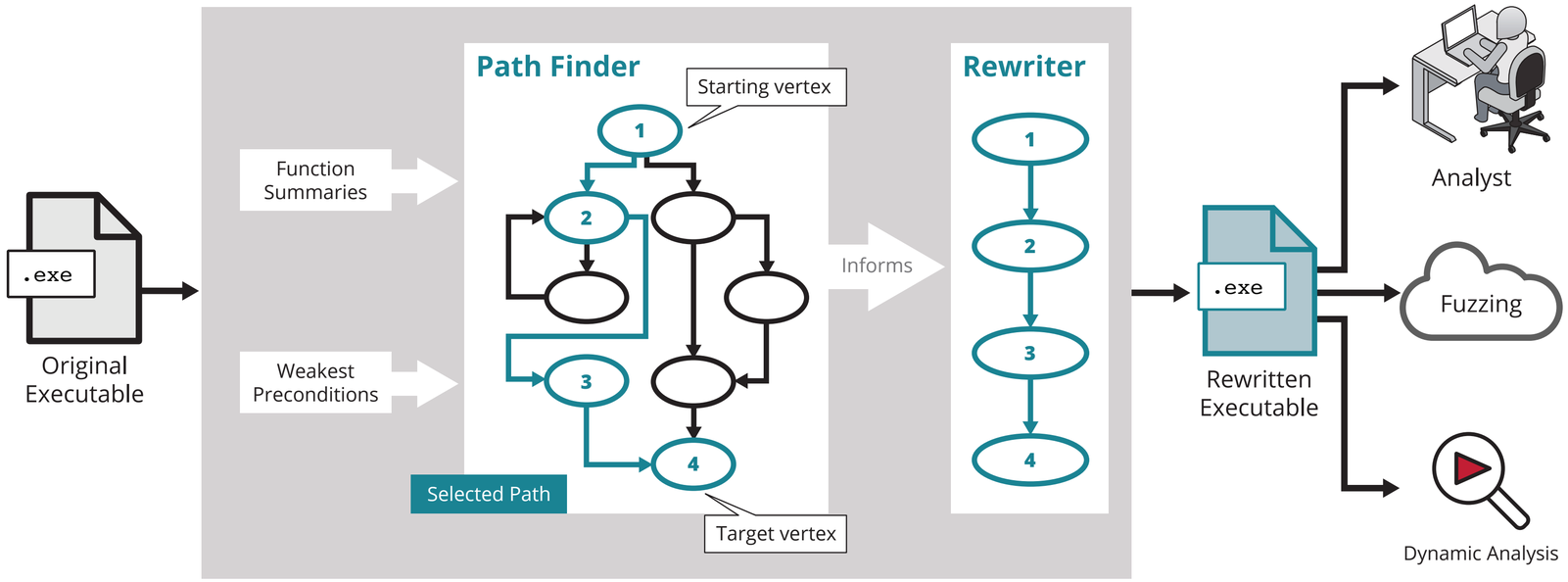
Automatically Understanding Executables

Reducing the cost of manual executable analysis for vulnerability discovery and malware analysis

Automated Executable Path Finding and Rewriting Process



Understanding the conditions that cause an executable to follow specific paths help DoD analysts identify vulnerabilities and understand malware behavior.

We aim to automate understanding the conditions required to cause an executable to reach a specific point in a control flow graph. This could test a specific feature in a piece of malware or establish whether it is possible to reach a vulnerable condition in software. This mitigates a tedious manual process.

There are potentially an infinite number of execution paths that we must search over. To cope with that complexity, we need multiple approaches.

To date we've implemented two path-finding algorithms with different performance and accuracy tradeoffs. We believe that combining the approaches will yield improved performance compared to either in isolation.

A binary rewriter can then create a new binary file that always follows the desired path when executed.

We're partnering with Dr. Arie Gurfinkel at the University of Waterloo to apply source code reachability techniques to executables.

Property Directed Reachability (PDR) has proven to be an effective technique for static analysis of source code reachability. Dr. Gurfinkel maintains a PDR implementation as part of Microsoft's open source SMT solver Z3.

This work is partially supported by the Office of Naval Research (ONR).

For more information about Pharos

https://github.com/sei-cmu/pharos

Automated variable name recovery through large-scale data mining and statistical analysis.

Reverse engineers often read decompiled code to understand the behavior of an executable. Modern decompilers do not attempt to recover meaningful variable names, and instead synthesize names such as *v12*. In this project, we use statistical techniques to learn appropriate variable names.

Decompiled C Code with Synthetic Names

```
v14 = &v15;
asxTab(a2 + 1);
for (v13 = asnContents(a1, &v15, 512LL); v13 > 0; --v13)
{
   v9 = (unsignedchar*)(v14++);
   printf(" %02X ", *v9);
}
```

Decompiled C Code with Recovered Names

```
cp = buf;
(void)asxTab(level + 1);
for (n = asnContents(asn, buf, 512); n > 0; n--)
{
   printf(" %02X ", *(cp++));
}
```

Current results. We evaluated our approach by comparing the variable names recovered by our system with the original variable names in the source code. When our system recovers exactly the same variable name, we call it an exact match. We also measure approximate matches, which occur when our system recovers an abbreviation that consists of at least half the characters of the original. For example, an approximate match would be recovering *buf* for the original variable name of *buffer*.