Semantic Similarity in Binaries with BinHunt

Benjamin Simmonds, UNSW Canberra

May 2019

Abstract

Extracting meaningful semantic differences between software binaries without source code is difficult. This is a challenging problem due to the overwhelming amount of syntactic noise that small changes can result in at the assembly level. Curiously when it comes to program semantics the "signal from the noise" can be distilled in a manner that is both static and processor agnostic, through the application of control flow and graph isomorphism analysis, symbolic execution and theorem proving. The graph isomorphism problem has no known polynomial time algorithm (i.e. is NP) making brute force approaches computationally infeasible. By blending various static analysis techniques and applying some generalisations, consider a novel approach to overcoming the computationally infeasibility of this problem domain with a view to binary difference analysis.

Introduction

Software binaries, or architecture specific compiled programs are often treated like black boxes by consumers. Over time their creators can release later versions of the programs, to address security vulnerabilities or provide modified functionality. This is not always, as is often the case with commercial closed source software, a transparent process. If it were possible to distill the traits between the binaries that contribute to functional differences between the programs, could infer semantic level changes. Semantic differences, unlike syntactic, correspond to changes in the programs functionality. Given the ability to extract meaningful semantic differences between two binaries, opens up several valuable use-cases. For example, being able to deduce the nature of changes between two versions of a particular program. In the case of polymorphic malware, could highlight the specific evasion techniques being employed. Similarly semantic related functions, across a large collection of binaries, plays an important role towards malware detection. From a malicious perspective, given pre-patched and post-patched binaries, being able to determine semantic similarity is useful for techniques such as automatic patch-based exploit generation for highlighting to the attacker the patched vulnerability.

The BinHunt paper published over a decade ago (2008), presents a static binary

comparison technique based on CG and CFG analysis, symbolic execution, theorem proving and induced graph isomorphism. My goal is to explore these various techniques in more detail in this paper, and consider their relevance in contrast to alternative techniques presented since the first publication of the BinHunt paper in 2008. The overall architecture of BinHunt can be summarised as a graph isomorphism based, block based similarity evaluator. Even though BinHunt hinges on graph isomorphism, which has no known polynomial time solution, applies some clever heuristics to make is usably fast in practice (Egele et al. (2014)).

Finally I highlight some findings around its commercial and technical feasibility, possible applications, scalability and limitations.

Literature Review

The BinHunt architecture is a multistage process that involves disassembly, intermediate representation, control flow and call graph analysis, symbolic execution and a subgraph isomorphism. To gain an appreciation for these techniques over others, need to establish a conceptual level understanding of the mechanics of these specific techniques.

BinHunt proposes a graph isomorphism based approach, call graph and control flow graph analysis, symbolic execution, theorem proving and induced subgraph isomorphism.

Static Analysis

BinHunt can be classed as a static, graph isomorphism-based profiler. It attempts to gauge semantic similarity of units within the binary, by analysing it at rest (i.e. statically), drawing conclusions based on the layout of instructions. BinHunt employs symbolic execution and theorem proving to aid in making these conclusions.

Unlike graph isomorphism approaches to statically analysing the binary, other static approaches exist. Nataraj et al. (2011) for example proposes a novel method for visualising and classifying binaries using image processing techniques.

This contrasts dynamic analysis, which unlike static works by executing the program binary in some sort of emulated or virtual environment while carefully observing and recording the various side-effects the program makes as it executes. A real-world dynamic approach such as Blanket Execution approach presented by Egele et al. (2014), which highlights a number of common challenges that static graph isomorphism based approaches struggle with, mainly to do with the brittleness from which the approach suffers due to trivial changes. Some examples include if the binaries being compared are produced with different compiler toolchains and/or optimisation levels. A change in optimisation level alone (e.g. `-O0` vs `-O3`) has been measured to impact the accuracy of a graph

isomorphism-based approaches upto 25% (P1 Egele et al. (2014)). Blanket execution of a function, unlike BinHunt, dynamically executes the function over and over until each instruction within the function has been exercised at least once. To achieve full coverage, successive runs of the program start so far, uncovered instructions. Side effects and various runtime information is recorded as features, for later similarity evaluation.

