



CRIMSON VISTA
CYBERSECURITY ENGINEERING

Uncovering Misappropriated Trade Secrets in High-Tech Source Code

WHITE PAPER

July 2020

Introduction

Trade secret misappropriation involving software can often take different forms. Identifying such misappropriation is rarely done via direct code matching, as most “copying” may present itself in the form of underlying structure and ideas rather than literal copying.

To demonstrate strategies for approaching complex cases, this whitepaper describes three case studies which cover:

- Direct copying of source code
- Formulas embodied in source code
- Engineering practice embodied in source code

There are other complex trade secret misappropriation scenarios that exist in the real world, but these representative case studies will provide a glimpse into how such scenarios can be addressed.

Case Study #1: Direct Copying of Source Code

In some circumstances, misappropriation of trade secret involving source code involves taking confidential design ideas, concepts, and/or implementations directly from the code. While in many ways this is the most obvious form of theft, it can still be a challenging process to find the copied data.

The following case study illustrates some common difficulties and exemplary solutions. For the purposes of this case study, we will call the owner of the trade secrets, Alice Inc., and the company that misappropriated the trade secrets, Bob Co.¹ The dispute between Alice Inc. and Bob Co. involved millions of lines of code across many versions of both parties’ software. Analyses went on for many months.

¹ In cybersecurity, it is common to use “Alice” and “Bob” to refer to party A and party B.

The easiest part of the investigation was finding literal string matches between the two companies' source code. This included:

- Comments that were literally unchanged, *including the same typographical errors*
- Prefix-renamed elements (e.g., A_foo, A_bar renamed B_foo, B_bar)
- Identical data structures, constants, and control flow

Despite evidence of literally copied code, it was clear to us that the copying was more extensive than this obvious surface level. More importantly, it seemed likely that the full extent of the misappropriated trade secrets was not embodied in the obviously copied code. Further investigation was warranted.

One method for looking for less obvious copying involves looking for sequences of matching elements. This is a common approach for finding plagiarism in university programming classes and various open source tools exist to identify cheating based on these sequences. For example, Moss (Measure of Software Similarity) is an open-source tool that uses this approach.²

During our investigation, we created a modified tool based on the concepts used in Moss, to search for copied code that may have changed over time. Moss, and many other tools like it, is primarily tuned for comparing *contemporaneous software artifacts*. That is an excellent approach to dealing with plagiarism in college classes, but can miss similarities in source code that was *based* on a copy but was subsequently modified and developed. This challenge is visualized in Figure 1.

² See, e.g., *Winnowing: Local Algorithms for Document Fingerprinting* by Schleimer, et al.; see also, <https://theory.stanford.edu/~aiken/moss/>

