12-2012

iBinHunt: Binary Hunting with Inter-Procedural Control Flow

Jiang MING
Penn State University

Meng PAN
Penn State University

Debin GAO
Singapore Management University, dbgao@smu.edu.sg

iBinHunt: Binary Hunting with Inter-Procedural Control Flow

Jiang Ming[1], Meng Pan[2], and Debin Gao[3]

[1] College of Info Sciences and Tech, Penn State University, jum310@ist.psu.edu
[2] D'Crypt Pte Ltd, pinpinmao@gmail.com
[3] School of Info Systems, Singapore Management University, dbgao@smu.edu.sg

Abstract. Techniques have been proposed to find the semantic differences between two binary programs when the source code is not available. Analyzing control flow, and in particular, intra-procedural control flow, has become an attractive technique in the latest binary diffing tools since it is more resistant to syntactic, but non-semantic, differences. However, this makes such techniques vulnerable to simple function obfuscation techniques (e.g., function inlining) attackers any malware writers could use. In this paper, we first show function obfuscation as an attack to such binary diffing techniques, and then propose iBinHunt which uses *deep taint* and automatic input generation to find semantic differences in *inter-procedural* control flows. Evaluation on comparing various versions of a `http` server and `gzip` shows that iBinHunt not only is capable of comparing inter-procedural control flows of two programs, but offers substantially better accuracy and efficiency in binary diffing.
Keywords: binary diffing, semantic difference, taint analysis

1 Introduction

Binary diffing tools for finding semantic differences between two programs have many security applications, e.g., automatically finding security vulnerabilities in a binary program given its patched version [17], large-scale malware indexing with function-call graphs [20], automatically adapting trained anomaly detectors to software patches [24], profile reuse in application development [33], etc. However, binary diffing is difficult due to different register allocation, semantically equivalent instruction replacement, and other program obfuscation techniques which make semantically equivalent programs syntactically different [17].

One of the latest solutions in binary diffing for finding semantic differences is to find similarity/difference in *control flow structure* rather than binary instructions [14, 12, 17, 20]. Such tools have the advantage of being resistant to semantically equivalent instruction replacements and other program obfuscation techniques, and therefore are more suitable in security analysis in which programs (potentially malware) are usually intentionally produced to make analysis difficult. An interesting aspect we analyze in this paper is whether such analysis should be based on *inter-procedural* control flow or *intra-procedural* control flow.

Most previous work [14, 12, 17, 20] focus on the intra-procedural control flow.[4] There is a good reason for this choice as the control flow comparison usually involves maximum common subgraph isomorphism, an NP-complete problem [18]. Working with all basic blocks in an inter-procedural control flow graph (ICFG) would require manipulation of graphs with thousands or tens of thousands of nodes, where finding a graph isomorphism becomes impractical. Working with basic blocks in an intra-procedural control flow graph (CFG), instead, is practical as the number of nodes does not usually go beyond hundreds. However, comparing the control flow structure of basic blocks in each function is vulnerable to function obfuscation techniques (e.g., function inlining) that could be used in producing the binary programs under analysis. This is a serious problem as applying some function obfuscation, e.g., function inlining, is extremely easy.

In this paper, we first demonstrate the attack of function obfuscation on binary diffing tools that compare intra-procedural control flow. We then propose a new binary diffing technique called *iBinHunt* that is resistant to such an attack. iBinHunt discards all function boundary information and compares the inter-procedural control flow of binary programs. It uses *deep taint*, a novel dynamic taint analysis technique that assigns different taint tags to various parts of the program input and traces the propagation of these taint tags to reduce the number of candidates of basic block matching. With deep taint, the set of matching candidates of each basic block changes from the set of all basic blocks in a program (in the order of thousands or tens of thousands) to just a few basic blocks on a particular execution trace with the same taint tag. To increase the coverage of execution traces on basic blocks, iBinHunt automatically generates program inputs that traverse different execute paths for the deep taint analysis.

We implemented iBinHunt and used it to compare various versions of a `http` server and `gzip`. Results show that iBinHunt finds semantic differences by analyzing the inter-procedural control flows with better accuracy, and is capable of comparing binary programs with relatively large differences, an improvement over previous techniques which are only shown to work on programs with small changes. We also show that iBinHunt is more efficient and faster in finding basic block matchings than previous techniques by a factor of two.

2 Existing binary diffing tools and function obfuscation

We focus on binary diffing tools for finding semantic differences instead of syntactic differences. Semantic differences refer to differences in functionality (i.e., input-output behavior), whereas syntactic differences refer to those in instructions [17]. Therefore we do not consider binary diffing tools that base its analysis on the binary instructions (bsdiff, bspatch, xdelta, JDiff, etc.), because they are more vulnerable to different register allocation, basic block re-ordering, functionally equivalent instruction(s), and other instruction obfuscation techniques.

[4] Some of them zoom in to do intra-procedural control flow analysis first, and subsequently zoom out for inter-procedural control flow analysis where each procedure is represented as a simple node with details ignored.