When it comes to precise disassembly of a binary, it is known to be an undecidable problem (Cesare and Xiang (2012), P57), that is, a problem for which it is proved to be impossible to construct an algorithm that always leads to a correct yes or no answer. For example the presence of indirect branch targets and call targets, makes quantifying the precise concrete runtime values challenging.

Disassembly

The process of translating the machine code of the binary files on disk (typically ELF or PE) into a sequence of machine specific human readable assembly instructions, for example x86. Cesare and Xiang (2012) highlights for static disassembly two main algorithms are available, Linear Sweep and Recursive Traversal. In *Linear Sweep* instructions are blindly disassembled one instruction after another, including any data segments that may exist in the instruction stream, which is undesirable. *Recursive Traversal* follows the order of control flow (CFG), overcoming the disassembly of possibly embedded data, however is prone to miss decoding difficult to interpret instructions (such as indirect jumps or similar situations).

Intermediate Representation

The process of translating machine specific assembly (e.g x86) to simpler and common instruction representation. One benefit from doing so, insulates downstream BinHunt pipeline from machine specific concerns, a point of complexity, particularly when dealing with CISC (Complex Instruction Set Computer) based architectures such as x86 and x64 (Gao, Reiter, and Song (2008), P5). In the case of BinHunt, a simplified *Intermediate Representation* (IR) is proposed, resulting in loss of expressiveness and precision, but greater reliability, performance and simplicity.

Intermediate code generation is most simply performed in a stateless manner, by translatign each instruction without maintaining any intermediate state (Cesare and Xiang (2012) P50). Various grammars exist such as Google's Reverse Engineering Intermediate Language or *REIL* (LLC (2011)) and the Valgrind IR *Vex*. An example x86 to REIL translation for example:

Example instruction in x86:

```
add eax 10
```

Same instruction in REIL:

```
add (eax, b4), (10, b4), (eax, b8)
```

Control Flow Graph (CFG) and Call Graph (CG) Analysis

Given the binary has been pulled apart to the point where actual analysis of semantic differences between representations can take place. BinHunt, like other graph isomorphism based approaches, leans on modelling the control flow of the instructions. Control flow is regarded as a more useful metric to base comparisons on, given its resilience to cosmetic changes such as specific register allocations or block reordering which commonly takes place.

Control flow analysis is not without it challenges; separation of code and data regions is difficult, as it the presence of indirect branch targets and call targets, making precise conclusions about static control flow undecidable i.e. not precisely and consistently possible (Cesare and Xiang (2012), P52).

One simple method of building a control flow graph is to filter out indirect targets, and representing each call graph (CG) by connecting an edge in the graph from the call site to the static call target (i.e. function calls). A control flow graph (CFG) can similarly be achieved by applying each edge in the graph to branch targets (Cesare and Xiang (2012), P53). These units between branch targets are referred to as *basic blocks* in the BinHunt paper.

BinHunt post CG and CFG rendering from both binaries, now has a possibly overwhelming number of graphs for each binary which need to be compared. The problem with doing graph isomorphic analysis at this point, is that isomorphic CG and CFG graphs may exist between the two binaries, but the nodes contained within the graphs may actually be functionally different.

A basic block similarity ranking is needed to overcome the problem of incorrectly classifying two graphs as isomorphic even with functionally different nodes, while at the same time tackling the issue of overcoming the computational infeasibility of doing graph isomorphism. This very block ranking solution forms part of the novel approach that BinHunt presents.

Symbolic Execution and Theorem Proving

Given a basic block, which are conceptually small units of IR instructions, need a way of assessing its functional fingerprint, against that of another comparison block. Without actually executing the block and making observations of various register states (as would a dynamic approach), statically need to hypothesis the resulting output registers impacting by various input registers. One approach for achieving a hypothesis as to the functional possibility of a given block of logic is known as *symbolic execution*, and is the approach taken by BinHunt.

