Toward the Deconstruction of Dyninst

Giridhar Ravipati* Andrew R. Bernat* Nate Rosenblum* Barton P. Miller* Jeffrey K. Hollingsworth†

*Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685
{giri,bernat,nate,bart}@cs.wisc.edu

†Department of Computer Science
University of Maryland
College Park, MD 20742
hollings@cs.umd.edu

Abstract

There are two problems that hinder the development of binary tools: a lack of code sharing and a lack of portability. Binary tools, whether static or dynamic, depend on similar analysis and apply similar modification techniques. However, implementations of these techniques are not shared between tools, forcing a developer to reinvent the wheel rather than leverage existing functionality. Tools are often limited to a small range of platforms, preventing users from using existing tools on new platforms.

This research describes the deconstruction of the Dyninst dynamic instrumentation library. Dyninst possesses powerful analysis and modification techniques, but these capabilities are hidden underneath the instrumentation-focused API. We are creating a suite of component libraries that will provide a platform-independent interface to a core piece of Dyninst functionality. These libraries will allow other tool developers to access the capabilities of Dyninst. In addition to a programmatic interface, we are also creating representations for interchange of data between tools. The deconstruction of Dyninst poses several interesting challenges, including the design of general interfaces between components that are also abstract and portable.

This paper lays out our approach to the deconstruction of Dyninst and also describes the first of these components, the SymtabAPI, a multi-platform library for parsing, modifying, and exporting symbol tables in object files. We also present a case study of the tool ***unstrip*** *that regenerates missing or incorrect symbol tables in binaries. Unstrip allows other analysis, reverse engineering, and debugging tools, which require symbol information, to operate successfully on stripped binaries.*

1. INTRODUCTION[1]

Binary code analysis and editing tools are becoming increasingly common. These tools are used to provide information about a binary program's content and structure and to modify the program. Binary code analysis and editing are used in a wide variety of applications, including reverse engineering [6,17], cyber-forensics [19,22], program tracing [1], debugging [30], program testing [18,21,24], performance modeling [15,25], and performance profiling [2,5,16]. However, developing a binary tool is an expensive process, due to both the complexity of operating directly on a binary and the myriad number of platforms in common use.

Binary tools rely on two categories of operations: *analysis* and *modification*. Analysis is used to derive semantic meaning from the binary code ranging from identifying basic blocks and functions to detailed data flow analysis. Modification allows tools to inject new code into a program or modify existing code. The combination of analysis and modification provides a powerful synergy, in which analysis can reduce the amount of required instrumentation, and modification can augment the quality of analysis.

The DyninstAPI [4, 10] is a library that provides a high-level, platform-independent interface to dynamic binary analysis and modification. The goal of the DyninstAPI is to simplify binary tool development. Three factors have contributed to the success of the DyninstAPI. First, its combination of analysis and modification capabilities. Dyninst performs extensive static analysis on binaries, building up a complete representation that the user can use to guide instrumentation. Second, its abstract interface. The DyninstAPI poses instrumentation as inserting a *snippet* (an abstract, machine-independent representation of the desired code) at a *point* in the program. This simple interface enables a wide range of complicated operations, including performance analysis, memory tracing, and code coverage, to be performed with minimal complexity of user code. Third, its portability to multiple platforms, including Linux, Windows, AIX, and Solaris, and multiple architectures, including IA-32, AMD-64, IA-64, POWER, and SPARC.

Portability and abstraction are powerful concepts that allow a user to rapidly develop effective tools for a variety of platforms. Due to the increasing number of different computing systems, portability has become a critical concern of the tool community. Abstraction in a binary tool has two purposes. First, an abstract user

1. This work is supported in part by Department of Energy Grants DE-FG02-93ER25176 and DE-FG02-01ER25510, and National Science Foundataion grant 0627501.