2.1 Existing binary diffing tools based on control-flow structure

To find semantic differences between two binaries, some latest binary diffing techniques [14, 12, 27, 17] base their comparison on intra-procedural control-flow structure. BinDiff [14] and its extension [12] use some heuristics (e.g., graphs with the same number of basic blocks, edges, and caller nodes) to test if two graphs or basic blocks are similar. BinHunt [17] compares basic blocks by symbolic execution and theorem proving, and then compares intra-procedural control-flow graphs to find the matchings between basic blocks. Call graphs are then compared to find matchings between functions. DarunGrim2 [21, 11] relies heavily on function boundary information due to its simplicity. For basic blocks in every function, DarunGrim2 first generates a fingerprint to abstract the instruction sequences and then uses that as a key to a hash table, from which fingerprint matching is performed to find differences in the two functions. Intra-procedural control-flow graphs have also been used frequently in malware clustering and classification [20, 2, 5, 22] because it's more resilient to instruction-level obfuscations. SMIT [20] searches for the most similar malware samples by finding a nearest-neighbor in malware's function-call graph database. Kruegel et al. [22] present an approach based on the analysis of a worm's intra-procedural control-flow graph to identify structural similarities between different worm mutations.

2.2 Function obfuscation

Binary diffing tools based on control-flow structure are more resistant to different register allocation, basic block re-ordering, functionally equivalent instruction(s), and other instruction-level obfuscation techniques. However, most of them rely heavily on function boundary information from the binary, i.e., they analyze the *intra-procedural* control-flow structure of each function. We believe that this is mainly due to efficiency of the graph comparison techniques used. The graph comparison problem (and the subgraph isomorphism problem) is NP-complete. Existing algorithms for subgraph isomorphism are efficient only in processing small graphs [23, 28, 17]. Appendix A shows the number of basic blocks in different functions in a typical server program binary, which suggests that graph isomorphism is practical when analyzing intra-procedural control flows.

However, function boundary information is not reliable due to well-studied *function transformation obfuscation* techniques [9], which include

- **Inlining functions:** a function call to f is replaced with the body of f while f itself is removed;
- **Outlining functions:** a new function f is created by extracting a sequence of statements into f and replacing them with a function call to f;
- **Cloning functions:** copies of the same function are created (with different names) to make them appear as different functions;
- **Interleaving functions:** various function bodies are merged into one function f, while calls to these functions are replaced by calls to f.

Here we focus on function inlining and outlining because they have a large impact on graph isomorphism as discussed in Appendix B.

3 Diffing binary programs with inter-procedural control-flow graphs

In Section 2.2, we discuss the function obfuscation attacks which existing binary diffing tools based on intra-procedural control-flow analysis cannot deal with. A natural solution to such attacks is to find repetitions of code sequences and combine them into one subroutine (to combat function inlining and cloning), and to flatten the hierarchical structure created by functions and to simply treat function calls as execution jumps (to combat function outlining and interleaving). After this there is only one graph left for each binary program containing all basic blocks and the corresponding control flows, and this graph is essentially the inter-procedural control-flow graph (ICFG).

However, such a simple solution has disadvantages in both accuracy and efficiency. Each basic block in one binary program will have a large number of candidates of basic block matchings in the other binary program. Even if all these candidates are examined, there could be multiple ones that are semantically similar that originally come from non-matching functions. However, since function information is ignored, all these basic blocks are good candidates and may make the result inaccurate. We also need to work on graph isomorphism of two graphs with large number of nodes. We tried this with BinHunt [17], one of the latest and most sophisticated binary diffing tools with graph isomorphism, and found that after working for 6 hours on basic block comparison with a desktop computer with a Core2 Duo CPU of 3.0 GHz and RAM of 4 GB on a server program `thttpd`, only 7% of the possible mappings had been compared.

3.1 Overview of iBinHunt

iBinHunt reduces the number of candidates of basic block matchings with a novel technique called *deep taint*. Taint analysis is to dynamically trace data from untrustworthy sources to monitor basic blocks in a program that process such data [26, 7, 35, 31, 16, 13]. We monitor the execution of the two binary programs under a common input and use taint analysis to record all basic blocks involved in the processing of the input. This reduces the number of candidates of basic block matching from all basic blocks in the binary to those tainted.

iBinHunt goes one step further to assign different taint tags to various parts of the input, a method we call *deep taint*. Deep taint differentiates various parts of the input by assigning them different taint tags, and monitors propagation of different taint tags to basic blocks on a dynamic trace. Only basic blocks from two binary programs that are marked with the same taint tags are considered matching candidates. This further reduces the number of candidates of basic block matchings by a factor of up to 74% in our experiments.

Deep taint and the taint tags help reduce the number of matching candidates. However, only a small number of basic blocks are on the trace of the processing of a single input, and we need to find the matching of a large number of basic blocks (if not all) to make the graph isomorphism efficient. iBinHunt increases the

coverage of execution traces on tainted basic blocks by automatically generating inputs that result in different execution traces in the binary program, a technique inspired by recent advances in white-box fuzz testing [19]. We first record the execution trace of a seeding input, and then symbolically replay the recorded trace and collect constraints of the input that lead to the recorded trace. The collected constraints are then negated and solved with a constraint solver to generate a new input, which will result in a different execution trace due to the negated constraint. A large number of inputs can be generated in this way, making more and more basic blocks tainted with different taint tags.

Next, we present the details of deep taint and automatic input generation.