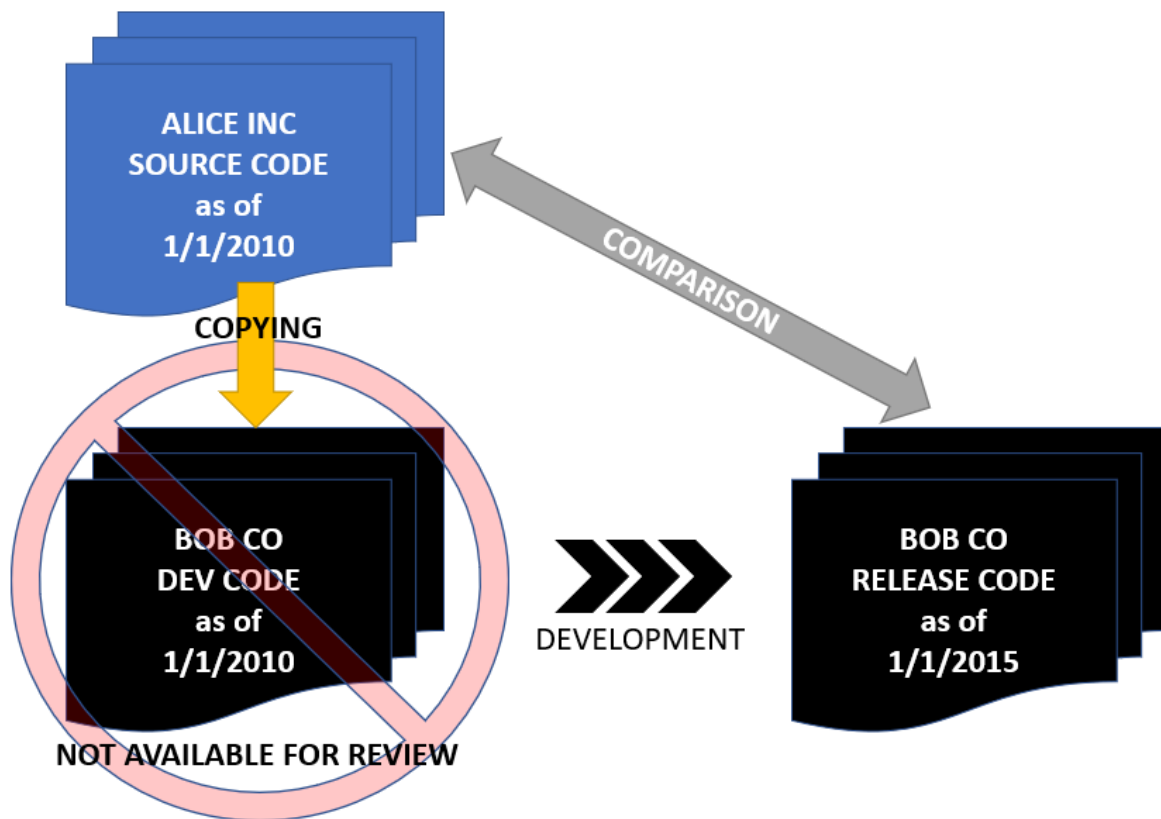


Figure 1. Code is misappropriated from Alice Co on 1/1/2010. The code as of that date is no longer available. Comparison uses code that has been under development for some time after the copying.

Ideally, analyses of code theft include the source code repositories that store an organization's code and track changes to the same. With this kind of access, analysts can compare the code more-or-less at the time the copying/misappropriation occurred. In the matter of Alice Inc. vs Bob Co., we did not have access to the repositories and were forced to analyze code that corresponded to official releases. This necessitated development of a comparison tool that attempts to account for code that, while originally copied in whole or in part, has subsequently undergone significant evolution.

One of the crucial differences between our tool and Moss is that Moss attempts to categorize the file as a whole. Moss compares one file to every other file and produces a score of how similar the two files are. For code that has evolved significantly, the overall comparison may not be high. However, our ultimate goal is not to show if the code is similar, but if trade secrets were misappropriated. We are looking to the source code to help us find those misappropriations. So, instead of scoring a comparison between files, our tool helped us find *artifacts* in the code that could point us in the right direction. Using the uncovered remnants of literal copying, we can proceed with an analysis of what

trade secrets were embodied in the copied code and what impact this misappropriation had on the subsequent development process.

In the case of Alice Inc. and Bob Co., it was clear that Bob Co. had designed much of the system around the designs of Alice Inc. Even though the code had evolved, the collection of uncovered artifacts enabled us to find starting points of comparison between the two code bases. Moreover, collecting a wide range of copying also helped to establish connections between software architectures that were far more important than each individual snippet of copied text.

Another challenge in identifying all of the copied intellectual property that Bob Co. had misappropriated was production deficiencies. We determined that some source code, which was likely highly damning, was not produced. While the failure to produce damning code was, itself, damning, we still wanted to see what they were failing to turn over. Fortunately we found what we needed in *binary libraries*.

As a refresher for those unfamiliar with source code and compiled code, human programming is built upon many layers of *translation*. The processor that runs a computer has a surprisingly small number of instructions. Performing even relatively simple computer programming tasks often must be written in complex and unintuitive forms. A human-readable programming language, like C++, allows the programmer to write in a more readable form. Once the program is written, a compiler *translates* the code into a version that is readable and understandable by the processor. This form is often called the compiled form or the binary form. A suite of compiled operations can be merged into a single file called a library. Windows DLL's are an example of a library file. The relationship of these files is depicted in Figure 2.