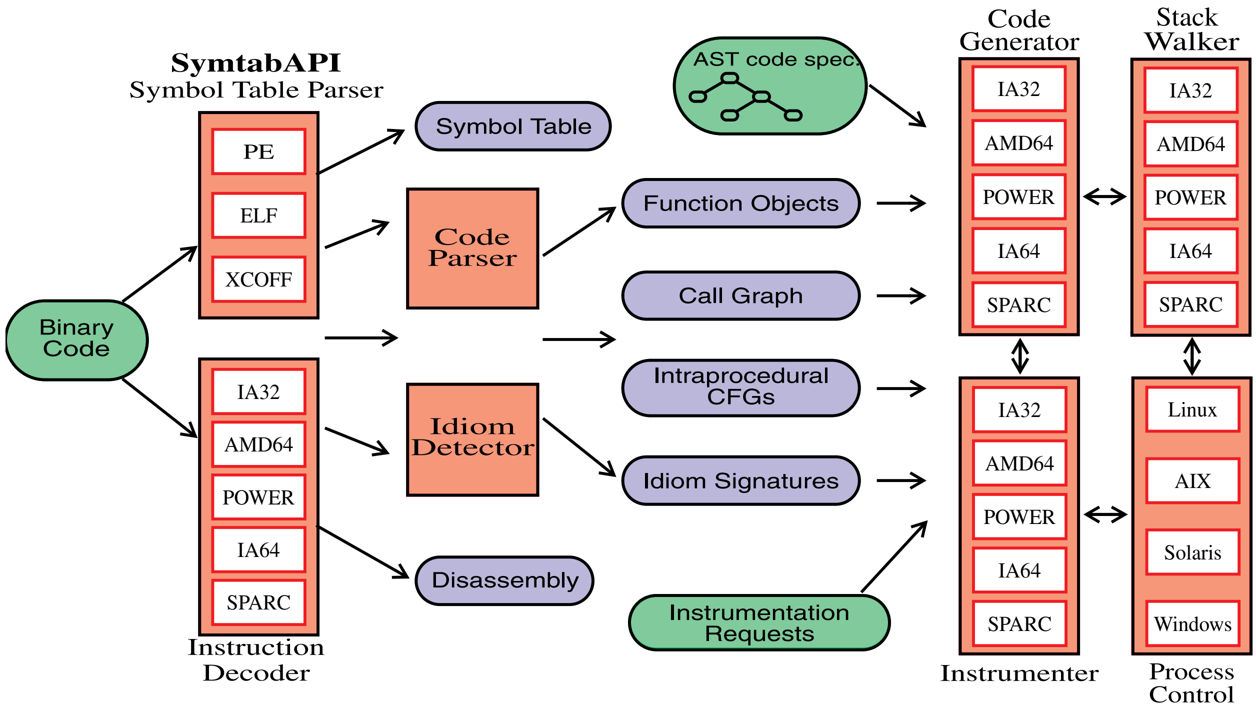

Figure 1: The Deconstructed Dyninst Internal Components

Rectangles represent the proposed components: symbol table parsing, instruction decoding, code parsing and idiom detection, code generation, instrumentation, stack walking, and process control; we have shown the initially supported platforms and formats for each component as well. Ovals represent inputs to the system and products of the parsing and analysis libraries.

interface hides much of the complexity of binary tool building, enabling users to easily create straightforward yet powerful tools. Second, pervasive use of abstract internal interfaces is a key requirement for portability, allowing platform-specific details to be encapsulated at low levels.

While the DyninstAPI is portable and abstract, it is not *general*. Dyninst has powerful analysis techniques, but these techniques support our modification capability rather than being presented on a standalone basis. Our static analysis techniques include regenerating function and control flow information from stripped binaries [9] and heavily optimized code. Our process state examination algorithms allow us to safely instrument and control multi-threaded code [31] and perform efficient stack walking.

Previously, many of these analysis capabilities were provided only in the context of instrumentation, and were inaccessible to tools that only require analysis capabilities. Recently there has been substantial interest in accessing the individual capabilities of Dyninst in support of new tool development. For example, a developer might want to statically parse and analyze a binary without executing it, or may want to only perform stack walking on a binary without inserting instrumentation. These capabilities are currently hidden behind the public Dyninst interface and have too many internal dependencies for easy re-use outside of Dyninst.

This paper describes our effort to deconstruct the monolithic Dyninst library into a suite of component libraries. Each library will provide a platform-independent interface to a core piece of Dyninst functionality, allowing other tools to benefit from our efforts. Thekey issue for this effort is to provide general interfaces while maintaining portability and an abstract interface. Currently each component provides the necessary features for Dyninst; we will generalize these interfaces to provide other tools with a comprehensive set of low-level, portable libraries. In addition, there is a substantial engineering effort required to cleanly isolate each component of Dyninst.