3.2 Deep taint for basic block comparison

Previous taint analysis treats taint sources as *streams*, e.g., byte streams from keyboard, effectively tainting all input bytes with a single taint tag. Basic blocks processing different parts of such input will therefore be tainted with the same taint tag. In iBinHunt, we differentiate these basic blocks if they process different parts of the input. For example, basic blocks that process the `version` field of an `http` request should never match with basic blocks that process the `host` field of the same `http` request. Differentiating these basic blocks will reduce the number of candidates of basic block matchings.

Table 1 shows an example of the different taint tags assigned to various parts of an `http` request. Each unique taint tag corresponds to a particular bit in a binary number that allows *disjunction* manipulation. Deep taint works on the *protocol level* with a finer granularity such that various protocol fields are monitored with different taint tags. The process of locating different fields of the program input can be automated with a protocol analyzer [10, 34, 4].

Input	`Get`	`index.html`	`HTTP/1.1`	`.`
Field	Method	URL	Version	Host
Taint tags	0001	0010	0100	1000

Table 1. Program input and its taint tags

Multiple taint tags for a basic block By monitoring the dynamic execution of an input, we can see the propagation of different taint tags to basic blocks in the program. Note that a basic block may appear multiple times on a dynamic execution trace due to loops. Such a basic block may record the same taint tag in the execution (when it processes the same part of the input in a loop) or different taint tags (when it processes different parts of the input).

Figure 1 shows an example of this in our experiment with `thttpd-2.25`. The highlighted instructions in the source code is located inside a `for` loop, which executes multiple times in the processing of an input and records multiple taint

tags. The dynamic execution trace we obtained recorded four different taint tags for a basic block `BB_10088`, which corresponds to the highlighted instructions in the source code. We take the disjunction of these tags to obtain the final taint representation for the corresponding basic block.

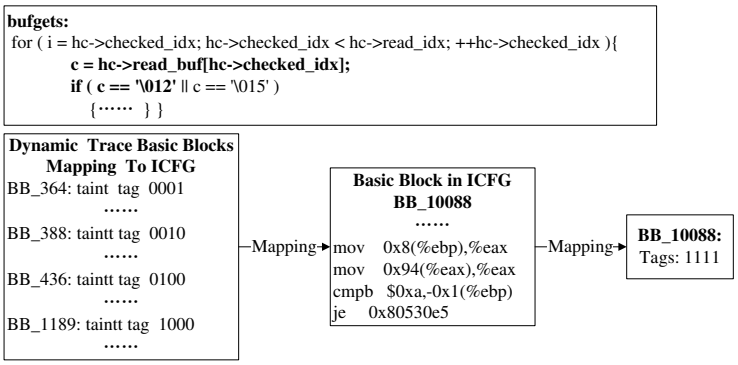


Fig. 1. Multiple taint tags

Basic block comparison As mentioned in Section 3.1, basic blocks from the two binary programs that have the same taint representation will be candidates for matching. We compare these candidate basic blocks by applying the same algorithm as BinHunt [17], in which symbolic execution is used to represent the outputs of a basic block in terms of its symbolic inputs, and a theorem prover is used to test if the outputs from the two basic blocks are semantically equivalent. Although this basic block comparison might take relatively long time to converge (due to the use of a theorem prover), the number of comparisons is limited to the small number of blocks with the same taint representation, and therefore iBinHunt is more efficient (see Section 4 for our evaluation results).

Basic block matching There are two other groups of blocks we need to consider for finding matched blocks. One group consists of blocks that are not semantically equivalent but have the same taint representation. They could very likely represent the differences between the two programs that iBinHunt is trying to locate. Another group consists of blocks that are not tainted but are on the dynamic execution trace. These blocks are not tainted due to various reasons, including limitations of taint analysis to avoid taint explosion [6, 29], not directly processing program inputs (e.g., signal processing), etc. However, they are also very likely to match with one another as they are on the dynamic trace of processing the same input. Appendix C shows an example of these two groups of blocks in `thttpd-2.19` and `thttpd-2.25`.

One way of dealing with these two groups of blocks is to define a matching strength for basic block comparison, and consider two blocks matching when the

matching strength exceeds certain threshold; an approach used in BinHunt [17]. We do not use this approach because 1) iBinHunt emphasizes using control-flow structural information rather than comparing binary instructions in basic blocks, and 2) the setting of such a threshold is difficult and different settings may lead to different results. Instead, we apply a more stringent requirement that basic blocks b_1 and b_2 are considered matched to one another if b_1 and b_2 have the same taint representation (possibly both non-tainted) and

- b_1 and b_2 are semantically equivalent (evaluated by symbolic execution and theorem proving as explained above); or
- a predecessor of b_1 and a predecessor of b_2 match; or
- a successor of b_1 and a successor of b_2 match.

We want to see how far we can go with such a stringent definition of matching. Note that it is possible that some matching blocks are not found unless a relaxed definition is used, which can be easily applied in iBinHunt for practical usage.

3.3 Automatic generation of program inputs

Although deep taint reduces the number of matching candidates in basic block comparison, it only helps finding the matchings for basic blocks on the corresponding execution trace. Therefore, deep taint applied to more program inputs is needed. However, random inputs are not the most desired because they may result in the same execution paths. We need to find inputs that traverse different paths in the binary program, which is a similar requirement to those in program testing where test cases are needed to cover more program execution paths.