Symbolic execution, a kind of static analysis, reasons about programs with unspecified inputs, which can represent any possible concrete value, sometimes coined a *free variable*.

When a program deals with a free variable, an entire space of possible execution paths is up for consideration, one for each possible value the variable could take. Where concrete execution corresponds to a single execution path in this space, symbolic execution is concerned with the entire (possibly infinite) space. The size of this plane of possibility is compounded when multiple free variables are considered within the same program. The size of the space is multiplied by each free variable that exists, as the program can behave differently for every possible combination of concrete values (Schroeder and Burget (2019)).

To overcome the computational infeasibility of this reasoning process, the space of all possibilities can be constrained by representing the relationships that exist between logical expressions in the basic block (program fragment).

```
int f(int x, int y) {
  return (x + y + 7);
}
```

For example, based on the above program, could model the logical expression constraint:

```
result = x + y + 7
```

Based on this constraint, could sample concrete values, highlighting the relationship that exists between x and y, and the result.

- x = -1, y = 0, result = 6
- x = 0, y = 1, result = 8
- x = 1, y = 2, result = 10,
- x = 2, y = 1, result = 10
- ...

The relationship between variables is the key in symbolic analysis, and not the actual concrete values. Being able to effectively ask questions about the field of constrained relations is where theorem proving plays an important role.

Theorem proving can be conceptualised simply as a querying tool over the field of possible relations that symbolic execution makes available. An example query could be *do any values of x and y exists that produce a result of 10?*. In this query x and y are "free variables", while the result in concrete i.e. 10. The query itself further constrains the field of possibilities in the set of relations (i.e. all relations the produce a result of 10). In the sample set of relations listed above, at least one model satisfies assigned to the x and y variables, meeting the conditions of the query. The free and concrete factors can be experimented with by leveraging the querying mechanic provided by theorem proving, e.g. *if x is 3, if there some value of y that will produce a negative result?* and so on.

Applied to the field of computer science theorem provers such as the Z3 solver from Microsoft (`https://github.com/Z3Prover/z3`), or the STP constraint solver (`https://stp.github.io/`) as used by BinHunt, tackle the broader problem of satisfiability modulo theories (SMT).

Satisfiability Modulo Theories (SMT) problem is a decision problem for logical formulas with respect to combinations of background theories such as arithmetic, bit-vectors, arrays, and uninterpreted functions. Z3 is an efficient SMT solver with specialized algorithms for solving background theories. SMT solving enjoys a synergetic relationship with software analysis, verification and symbolic execution tools." (Bjørner et al. (2018))

An interesting real world application of Z3 took place in the BSides Canberra 2019 CTF (Tom (@x86party) (2019)). After decompiling some Java that did native JNI interop with a C-based password checker function, Tom found that dozens of constraints were evaluated against the input string, for example:

```
if (((iVar7 - iVar15) - iVar12) - iVar2 == -0x72) {
  iVar13 = (int)__s[0x11];
  iVar3 = (int)__s[6];
  if ((int)*__s * iVar13 + iVar3 == 0x27e5) {
  ...
```

Tom opted to model the constraints as a satisfiability problem using the Z3 theorem prover.

"My Z3 solver is pretty basic. Each byte in the input string is treated as an integer, numerical constraints are applied to each input byte using a solver object, the solver object checks if the constraints can be satisfied (using math(s)), if the constraints can be satisfied, print the result." (Tom (@x86party) (2019))

In the case of BinHunt, the STP (`https://stp.github.io/`) theorem prover is put to work to measure the possible equality that might exist between two basic blocks, by assessing possible output register results for each. While this is useful in determining if two basic blocks will produce equivalent side-effects, it does not assist in helping identify the specific registers to pick for evaluation, as it is highly likely, that different registers will be used between the different blocks being compared (which in turn stem from different binaries) while providing identical functionality. Here in lies a major difficulty that afflicts graph isomorphism based approaches, like BunHunt, which must brute force evaluate all possible pairwise combinations of registers, deeming a functionally equivalent basic block match if a set of registers is discovered where the values are the same.