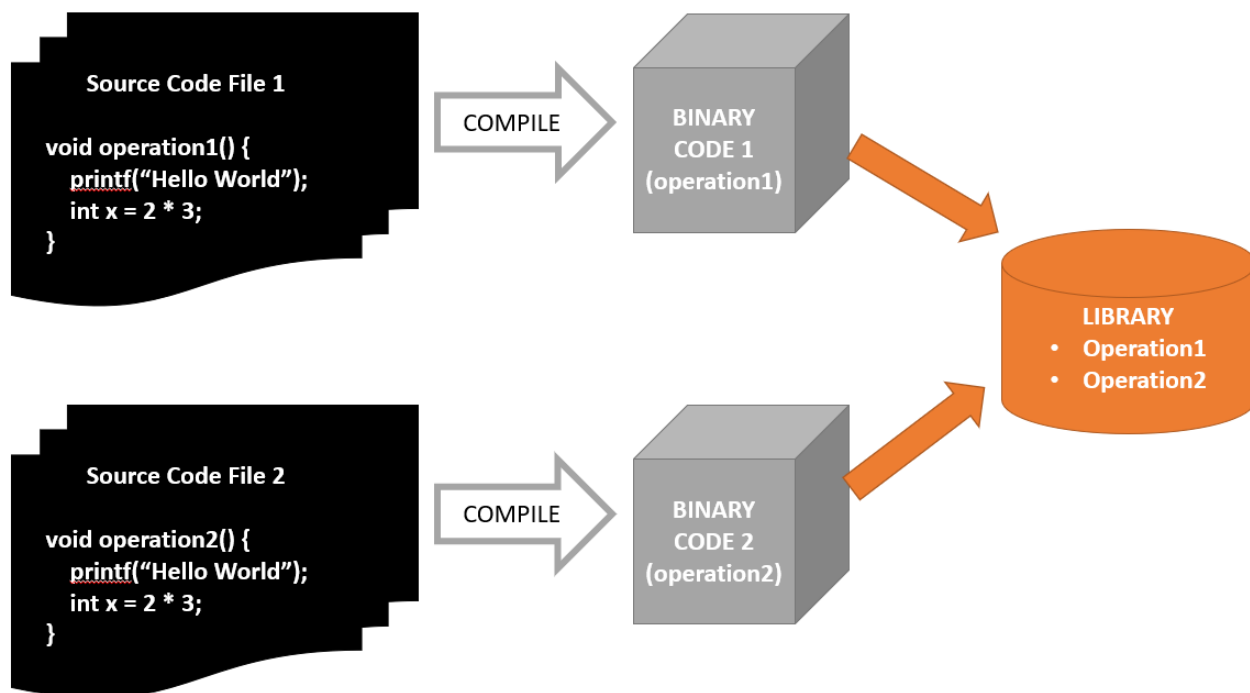


Figure 2. Source code is written in multiple files in a human-readable format. These files are compiled into binary forms that can be consolidated into a single library.

A library is not, by itself, a complete and functional program. Rather, it is a set of operations that can be incorporated into other programs. Using a library, the same operations do not have to be recreated each time but can simply be reused. It is common for software companies to have their own proprietary libraries for their products.

In the case of Alice Inc., we found that Bob Co. had lifted libraries and were using them directly in their own product. The initial discovery came as a result of our earlier source code evaluation. Some of the most obviously copied files, that had undergone the least changes, were *header* files. A header file defines names of operations but not what they actually do. When we found these header files had been copied, we searched for the corresponding files that define the implementation. They had not been produced.

We had been given access to the library files, however, and subsequently determined that the header files corresponded to the compiled code in these libraries. While these files are mostly not human readable, they had human readable components within them that include the names of the operations. The names of the operations in the libraries and the header files were matches.

Moreover, we investigated these binaries to make sure that they were, in fact, copied from Alice Inc. To do this, we made use of a “decompiler”. A decompiler works backwards from a compiler to produce source code from binary code. Unfortunately, a decompiler cannot recover the original literal source code because the process is not one-to-one. However, it does recover source code with matching function names and the general control flow and computations intact. Using these decompiled versions of the source code, we were able to show with high probability that the library was original Alice Inc. code.³⁴ See illustration in Figure 3.