White-box exploration on binary files has been used in many previous work [3, 25, 19]. We apply the same idea to generate execution traces in an iterative process that incrementally explores new execution paths. In each iteration, we first monitor and record an execution trace. We then use a constraint collector [30] to run symbolic execution on the recorded trace and gather the constraints on inputs on every branching conditions. These constraints capture how the input was processed in the corresponding dynamic execution. We then negate one of these constraints collected to obtain the input constraints that would result in a different execution path, and solve these constraints with the theorem prover to obtain a corresponding real input for deep taint in the next iteration.

There are typically many branching locations on an execution trace. We pick one that may result in the largest number of new basic blocks explored by counting all the uncovered basic blocks of the corresponding sub-tree.

4 Implementation and evaluation

4.1 Implementation of iBinHunt

Figure 2 shows the architecture of iBinHunt. In the rest of this subsection, we briefly describe how each component of iBinHunt was implemented.

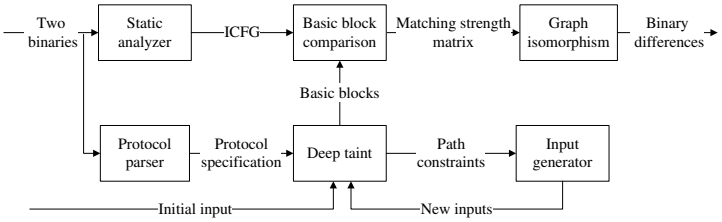


Fig. 2. Architecture of iBinHunt

Static analyzer iBinHunt uses the same static analyzer as in BinHunt [17]. It first disassembles the two binary programs to obtain the x86 instructions, and then converts the x86 instructions into an intermediate representation (IR) for further analysis. The IR we use is the same as in BinHunt and BitBlaze [1, 30], which consists of roughly a dozen different statements. Control flow is analyzed on the IR of the two binary programs to obtain the inter-procedural control-flow graph (ICFG), where nodes correspond to basic blocks in the program and edges correspond to transitions among the basic blocks.

Protocol analyzer We assume that the protocol specifications are known, and therefore a protocol analyzer is not needed. In case the protocol specification is not known, any automatic protocol analyzer [10, 34, 4] can be used.

Deep taint Deep taint was based on TEMU [36] and QEMU[5]. TEMU uses a shadow memory to store the taint status. We modify the shadow memory and add a small data structure for each taint byte to store its corresponding taint tag. Currently deep taint supports up to 64 different tags.

Basic block comparison The dynamic traces from deep taint are first mapped to the ICFG. This mapping is simple as the `eip` value recorded in deep taint and the program counter value in ICFG differ by the length of the corresponding instruction. Once this mapping is obtained, comparison of two basic blocks from the two binary programs is carried out if they have the same taint representation and are on dynamic traces recorded given the same program input.

We use the same basic block comparison technique as in BinHunt [17], i.e., symbolic execution is first used to represent outputs of the basic blocks with their input symbols, and a theorem prover (STP [15]) is then used to check if the outputs from the two basic block are semantically equivalent. Note that the basic block comparison performed here is slightly different from BinHunt in that here the comparison is context aware, i.e., the permutation of outputs of the equivalent basic blocks is the permutation of inputs of the successor blocks. This is because the basic blocks to be compared here are on a particular execution path, where there is always a unique predecessor and a unique successor.

[5] QEMU, `www.qemu.org`

Graph isomorphism iBinHunt also uses the same (customized) backtracking technique to find the maximum isomorphic subgraph as in BinHunt [17].

Difference from BinHunt Although some components of iBinHunt are very similar to those in BinHunt as explained above, there is a major difference between the two, namely iBinHunt uses a dynamic component of deep taint while BinHunt bases purely on static analysis of the binary programs.

Input generator Path constraints are collected as in `appreplay` [30]. We use STP [15] to find a new input that satisfies the negated constraints.

4.2 Evaluation

We applied iBinHunt to find semantic differences in several versions of `thttpd` and `gzip`. We chose to work on `thttpd` and `gzip` for two main reasons. First, they were commonly used programs for which we could find various older versions that are substantially different from the latest one, an evaluation criteria we have for iBinHunt. Second, both `thttpd` and `gzip` had known vulnerabilities in their earlier versions, which is a typical application scenario of iBinHunt.

To evaluate iBinHunt in its resistance to function obfuscation, we simply use iBinHunt to analyze the inter-procedural control-flow graphs instead of enumerating different obfuscation techniques. As discussed in Section 3, iBinHunt removes repetitions and flattens function structures, which will result in the same ICFG no matter what function obfuscation techniques are used.

There are two main aspects on which we want to evaluate. First, we want to see how many basic blocks can be matched, how many matchings are identified by deep taint, and how long it takes to find these matchings. Second, we want to take a closer look at the differences found, and confirm these differences by comparing them to the ground truth (program source code).