This paper lays out our approach to the deconstruction of Dyninst and also describes the first of these components, the SymtabAPI, a library for parsing symbol tables and section information in object files. This deconstruction has four benefits. First, users will be able to incorporate these libraries in their tools, gaining access to the desired capabilities of Dyninst. Second, all analysis products will be exportable in a standardized format to encourage interoperability between libraries and tools. Third, deconstruction will make Dyninst itself more flexible and modular. For example, it would be possible to create a static binary rewriter by combining the appropriate components. Finally, this deconstruction will substantially improve our ability to test Dyninst for correctness.

We also present a case study of the tool **unstrip**, which regenerates missing or incorrect symbol information in binaries. Adding symbol table information allows reverse engineering and analysis with tools that would otherwise be incapable of operating. Unstrip also provides an example of the power of the SymtabAPI: it consists of 221 lines of code and was implemented in one programmer-day.

Our work is proceeding in four stages. In the first stage, we have identified the components that will form the core libraries, as shown in Figure 1. Second, we will refine and generalize these internal interfaces to be more widely applicable. The capabilities of these internal modules are currently targeted to the requirements of Dyninst; we will develop the appropriate extensions necessary for generalized use. Third, we will rebuild Dyninst on top of the suite of components as an example tool. Fourth, we will use these components to build new tools, such as a multiplatform static rewriter.

The rest of this paper is organized as follows. In Section 2 we discuss related work. In Sections 3 and 4 we discuss the design and interface of the SymtabAPI. In Section 5 we describe **unstrip**, a tool for reconstructing symbols tables from stripped binaries. This tool demonstrates the synergy between the analysis capabilities of Dyninst and symbol table creation capabilities of the SymtabAPI. In the final section we provide a short summary of our work.

2. Related Work

The components that are being created by deconstructing the DyninstAPI cover two major areas: code *analysis* and code *modification*. Both analysis and modification are critical requirements for binary tools, and several projects have developed different analysis and modification techniques.

The IDA Pro disassembler [11] provides strong parsing and analysis capabilities, and shares several characteristics with our disassembly and idiom detection components. IDA Pro disambiguates code from data in stripped binaries through a combination of a static call graph traversal and pattern matching. However, IDA Pro relies heavily on user intervention to produce accurate results. Our algorithms are designed to have almost zero false positive code identifications while minimizing false negatives; these analyses are crucial to handling large scale and high volumes of stripped binaries [9].

The GNU project provides the binutils [8] package that provides a platform-independent interface to low-level binary file operations. In particular, the libbfd library provides symbol table and section parsing capabilities. However, there are substantial differences in the approach used by the SymtabAPI and libbfd. The libbfd interface provides the user a set of data structures representing a subset of the information present in the object; this subset is limited to data common between all file formats supported by libbfd. The remaining information is kept internal and cannot be accessed by the user. In addition, any querying, processing, or updating of those symbols must be done in user code. This approach is effective for a compiler environment in which existing symbols are copied, but less useful for binary tools that require all available symbol table data. The SymtabAPI provides a symbol type that is a superset and includes all possible information. Furthermore, common operations, such as searching and updating, are provided as library functions.

The instrumentation techniques used by the DyninstAPI are similar to those used by static binary rewriters, such as the the ATOM [26], DIABLO [7], EEL [12], Etch [23], and Vulcan [27] binary rewriters. However, these rewriters either require substantial symbol information about a binary or are limited to a particular platform. The required information ranges from relocation data [26] to completely managed code [7, 23, 27]. In contrast, Dyninst uses opportunistic algorithms that can take advantage of, but do not require, this information; we can operate successfully on completely stripped binaries with no symbol information. The EEL rewriter does not require supplementary information, but operates only on SPARC binaries.

Dynamic instrumentation tools such as PIN [13], Valgrind [20], DIOTA [14], and Dynamo [3] follow the control flow of the program, analyzing and instrumenting basic blocks and instruction traces immediately prior to their execution. This approach obviates the need for static analysis and easily distinguishes executed code from data. However, these tools do not provide detailed binary analysis. In contrast, Dyninst provides the user with detailed control flow information. This information (e.g., functions, variables, and control flow graphs) allows the user to gather the same information with less instrumentation, leading to lower overall overhead. For example, code coverage can be made more efficient by incorporating dominator analysis [29].