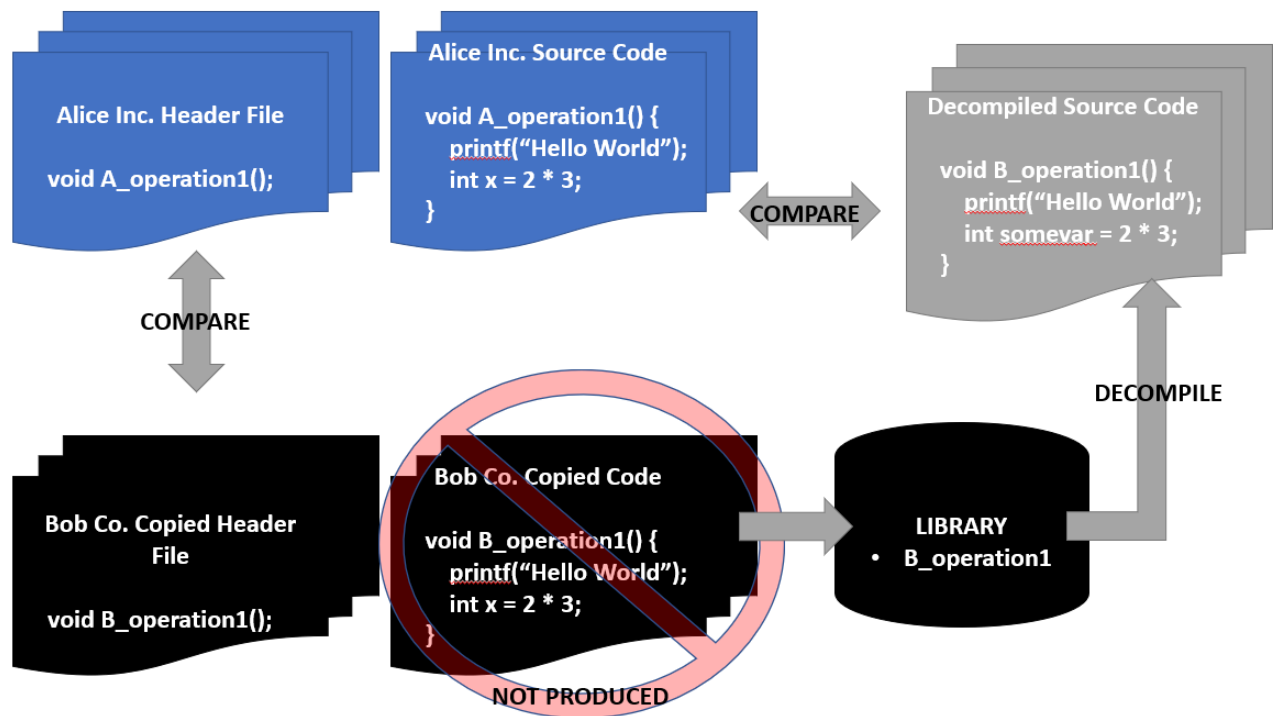


Figure 3. Although the header files were clearly copied, the main code files were not produced. To complete our analysis, we used a decompiler to obtain a close approximation of the original code.

Literal copying can be proven in a variety of ways depending on what files were produced and what tools are available. Comparing the literal similarity of code files can be extremely helpful in some circumstances, but does not identify all forms of copied code, especially after additional development

³ Technically, we did not need the decompiler for this. It is possible to prove a correspondence directly from the machine code. However, this takes much longer and is less jury/judge friendly.

⁴ Actually, Bob Co. did not copy the libraries directly because there were minor changes. This means at some point they had the original source code, modified it, compiled it, and then destroyed it. Or, alternatively, they had it and were simply refusing to produce it. However, the copied libraries were, more-or-less, identical and Bob co. used them without additional alteration over multiple years and versions.

and evolution. However, by understanding the structure of computers and their software, code similarity tests can be derived even when portions of the code are not produced.

Case Study #2: Formulas Embodied in Source Code

The case study of Alice Inc. and Bob Co. dealt with source code that was literally copied. In many cases, however, the misappropriating party takes ideas and concepts from the original source code and then re-inserts those same ideas and concepts into new source code. This may mean that no files were copied in whole or in part. Comparative textual analysis is typically less helpful in revealing examples of misappropriation under these conditions.

This case study examines another matter in which we⁵ participated in source code evaluation. This time, we will call the owner of the trade secrets Charlie Inc. and the misappropriating party David LLC. Charlie Inc. produces software that models stock market behavior and makes predictions for investment purposes. Former employees have left Charlie Inc. and created the competing corporation David LLC. Charlie Inc. alleges that David LLC has misappropriated their confidential trade secrets relating to the stock market modeling and prediction.

During our investigation of David LLC's source code, there were multiple challenges. The review was conducted under an excruciatingly difficult protective order. We were not permitted to even bring paper into the review room, let alone any computational devices. We were also not permitted to do any programming on the review machine making it difficult to create on-the-fly tools for analysis as we did in the case of Alice Inc. vs Bob Co.

The other challenge was that there appeared to be little, if any, literal copying. While Charlie Inc.'s source code was written in the C++ programming language David LLC's was written in Java. There were no obvious examples of simple rewriting (e.g., copying C++ code over and then converting it to Java). It appeared that the new source code was written without relying on the literal contents of the original source code.

⁵ The matter described in this Case Study occurred before Crimson Vista was formed; however, current scientists of Crimson Vista were present representing a previous employer.

During the weeks of the analysis, we examined the source code from many different angles. We looked for data structures that were similar, code architectures, and control flow algorithms. In the end, we did not feel very confident that these elements were copied or influenced by the earlier source code.

As the investigation progressed however, we also became more attuned to how the software worked. As nobody on the team was an expert in these kinds of financial systems, we had to acquire domain-specific knowledge along the way. In so doing, various aspects of the case became more clear including the reason for the paranoid security. Specifically, we learned that the source code contained formulas that, although encoded in a couple of lines, might generate millions of dollars in revenue. It was required that these concepts be kept protected at all costs.