Table 2 and Table 3 show the simple statistics of the various versions of `thttpd` and `gzip`, respectively. Note that in some cases, the differences account to nearly 40% of the source code, which we consider very big changes between the two versions. Due to the space limitation, we do not detail all these changes, most of which are due to bug fixing and new features added.

thttpd-	2.20	2.20c	2.21	2.25
2.19	252/6029	254/5843	1483/6641	2908/7271

Table 2. Different versions of thttpd (number of lines changed / total number of lines)

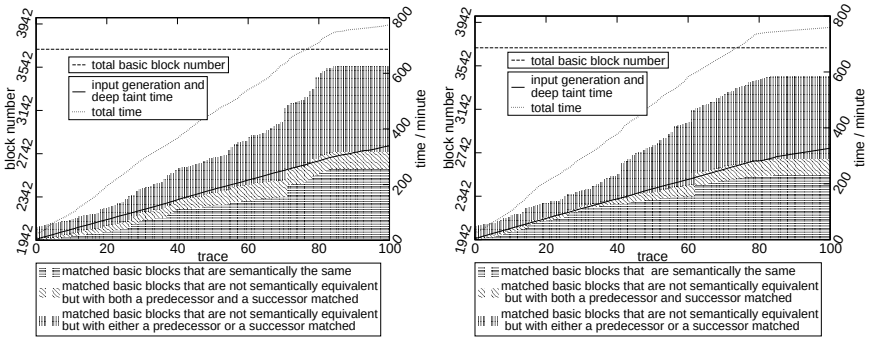
We performed our experiments on two machines, one with a Core2 Duo CPU of 2.6 GHz and RAM of 4 GB (for deep tainting) and another with a Core2 Duo CPU of 3.0 GHz and RAM of 4 GB (for all other components).

Figure 3 and Figure 4 show the results of `thttpd` and `gzip`, respectively. Each graph shows six different types of information.

gzip-	1.3.12	1.3.13	1.40
1.2.4	1317/4959	1351/4929	1446/4841

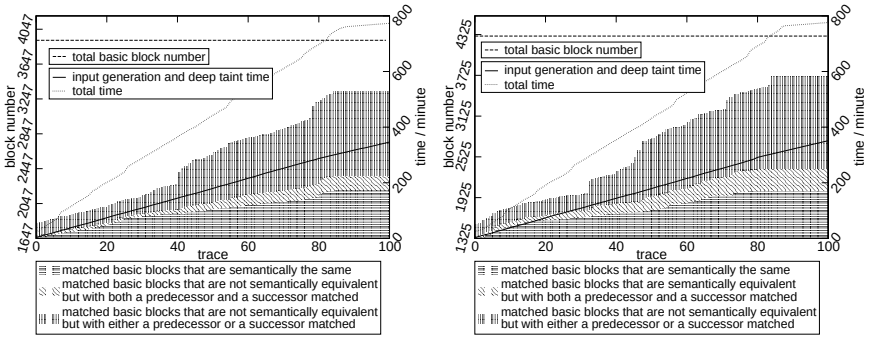
Table 3. Different versions of gzip (number of lines changed / total number of lines)

- Shaded areas: the three shaded areas show the number of matched blocks according to our definition of matching in Section 3.2. The horizontal shaded area corresponds to matched basic blocks that are semantically the same; the 135-degree shaded area corresponds to matched ones that are not semantically equivalent but have both a predecessor and a successor matched; and the vertical shaded area corresponds to those that are not semantically equivalent but have either a predecessor or a successor matched.
- Lines: the lower slanted line indicates the time taken for input generation and deep taint; the upper slanted line indicates the total time spent; and the horizontal line shows the total number of basic blocks in the binary program;