3. SymtabAPI

The first component of the deconstructed DyninstAPI is the SymtabAPI, a library for parsing symbol tables and object file headers. The design of the SymtabAPI library leverages the experience we have gained in the design, implementation, and use of the DyninstAPI. This includes the design concepts of portability and abstraction. In addition, *extensibility* and *generality* are critical requirements. Unlike the DyninstAPI, the SymtabAPI will be incorporated into existing tools that have pre-existing requirements; our design must be flexible enough to fulfill these requirements.

Abstraction: The SymtabAPI interface must be platform-independent, providing an abstract view of binaries and libraries across multiple platforms. An abstract interface provides two benefits: it simplifies the development of a tool since the complexity of a particular file format is hidden, and it allows tools to be easily ported between platforms.

Interactivity: The user must be able to use the library incrementally, updating symbol information with the results of further analysis or user input. Tools frequently use more sophisticated analyses to augment the information available from the binary directly; it should be possible to make this extra information available to the SymtabAPI library. An example of this is a tool operating on a stripped binary. Although the symbols for the majority of functions in the binary may be missing, they can be determined via more sophisticated analysis. In our model, the tool would then inform the SymtabAPI library of the presence of these functions; this information would be incorporated and available in the future. In addition, it should be possible to emit this updated information into a new symbol table stored in the binary for future use by other tools.

Extensibility: The data structures used by the library should be user extensible to contain extra data needed by the tool builder. Non-extensible structures simply force tools to copy information into their own internal structures, which is inefficient. Our approach allows tools to extend the SymtabAPI structures to fit their own requirements.

Generality: The SymtabAPI interface should be able to fulfill the requirements of a wide variety of tool designs. In particular, we are interested in three aspects of binary parsing. First, the SymtabAPI should be able to parse the common ELF, PE, and XCOFF file formats. Second, the library should be able to operate on binaries located either on disk or in memory. Aspects of the binary may be complete only after being loaded into memory (e.g., inter-library addresses on AIX), or a binary image may exist only in memory (e.g., the vsyscall page on Linux). Third, the library should operate both on executables (a.out/.exe) and libraries (.so/.dll) interchangeably. Programs are becoming increasingly modular, and as a result transparent operation on both the executable and libraries is critical.

The SymtabAPI provides the following abstractions: symbols, modules, archives, relocations, and exception blocks. Symbols identify functions and variables within the binary object. Modules represent a particular source file in cases where multiple files were compiled into a single binary object; if this information is not present, we use a single *default* module. Archives represent a collection of binary objects stored in a single file (e.g., a static archive). Relocations provide the necessary information to move symbols within an object. Exception blocks contain the information necessary for run-time exception handling. Details of these abstractions are discussed in the next section.

Parsed information is kept in an *internal store*. Information from the store is accessed through a query-based interface; users provide the desired characteristics to match (e.g., name, type, or address) and the library returns all matching abstractions. These abstractions can then be updated by the user and new queries performed; any changes are merged back into the store and returned in future lookups. In addition, entirely new abstractions can be added to supplement the information derived from the binary. Finally, the internal store can be exported, either as a new symbol table or as a separate file.

The query/update/export cycle provides a powerful capability: the ability to regenerate an incomplete or missing symbol table from information gathered via analysis or user intervention. A tool could use heavy-weight analysis to identify likely functions without requiring symbol information [9]. These functions could then be labelled with semantically meaningful names through manual interaction. Finally, a new symbol table would be emitted, including the missing function symbols.

All SymtabAPI abstractions are designed to be extensible, allowing users to annotate particular abstractions with tool-specific data. We believe that common formats will emerge for these annotations, allowing tools to share information. We intend to develop new section types to store these additional annotations in the binary itself.