Exact Subgraph Isomorphism

The *workhorse* algorithm and arguably most novel concept of the BinHunt paper, is its approach for testing the similarity between large collections of graphs, derived from both binaries.

The graph isomorphism problem, has been the focus of many great minds for several decades, is one out of only a few problems that exist in the domain of computational complexity theory belonging to the NP complexity class.

> "NP (nondeterministic polynomial time) is a complexity class used

> to classify decision problems. NP is the set of decision problems for which the problem instances, where the answer is"yes", have proofs verifiable in polynomial time." (Tardos and Kleinberg (2006)).

Fortunately the generalisation of graph isomorphism, the *subgraph isomorphism problem*, is known to be NP-complete, a complexity class solvable by brute force searching.

Graph isomorphism is one of only two problems, whose complexity remains unresolved, the other is *integer factorisation*. It is known that if the problem is NP-complete, as is the subgraph isomorphism problem, then the polynomial hierarchy collapses to a finite level (Schöning (1988)).

BinHunt integrates similarity heuristics, as discussed in the symbolic execution and theorem proving section above, in the form of a *backtracking* based customisation. A backtracking algorithm is used to identify and shortlist possible graph node match candidates. The algorithm incrementally builds possible candidates matches, and abandons (backtracks) a candidate match once it is evident a better one exists.

The BinHunt paper formalises its approach to the problem of determining exact subgraph isomorphism as the *Maximum Common Induced Subgraph Isomorphism* problem. The formalisation is the foundation for definition of the backtracking algorithm, which integrates subgraph similarity measurement and best possible candidate node shortlisting, which is the node that has the highest matching strength with nodes in the other comparison graph. Interestingly, if there are multiple nodes with the same matching strength, the node with the great connectivity (ingress and egress edges).

The algorithm is put to work to find the highest similarity between functions in each binary, and then the highest similarity between basic blocks within functions. A beneficial characteristic of the backtracking algorithm is that poor matches are weeded out and replaced by better matches, resulting in decent accuracy in match results.

Ultimately the algorithm outputs match result between the functions contained in the two binaries, and subsequently a matching result between basic blocks within two matched functions. Given these matchings combined with the matching strengths (yielded from symbolic execution and theorem prover) highlight where semantic differences are most likely to occur.

Commercial Feasibility

Unfortunately BinHunt is only evaluated against three case studies, all related to differences between similar versions relating to patched vulnerabilities. In particular it has not been shown how well BinHunt performs against binaries compiled on different toolchains or with varying compiler optimisation levels. Egele et al. (2014) highlights that while graph isomorphism approaches work

well when two semantically equivalent binaries have control flow graphs (CFG), it is trivial to create semantically equivalent binaries that have radically different CFGs. A change in optimisation level alone (e.g. `-O0` vs `-O3`) has been measured to impact the accuracy of a graph isomorphism-based approaches upto 25% (Egele et al. (2014), P1).

When it comes to precise disassembly of a binary, it is known to be an undecidable problem (Cesare and Xiang (2012), P57), that is, a problem for which it is proved to be impossible to construct an algorithm that always leads to a correct yes or no answer. For example the presence of indirect branch targets and call targets, makes quantifying the precise concrete runtime values challenging. Perhaps strict precision in this area is not a prerequisite in order to provide useful approximations.

It is also noted that BinHunt avoids doing symbolic analysis and theorem proving at the function level, due to major performance reasons of functions of even a moderate size, and instead reserves this analysis for only basic blocks (Gao, Reiter, and Song (2008), P245). During this analysis on basic blocks, pairwise comparison of registers for each block is performed.

Current state of the art binary comparison tools use a variety of pairwise comparisons. However this method is unscalable for clustering large datasets, for size N, since they require O(N2) comparisons (Jin et al. (2012)). More scalable solutions in this problem space involve creating semantic hashes, which captures the semantics of a function symbolised simply as a hash (Cesare, Xiang, and Zhou (2012), P5).