(a) thttpd-2.19 vs. thttpd-2.20

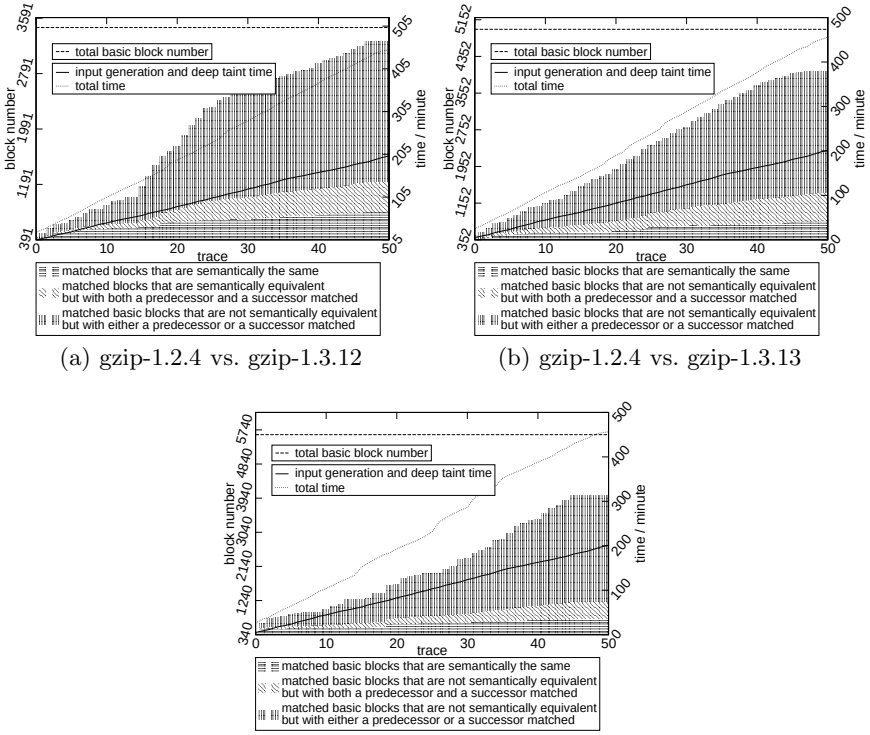
(b) thttpd-2.19 vs. thttpd-2.20c



(c) thttpd-2.19 vs. thttpd-2.21

(d) thttpd-2.19 vs. thttpd-2.25

Fig. 3. Evaluation on different versions of thttpd



(a) gzip-1.2.4 vs. gzip-1.3.12

(b) gzip-1.2.4 vs. gzip-1.3.13

(c) gzip-1.2.4 vs. gzip-1.40

Fig. 4. Evaluation on different versions of gzip

Matching basic blocks Although we use a relatively stringent definition of matching (see Section 3.2), iBinHunt manages to find most of the matching blocks. For example, Figure 4 shows that about 90% of the basic blocks are matched in comparing `gzip-1.2.4` and `gzip-1.3.12`, which have over 25% of the lines of code changed. We also study the matchings found, and confirm that they are correct. Most differences are reflected in these matchings, too, with some differences not found; see Section 4.3 for more discussions.

Effectiveness of deep taint Among successfully matched basic blocks, we count the number of them that actually contain the same taint representation (the rest are not tainted). Results (see Table 4 and Table 5) show that more than 34% and 67% of the matched basic blocks in `thttpd` and `gzip`, respectively, contain the same taint representation. This shows that 1) deep taint is effective in helping to identify basic block matchings since a large number of these matchings do contain the same taint representation; 2) even though many basic blocks are not tainted by our limited number of program inputs, their neighbors are tainted in most cases and the tainted neighbors help matchings to be identified.

thttpd-	2.20	2.20c	2.21	2.25
2.19	34.8%	38.2%	39.9%	37.4%

Table 4. Matched basic blocks with the same taint representation (thttpd)

gzip-	1.3.12	1.3.13	1.40
1.2.4	67.9%	72.2%	72.6%

Table 5. Matched basic blocks with the same taint representation (gzip)

Accuracy iBinHunt has better accuracy in basic block matching because deep taint reduces the number of matching candidates. Typically, the number of candidate matchings is 8% and 5% of total basic block pairs in our experiments with `thttpd` and `gzip`. Refer to Appendix D for another example of accuracy improvement of iBinHunt.

Handling binary programs with big differences The results clearly show that iBinHunt is good in handling binary programs with big differences, a property previous tools for finding semantic differences [17] do not have. These can be seen from the percentage of basic block matched (all shaded areas), which does not decay significantly when dealing with binary programs with larger differences.

Time taken in the analysis From Figure 3 and Figure 4, we see that when more traces are used, more basic blocks are matched until a steady state is reached. 85 and 50 inputs were needed before the number of matched basic blocks stops increasing for `thttpd` and `gzip`, respectively. These 85 or 50 input generations and deep taint analysis are incremental and cannot be parallelized. However, the basic block comparison can be easily parallelized to shorten the time needed. Also note that our implementation is an un-optimized one and there are rooms for improvements. That said, we still see more than a factor of 2 improvement when compared to BinHunt [17] (see Table 6 and Table 7). The starting percentage corresponds to basic blocks that are syntactically the same.

	Percentage of basic blocks matched			Time
	Starting	Ending	Progress made	spent
BinHunt	31%	38%	7%	6 hours
iBinHunt	31%	47%	18%	6 hours

Table 6. Progress made in comparing thttpd-2.19 and thttpd-2.25

Note that results in Table 6 and Table 7 are obtained without parallelizing basic block comparison for a fair comparison. Parallelizing the comparison could speed up the process a lot to make iBinHunt practical in analyzing real programs.

	Percentage of basic blocks matched			Time
	Starting	Ending	Progress made	spent
BinHunt	11%	16%	5%	3 hours
iBinHunt	11%	25%	14%	3 hours

Table 7. Progress made in comparing gzip-1.24 and gzip-1.40

4.3 Discussions

Although we focus on analyzing the inter-procedural control-flow graph in demonstrating the advantages of iBinHunt in this paper, iBinHunt is also resistant to other types of program obfuscations, e.g., control flow flattening [32, 8], that existing binary diffing tools cannot handle. This is mainly due to the deep taint analysis we employ, which is a dynamic analysis approach.

The power of iBinHunt is limited by the non-perfect basic block coverage. This is mainly due to limitations of white box exploration technique [19], e.g., path explosion and imperfect symbolic execution to system calls.

Since iBinHunt uses deep taint, it also suffers from some limitations of taint analysis in general, e.g., control dependence, pointer indirection, and implicit information flow evasions [6, 29].

We performed our evaluation and analysis by comparing iBinHunt with another state-of-the-art binary diffing tool BinHunt [17]. We could have made compassion with other binary diffing tools, e.g., BinDiff. However, due to the many heuristics BinDiff and other binary diffing tools use, it is hard to have a fair comparison with iBinHunt, in which such heuristics are not used. We leave it as future work to compare with other binary diffing tools.

5 Conclusion

In this paper, we first introduce function obfuscation attacks in existing binary diffing tools that analyze intra-procedural control flow of programs. We propose a novel binary diffing tool called iBinHunt which, instead, analyzes the inter-procedural control flow. iBinHunt makes use of a novel technique called deep taint which assigns different taint tags to various parts of the program input and traces the propagation of these taint tags in program execution. iBinHunt automatically generates program inputs to improve basic block coverage. Evaluations on comparing various versions of `thttpd` and `gzip` show that iBinHunt offers better accuracy and efficiency than existing binary diffing tools.

