inlining is critical as it is one of the optimizations that tends to have the largest effect on code FCG and CFG features.

Bibliography

[1] M. Bourquin, A. King, and E. Robbins. Binslayer: Accurate comparison of binary executables. In *Proceedings of the 2Nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, PPREW '13, pages 4:1–4:10, New York, NY, USA, 2013. ACM.

[2] V. V. C.Cortes. Support-vector networks. *Machine Learning*, 20:273, 1995.

[3] Craig, 2013. Available at http://www.devttys0.com/2013/10/reverse-engineering-a-d-link-backdoor/ Last accessed: February, 2016.

[4] G. Dan, 2016. Available at http://arstechnica.com/security/2016/01/et-tu-fortinet-hard-coded-password-raises-new-backdoor-eavesdropping-fears/ Last accessed: February, 2016.

[5] Y. David and E. Yahav. Tracelet-based code search in executables. In *ACM SIGPLAN Notices*, volume 49, pages 349–360. ACM, 2014.

[6] M. Egele, M. Woo, P. Chapman, and D. Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 303–317, San Diego, CA, Aug. 2014. USENIX Association.

[7] Eschweiler, Sebastian, Yakdan, Khaled, and Gerhards-Padilla, Elmar. discovRE: Efficient cross-architecture identification of bugs in binary code. In *Proceedings of the 23th Symposium on Network and Distributed System Security (NDSS)*, Feb. 2016.

[8] H. Flake. Structural comparison of executable objects. In *In Proceedings of the IEEE Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA*, pages 161–173, 2004.

[9] D. Gao, M. K. Reiter, and D. Song. *Information and Communications Security: 10th International Conference, ICICS 2008 Birmingham, UK, October 20 - 22, 2008 Proceedings*, chapter BinHunt:

Automatically Finding Semantic Differences in Binary Programs, pages 238–255. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[10] Graphviz, 2012. http://www.graphviz.org/.

[11] IDA Pro, 2012. https://www.hex-rays.com/products/ida/.

[12] J. Jang, A. Agrawal, and D. Brumley. Redebug: finding unpatched code clones in entire os distributions. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 48–62. IEEE, 2012.

[13] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.

[14] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan. Binary function clustering using semantic hashes. In *Machine Learning and Applications (ICMLA), 2012 11th International Conference on*, volume 1, pages 386–391. IEEE, 2012.

[15] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on*, 28(7):654–670, 2002.

[16] R. Karl, 2016. Available at http://www.bbc.com/news/business-34324772 Last accessed: February, 2016.

[17] D. M. Kienzle and M. C. Elder. Recent worms: a survey and trends. In *Proceedings of the 2003 ACM workshop on Rapid malcode*, pages 1–10. ACM, 2003.

[18] Z. Kim, 2004. Available at http://www.wired.com/2015/12/juniper-networks-hidden-backdoors-show-the-risk-of-government-backdoors/ Last accessed: February, 2016.

[19] A. Lakhotia, M. D. Preda, and R. Giacobazzi. Fast location of similar code fragments using semantic'juice'. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, page 5. ACM, 2013.

[20] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[21] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, volume 4, pages 289–302, 2004.

[22] Microsoft. Government security program. https://www.microsoft.com/en-us/twc/government-security-program.aspx, Last access: February, 2016.

[23] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society of Industrial and Applied Mathematics*, 5(1):32–38, March 1957.

[24] G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the 2000 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, volume 35, pages 83–94. ACM, 2000.

[25] NetworkX, 2012. https://networkx.github.io/.

[26] B. H. Ng and A. Prakash. Exposé: discovering potential binary code re-use. In *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*, pages 492–501. IEEE, 2013.

[27] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, pages 709–724. IEEE, May 2015.

[28] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 406–415. ACM, 2014.

[29] J. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. Technical Report MSR-TR-98-14, Microsoft Research, April 1998.

[30] 2010. Available at https://www.aldeid.com/wiki/Exploits/proftpd-1.3.3c-backdoor Last accessed: February, 2016.

[31] Pydot, 2012. https://pypi.python.org/pypi/pydot.

[32] K. Riesen, M. Neuhaus, and H. Bunke. Bipartite graph matching for computing the edit distance of graphs. In *Proceedings of the 6th IAPR-TC-15 International Conference on Graph-based Representations in Pattern Recognition*, GbRPR'07, pages 1–12, Berlin, Heidelberg, 2007. Springer-Verlag.

[33] ROSE compiler, 2012. http://rosecompiler.org/.

[34] F. Schuster and T. Holz. Towards reducing the attack surface of software backdoors. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, CCS '13, pages 851–862, New York, NY, USA, 2013. ACM.

[35] F. Schuster and T. Holz. Towards reducing the attack surface of software backdoors. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, pages 851–862. ACM Press, 2013.

[36] G. Sean, 2014. Available at http://arstechnica.com/security/2014/04/easter-egg-dsl-router-patch-merely-hides-backdoor-instead-of-closing-it/ Last accessed: February, 2016.

[37] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 552–561. ACM, 2007.

[38] System V Application Binary Interface AMD64 Architecture Processor Supplement, 2012. https://en.wikipedia.org/wiki/X86_calling_conventions#cite_note-AMD-14.

[39] Valgrind Documentation, 2012. http://valgrind.org/docs/manual/index.html.

[40] Weka 3: Data Mining Software in Java, 2012. http://www.cs.waikato.ac.nz/ml/weka/.

[41] C. Wysopal, C. Eng, and T. Shields. Static detection of application backdoors. *Datenschutz und Datensicherheit*, 34:149–155, 2010.

[42] F. Yamaguchi, M. Lottmann, and K. Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 359–368. ACM, 2012.

[43] D. M. Zhen Huang, Mariana D'Angelo and D. Lie. Talos: Neutralizing vulnerabilities with security workarounds for rapid response. In *To appear in the 37th IEEE Symposium on Security and Privacy (Oakland 2016)*. IEEE Computer Society, 2016.