## Hybrid Code Analysis

Activating hybrid code analysis causes Dyninst to augment its static analysis of the code with run-time code discovery techniques. Hybrid analysis is useful even for binaries that have not been deliberately obfuscated, as Dyninst’s static analysis of the program binary may miss code for which there are no function symbols, and may parse functions incompletely if they contain indirect jumps whose targets are not recognizable as jump table entries. This mechanism also allows for the analysis of analysis-resistant binaries that are stripped of symbol information and may dynamically unpack binary code or modify existing code. These use cases have corresponding modes of hybrid code analysis:

1. **Instrumentation-based discovery:** This mode makes no modification to Dyninst’s static analysis, except to mark statically unresolved control transfers for instrumentation. The following types of control transfers are instrumented: indirect jump instructions that do not follow jump table conventions, indirect calls with statically unresolved targets, and static control transfers to invalid or un-initialized memory regions.
2. **Malware-mode discovery:** This mode makes Dyninst’s static analysis more conservative, and compensates through heavier use of instrumentation. It also assumes that the binary may attempt to modify its own code, so it write-protects all memory pages that contain code, causing overwrite attempts to raise signals. This mode can only be activated if the user registers code-discovery and code-overwrite callbacks, in which case the user can specifically request that this mode be activated, or rely on Dyninst’s heuristics for detecting the likely presence of code obfuscations.
3. **Static analysis only:** This is Dyninst’s traditional mode of analysis; no run-time code discovery is performed.

**Heuristic analysis-mode selection.**  The user can set the desired hybrid-analysis mode or rely on Dyninst’s heuristics. Dyninst chooses static-analysis only for binaries with symbol information. Instrumentation-based discovery is the default for stripped binaries, but malware mode is activated if one of the following conditions is met:

1. The binary lacks a .text section and its first section is writable
2. The binary’s .text section is writable or uninitialized
3. The program contains static control transfers to un-initialized regions
4. The binary contains contains control-transfer obfuscations

enum BPatch\_hybridMode { BPatch\_staticMode, BPatch\_instMode, BPatch\_malwareMode};

bool BPatchImage::setHybridMode(BPatch\_hybridMode mode);

Set the hybrid analysis mode to static analysis only (turns of heuristic enabling of hybrid modes), set it to do instrumentation-based code discovery only, or set it to malware mode, in which the program is assumed to be analysis-resistant.

static void BPatchCodeDiscoveryCallback (std::vector<BPatch\_function\*> &newFuncs, std::vector<BPatch\_function\*> modifiedFuncs);

This callback is invoked whenever previously un-analyzed code is discovered through runtime analysis, and delivers a vector of functions whose analysis has been modified and a vector of functions that are newly discovered.

static void BPatchCodeOverwriteBeginCallback (std::vector<BPatch\_basicBlock\*> &overwriteLoopBlocks);

This callback allows the user to remove any instrumentation as the program starts writing to a code page, which may be desirable as instrumentation cannot be removed during the overwrite loop’s execution, and any breakpoint instrumentation will dramatically slow the loop’s execution. Only invoked if hybrid analysis mode is set to BPatch\_malwareMode.

static void BPatchCodeOverwriteEndCallback(std::vector<Address> &deadFuncAddrs, std::vector<BPatch\_function\*> &modifiedFuncs, std::vector<BPatch\_function\*> &newFuncs);

This callback delivers the effects of the overwrite loop when it is done executing. In many cases no code will have changed. This function is only called if the Dyninst’s hybrid analysis mode is set to BPatch\_malwareMode.

Execute at the signal handler’s entry point. If the handler shifts execution to un-analyzed code, this will cause the BPatchCodeDiscoveryCallback to be invoked.

BPatch\_process::removeDebuggingArtifacts();

Remove debugging artifacts. This function invoked automatically if hybrid analysis mode is set to BPatch\_malwareMode, but there is no harm in calling it twice.

bool BPatch\_flowGraph::isValid();

When a function’s analysis is updated, its BPatch\_function object remains valid, but its BPatch\_flowGraph does not. A new CFG can be obtained from the BPatch\_function when the CFG is found to be invalid.

enum {BPatch\_noInterp, BPatch\_interpAsTarget, BPatch\_interpAsReturnAddr} BPatch\_stInterpret;

BPatch\_stopThreadExpr(BPatch\_snippet \*calculation, BPatch\_stInterpret interp = BPatch\_noInterp, bool useCache = false);

Interpret the calculation as an address, which will cause addresses corresponding to relocated code and Dyninst’s instrumentation to be translated into an unrelocated address.