References

1. BitBlaze: Binary analysis for computer security. `http://bitblaze.cs.berkeley.edu/`.
2. I. Briones and A. Gomez. Graphs, entropy and grid computing: Automatic comparison of malware. In *Proceedings of the 2004 Virus Bulletin Conference*, 2004.

3. D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. Bitscope: Automatically dissecting malicious binaries. Technical report, CMU-CS-07-133, School of Computer Science, Carnegie Mellon University, March 2007.
4. J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM Conference on Computer and Communication Security*, Chicago, IL, November 2009.
5. E. Carrera and G. Erdelyi. Digital genome mapping al advanced binary malware analysis. In *Proceedings of the 2004 Virus Bulletin Conference*, 2004.
6. L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Proceedings of the GI International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA)*, 2008.
7. J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *13th USENIX Security Symposium*, 2004.
8. S. Chow, Y. Gu, H. Johnson, and V. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *Proceedings of 4th International Conference on Information Security*, 2001.
9. C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, Technical Report 148, Department of Computer Sciences, The University of Auckland, July 1997.
10. W. Cui. Discoverer: Automatic protocol reverse engineering from network traces. In *In Proceedings of the 16th USENIX Security Symposium*, 2007.
11. J. Oh. DarunGrim. A binary diffing tool. http://www.darungrim.org.
12. T. Dullien and R. Rolles. Graph-based comparison of executable objects. In *Proceedings of SSTIC 2005*, 2005.
13. M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *Proceedings of the 2007 Usenix Annual Conference*, 2007.
14. H. Flake. Structural comparison of executable objects. In *Proceedings of the GI International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment 2004*, 2004.
15. V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV 2007*, 2007.
16. V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, 2009.
17. D. Gao, M.K. Reiter, and D. Song. Binhunt: Automatically finding semantic differences in binary programs. In *Poceedings of the 10th International Conference on Information and Communications Security (ICICS 2008)*, 2008.
18. M. R. Garey and D. S . Johnso. Computers and intractability : A guide to the theory of np-completeness. 1979.
19. P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium (NDSS 2008)*, 2008.
20. X. Hu, T. Chiueh, and K. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, 2009.
21. O. Jeongwook. Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries. In *Black Hat*, 2009.

22. C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Proceedings of the 2005 Symposium on Recent Advances in Intrusion Detection*, 2005.
23. G. Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcolo*, 9, 1972.
24. P. Li, D. Gao, and M. K. Reiter. Automatically adapting a trained anomaly detector to software patches. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (RAID 2009)*, 2009.
25. D. Molnar, X.C. Li, and D. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of USENIX Security Symposium*, 2009.
26. J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis,and signature generation of exploits on commodity software. In *Network and Distributed System Security Symposium (NDSS 2005)*, 2005.
27. Tenable Network Security Inc. PatchDiff. A patch analysis plugin for ida. http://cgi.tenablesecurity.com/tenable/patchdiff.php.
28. J. Raymond and P. Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of Computer-Aided Molecular Design*, 16, 2002.
29. E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010.
30. D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Gyung Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *4th International Conference on Information Systems Security. Keynote invited paper*, 2008.
31. G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.
32. C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *Proceedings of International Conference of Dependable Systems and Networks*, 2001.
33. Z. Wang, K. Pierce, and S. McFarling. Bmat – a binary matching tool for stale profile propagation. *Journal of Instruction-Level Parallelism 2 (2000)*, 2000.
34. G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS 08)*, 2008.
35. H. Yin and D. Song. Panorama: capturing system-wide information flow for malware detection and analysis. In *ACM Conference on Computer and Communications Security (CCS 2007)*, 2007.
36. H. Yin and D. Song. Temu: Binary code analysis via whole-system layered annotative execution. Technical report, EECS Department, University of California, Berkeley, Jan 2010.

A Size of different functions in thttpd

Figure 5 shows the cumulative histogram of functions with different number of basic blocks in `thttpd`, an `http` server. It can be seen that 96% of the 459 non-

empty functions have fewer than 30 basic blocks. Only 7 functions have more than 50 basic blocks. This makes the graph comparison simple, as in most cases we only need to deal with graphs of fewer than 30 nodes. Graph isomorphism is therefore practical in analyzing programs like `thttpd`.

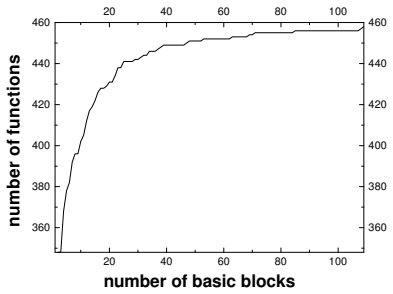


Fig. 5. Number of basic blocks in different functions (cumulative histogram)

B Function inlining and outlining

Figure 6 shows the basic idea of function inlining and outlining transformations. Such simple attacks are effective in confusing existing binary diffing tools because inlining and outlining can arbitrarily increase or decrease the size of any functions. The intra-procedural control-flow graph may contain unreliable information, resulting in a small maximum common subgraph (as in many binary diffing tools, e.g., [14, 12, 27, 17]) or complete failure when the whole program contains only a single function (as in some malware analysis tools, e.g., [20]).