In addition to increasing our understanding of the value of the formulas, we also were getting a better grasp of how they were formed and used within both code bases.⁶ Even as our investigation focused in on these elements, we still found nothing clearly copied. The formulas were such that it was difficult to compare them mathematically.

What we did find that we could do, however, was *simulate* them. Although we could not bring paper into the review room, we had a break-out room across the hall. To capture a formula, we would examine the source code then run across the hall to write down what we had observed.⁷ We repeated this process as many times as was necessary to ensure that we had the formulas correctly represented.

Once the formulas were exfiltrated, we constructed a simulation framework that would take formulas from both companies and compare them. These simulations made no attempt to model the complexities of the stock market or other financial considerations. Rather, they took the formulas and evaluated how they performed over a wide range of inputs. We discovered multiple formulas that behaved in nearly identical ways.

Because the formulas are trade secrets of Charlie Co., if no misappropriation had occurred then the two formulas would nearly certainly not have behaved identically. Although the code was not literally

⁶ The client did not originally refer to them as formulas but instead by more technical jargon. It took some time for both sides to learn how to speak the same language.

⁷ Conceptually, this was bypassing the goal of keeping the formulas from getting out of the room. For whatever reason, it was permitted. On the other hand, we find that most protective orders are unnecessarily restrictive. But that is a topic for a different white paper.

copied, by understanding the functionality of each codebase and identifying the critical formulas, we were able to locate trade secret misappropriation.

Case Study #3:

Engineering Practice Embodied in Source Code

For the final case study, we will create a hypothetical legal dispute between Eve LLC and The Frank Company. This hypothetical action will aggregate together a range of trade secret experiences into a single narrative.

Eve LLC is a software company that performs Big Data Analytics as a service to major corporations and the United States government. The company includes an engineering team for current products and services as well as a research team exploring future opportunities and directions. There is no hardline between the two teams and a single employee may participate in both endeavors.

A few years ago, some of the people working in the research group left Eve LLC after enticements to come and join The Frank Company. While The Frank Company has some competing services with Eve LLC, it is only after the former Eve LLC employees join that they really make aggressive pushes to challenge Eve LLC in the marketplace.

In a somewhat unusual twist, one of the first analyses we are asked to perform is a strange kind of “reverse patent analysis.” The Frank Company filed a patent very shortly after the arrival of the former Eve LLC employees. Eve LLC believes that the patent was created based on work in their Research group.

The Eve research group did not have any source code related to this project but does have extensive documentation of the ideas, concepts, and designs under development in the Research group before the employees left. We provided a strong technical analysis illustrating that the concepts embodied in the claims were concepts described and recorded in Eve LLC research documents. All of this, so far, has not involved any source code.

However, The Frank Company has built source code around the patent and has claimed that their Data Analytics Platform is using patented technology. We conducted a source code review of The Frank Company’s product and determined that it practiced one or more claims of The Frank Company’s own (misappropriated) patent. This unusual twist of a misappropriated patent has made

it easier to prove that The Frank Company's source code includes trade secret concepts from Eve LLC's research lab.

Moreover, a detailed investigation of source code uncovers additional misappropriated trade secrets from Eve LLC's engineering team. Internal documentation from The Frank Company indicates that their product had been less than successful in previous iterations. When the former Eve LLC employees arrive, they introduce new **training sets** for the data analytics platform. Documentation indicates that the individual training sets are open source and, therefore, not a trade secret. However, the knowledge of what combination of training sets will provide effective results for a given target is a matter of significant research in the industry. Eve LLC's combination, which is not generally known, has been very effective in giving the company an edge against competitors. The former Eve LLC employees have introduced that combination of training sets to The Frank Company, misappropriating a highly valuable trade secret.

Here, the source code analysis proving trade secret misappropriation was not a comparison between two sets of source code. Rather, it was a thorough analysis of trade secret misappropriation identified by comparing software development process documentation to patent documents. Then, the patent documentation was finally shown to be implemented by one set of source code belonging to The Frank Company.

Conclusion

Trade secret misappropriation in high-tech software can manifest in various forms. The three case studies included above illustrate strategies to identify direct copying of source code, formulas embodied in source code, and engineering practice embodied in source code.

These strategies are effective due to domain knowledge, computer science expertise, and experience reviewing high-profile codebases involved in a variety of litigations. With this experience, teams are able to identify trade secret misappropriation where it occurs, and adapt to new challenging scenarios where identification is not straight forward.