The current implementation supports the ELF (IA-32, IA-64, AMD-64, POWER, and SPARC), XCOFF (POWER), and PE (Windows) object file formats. We use a canonical format to internally represent the binary. This format abstracts away platform-specific details, leaving the interface to the SymtabAPI platform-independent. The canonical format consists of three components: a *header* block that contains general information about the object (e.g., its name and location), a set of *symbol lists* that index symbols within the object for fast lookup, and a set of *additional data* that represents additional information that may be present in the object (e.g., relocations or exception information). Adding a new format requires no changes to the interface and hence will not affect any of the tools that use the SymtabAPI. In addition, we provide an external name *demangling* interface that allows the SymtabAPI to access a compiler-specific demangling library.

This design fulfills our four requirements of abstraction, interactivity, extensibility, and generality. The SymtabAPI provides abstractions that can be modified or extended on-the-fly, allowing tools to supplement and extend the symbol table information available. Our abstract interface to symbol table parsing enables tools to be easily ported to different file formats and architectures, including new file formats. Anecdotally, adding support for the POWER/ELF format used by BlueGene/L required less than one hour of effort.

4. INTERFACE

This section presents a summary of the operations supported by the API, details of the SymtabAPI interface and some code snippets to show how the API is used. The basic operations supported by the library are:

- *Parsing the symbols in a binary, either on disk or in memory* - The binary object must be parsed before further operations can be performed. Parsing maps a provided file *identifier* to a SymtabAPI *handle*, represented by a Dyn_Symtab object pointer. The identifier can either be a full path name to a file on disk or a pointer to the beginning of a file in memory. Parsing creates object representations of all symbols, relocations, and exception blocks in the object. The address and size of all objects are derived directly from the symbol table, as are mangled symbol names. Demangled (pretty) and typed names are computed through an external *demangling* interface; this interface allows the SymtabAPI to leverage existing demangling libraries (e.g., the GNU libiberty [8]).
- *Querying for symbols* - Queries can be made based on name, symbol type (e.g., function or variable), or address, and return a vector of Dyn_Symbol objects. Figure 2 shows example source code for parsing a file and finding all symbols named “main”.
- *Updating existing symbol information* - This is helpful in making corrections to or adding aliases to already existing symbols. The desired symbol is identified via a query, and the appropriate object is modified directly. Figure 4 shows an example of an update, with the user adding an alias for function foo and correcting its size.
- *Adding new symbols* - The user can create new symbols and add them to the existing symbol lists. These new symbols are available for further data queries. Figure 5 shows an example of adding a new symbol. These symbols can either be derived from binary analysis (e.g., of a stripped binary) or provided directly by a user.
- *Exporting symbols* - The user can export the parsed information, either *saving* data to a separate file or *emitting* an entirely new symbol table. This capability is useful in two ways. First, the saved data can be used for off-line analysis of the binary. Second, emitting a new symbol table will allow the SymtabAPI to interoperate with other tools that use their own symbol table parsing techniques. For example, a tool could analyze a stripped binary and recreate the missing symbol information; this information would then allow other tools that depend on a symbol table to operate.
- *Getting relocation/exception information* - The user can either request all relocation and exception data, or query for particular data based on an address.
- *Getting header information* - Information about the object file can be obtained by using the handle. This information includes the name of the object, the number of symbols in the object, or information about sections (e.g., text or initialized data) within the object.

4.1. CLASS INTERFACE

The SymtabAPI follows the style of the DyninstAPI, providing an object-based C++ interface. The primary classes are Dyn_Symtab, Dyn_Module, Dyn_Symbol, and Dyn_Archive. In addition, we represent exception information in the Dyn_ExceptionBlock class. We briefly describe each of these classes; complete details of the API can be found in the SymtabAPI Programmer’s Guide [28].

Dyn_Symtab: The Dyn_Symtab class represents a single object and provides a handle to the information contained in the object. This class contains all symbols and additional file-level information (e.g., relocations or exception tables). Access to this information is provided through query methods of the class.

Dyn_Module: This class represents a single *source file* in cases where the object was compiled from multiple source files (and this information is present in the symbol table). If this module information is not present, all symbols are assigned to a *default* module. Modules provide a mechanism for distinguishing between multiple symbols with identical names. This can occur if symbols were unique within their respective source files, but not in the binary (e.g., static functions in C). However, module information is frequently discarded by the compiler and its presence is not assumed.