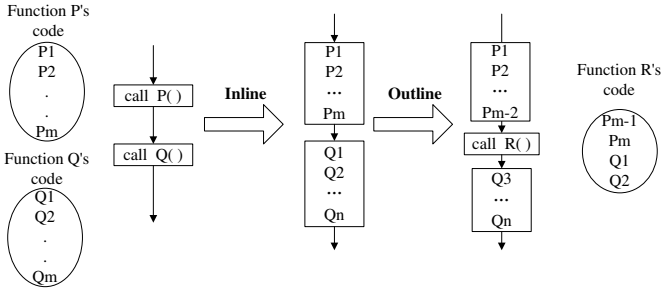
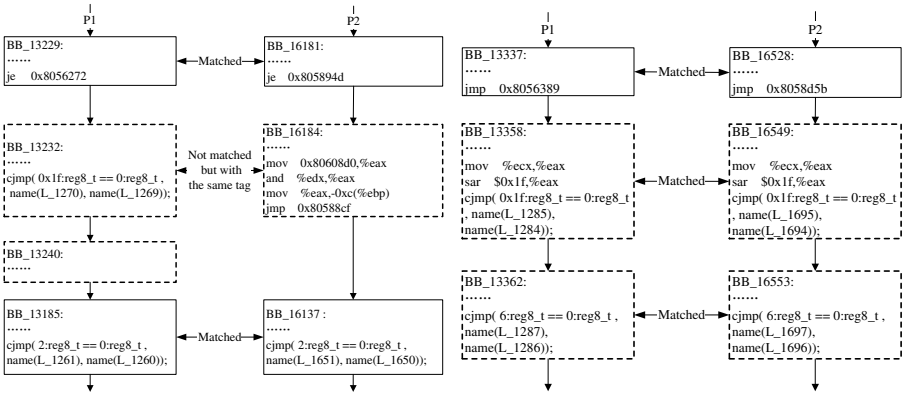


Fig. 6. Inlining and outlining transformations

C Example of potential matching blocks

Figure 7 shows an example of these two groups of blocks in `thttpd-2.19` and `thttpd-2.25`. In Figure 7(a), `BB_13232` and `BB_16184` are not semantically equivalent, but they have the same taint representation (0011). They both originally come from function `find_hash()` corresponding to a difference in the hash algorithm used in the two versions of `thttpd`. In Figure 7(b), the four dashed blocks are not tainted. A closer look into the corresponding source shows that these blocks are part of the function `tmr_create()`, which does some simple time routine and therefore are not tainted.



(a) Blocks with same taint representation (b) Blocks not tainted

Fig. 7. Potential matching blocks

D Improved accuracy of iBinHunt

Figure 8 shows an example in which iBinHunt outputs basic block matching with improved accuracy.

In this example, `BB_1371` from `thttpd-2.25` should match with `BB_1689` in `thttpd-2.19`, both of which deal with the "-i" argument. However, `BB_1687` in `thttpd-2.19` also contains the same (type of) instructions, which confuses the binary diffing tool in the matching. We tried BinHunt [17] and found that BinHunt, in fact, finds the wrong matching in this case.

On the other hand, iBinHunt easily avoids such errors because the different taint representation `BB_1687` has, and therefore `BB_1687` is not even on the list of matching candidates of `BB_1371`.

Besides confirming that the differences found by iBinHunt correspond to semantic differences in the source code, we also verified that these differences include many patches to vulnerabilities in the earlier version. Therefore, iBinHunt

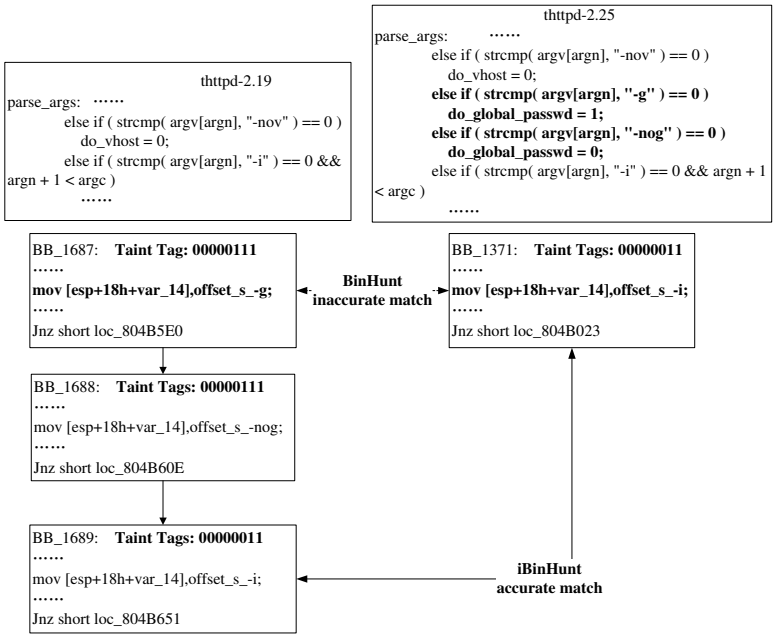


Fig. 8. Accuracy improvement

can be used to automatically find vulnerabilities by comparing different versions of a program for automatic vulnerability discovery, which is an important security application.