"Determining whether two programs are semantically equivalent is also known to an undecidable problem which is why for example malware detection is often based on heuristic and unsound solutions." (Cesare and Xiang (2012), P57)

While the above highlights some issues with BinHunt, in a commercialised capacity what BinHunt brings to the table is still hugely useful, possibly when put in combination with a blend of other techniques.

Conclusion

At the time of publication in 2008, the BinHunt researchers presented a combination of techniques in new and innovative ways. Additionally the concepts put forward by BinHunt, have inspired follow-up research (150+ cited usages in published papers), cementing the significant contribution it has made in this field. Some noteworthy novel contributions include:

- Shows static analysis, with all its computationally infeasible difficulties, when overcome creatively can yield fruitful results.
- Proves functional equivalence guarantees between comparisons of basic blocks using rigorous symbolic execution and theorem proving techniques.

- A clever matching strength ranking system to quickly categorise semantic block similarity by combining symbolic analysis and theorem proving.
- For the sake of efficiency, uses block over function level similarity analysis.
- Formalises and shortcuts the problem of determining exact (not approximate) subgraph isomorphism as the *Maximum Common Induced Subgraph Isomorphism* problem, which forms the basis for backtracking algorithm that leverages subgraph similarity measurement.

Although BinHunt is not a perfect solution (disassembly process is an undecidable problem, assembly to IR translation is lossy, static control flow and call graph analysis is not always precisely and consistently possible), and has attracted some criticisms, it successfully highlights innovation towards overcoming hard problems by combining a range of existing techniques, in ways not done previously.

References

Bjørner, Nikolaj, Leonardo de Moura, Lev Nachmanson, and Christoph M Wintersteiger. 2018. "Programming Z3." In *International Summer School on Engineering Trustworthy Software Systems*, 148–201. Springer.

Cesare, Silvio, and Yang Xiang. 2012. *Software Similarity and Classification*. Springer Science & Business Media.

Cesare, Silvio, Yang Xiang, and Wanlei Zhou. 2012. "Malwise—an Effective and Efficient Classification System for Packed and Polymorphic Malware." *IEEE Transactions on Computers* 62 (6): 1193–1206.

Egele, Manuel, Maverick Woo, Peter Chapman, and David Brumley. 2014. "Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components." In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 303–17.

Gao, Debin, Michael K Reiter, and Dawn Song. 2008. "Binhunt: Automatically Finding Semantic Differences in Binary Programs." In *International Conference on Information and Communications Security*, 238–55. Springer.

Jin, Wesley, Sagar Chaki, Cory Cohen, Arie Gurfinkel, Jeffrey Havrilla, Charles Hines, and Priya Narasimhan. 2012. "Binary Function Clustering Using Semantic Hashes." In *2012 11th International Conference on Machine Learning and Applications*, 1:386–91. IEEE.

LLC, Google. 2011. "REIL - the Reverse Engineering Intermediate Language." February 2011. https://www.zynamics.com/binnavi/manual/html/reil_language.htm.

Nataraj, Lakshmanan, Sreejith Karthikeyan, Gregoire Jacob, and BS Manjunath. 2011. "Malware Images: Visualization and Automatic Classification." In *Proceedings of the 8th International Symposium on Visualization for Cyber Security*, 4. ACM.

Schöning, Uwe. 1988. "Graph Isomorphism Is in the Low Hierarchy." *Journal of Computer and System Sciences* 37 (3): 312–23.

Schroeder, Brian, and Joel Burget. 2019. "A Gentle Introduction to Symbolic Execution." April 2019. https://blog.monic.co/a-gentle-introduction-to-

symbolic-execution/.

Tardos, Eva, and Jon Kleinberg. 2006. “Algorithm Design.” Reading (MA): Addison-Wesley.

Tom (@x86party), TSS. 2019. “CTF Writeup - You Shall Not Pass.” March 2019. https://medium.com/tsscyber/ctf-writeup-you-shall-not-pass-2c7a9254549b.