Dyn_Symbol: This class represents a symbol in the object file. Symbols commonly represent functions and variables, although other information may also be encoded in this fashion. This class contains the information about the symbol, which includes its names (man-

```
// Name the object file to be parsed:
std::string file = "libfoo.so";
// Declare a pointer to an object of type Dyn_Symtab; this represents the file.
Dyn_Symtab *Obj = NULL;
// Parse the object file
if (!Dyn_Symtab::openFile(file, Obj)) {
    // Error in parsing; the code can query the error and perform the necessary actions.
    switch (Dyn_Symtab::getLastError())

            ...
}
// Create a vector to contain lookup results
std::vector<Dyn_Symbol *> syms;
// Search for all symbols with demangled name "main" and type any (ST_UNKNOWN)
Obj->findSymbolByType(syms, "main", Dyn_Symbol::ST_UNKNOWN);
```

Figure 2: Example of code parsing and symbol lookup

A simple example of parsing and lookup. First, the object "libfoo.so" is parsed as a file on disk. The error handling code is omitted for simplicity. Next, the resulting handle is used to find all symbols matching "main"; these symbols are added to the syms vector.

```
//Name the object file to be parsed
std::string file = "libfoo.a"
// Declare an pointer to an object of type Dyn_Archive. This represents the archive.
Dyn_Archive *archive = NULL;
//Declare an pointer to an object of type Dyn_Symtab. This represents the object.
Dyn_Symtab *Obj = NULL;
//Try to parse the archive file; error handling omitted for simplicity
Dyn_Archive::openArchive(file, archive);
// Get the Dyn_Symtab object for the desired object
archive->getMember("bar.o", Obj);
// Obj now contains a handle to the desired binary object "bar"
```

Figure 3: Example of parsing an archive file

This example shows how an object can be accessed from an archive in a two-step process. First, the archive libfoo.a is parsed off disk; this creates an internal list of all objects within the archive. Next, the desired object "bar.o" is accessed.

```
// Obj represents a handle to a parsed object file.
std::vector <Dyn_Symbol *> syms;
// search for symbol of any type with demangled (pretty) name "main".
if (Obj->findSymbolByType(syms, "main", Dyn_Symbol::ST_UNKNOWN)) {
    // change the type of the found symbol (most of the attributes can be changed).
    syms[0]->setType(Dyn_Symbol::ST_OBJECT);
    // Add a new demangled (pretty) name to the found symbol.
    syms[0]->addPrettyName("_main");
    // These changes are automatically added to the data store; no further actions are
    // required by the user.
}
```

Figure 4: Updating a symbol

Updating a symbol is a straight forward process. First, locate the desired symbol via the query interface. Next, update the desired data fields, which can include the names, type, address, or size.

```
// Obj represents a handle to a parsed object file.
// Create an entirely new function symbol
Dyn_Symbol *newSym = new Dyn_Symbol("func1", // Name of the new symbol
                                  "DEFAULT_MODULE", // Its source module (default)
                                  Dyn_Symbol::ST_FUNCTION, // Its type (function)
                                  Dyn_Symbol::SL_GLOBAL, // Scoping (global)
                                  123456); // And address (from start of text section)
// Add the newly created symbol to the data store; we omit error handling for clarity
Obj->addSymbol(newSym);
// The new symbol will be included in any future searches; assert this is true
assert(Obj->findSymbolByType(syms, "func1", Dyn_Symbol::ST_UNKNOWN) == true);
```

Figure 5: Adding a symbol

Adding a symbol is done directly. First, a new object representing the symbol is created. Next, the object is added to the handle corresponding to the appropriate object. The new symbol becomes part of the data store and is included in future queries.

gled, demangled (pretty), and the full typed prototype), the offset within the file and the symbol size, the module the symbol is associated with, and its type (e.g., function or variable). In addition, this class provides methods to update symbol information.

Dyn_Archive: This class represents an *archive*, a single file that is a collection of several objects. Archives are a common representation for static libraries. The XCOFF format also allows archives to contain dynamic libraries. The Dyn_Archive object provides a container for all of the binary objects within the file, and individual Dyn_Symtab handles can be accessed either by name or as a group. An example of code for handling archives is shown in Figure 3.

Dyn_ExceptionBlock: This class represents an exception block in the object. This class gives all the information pertaining to that exception block such as the try block start address and size and the catch block start address.

In addition to the symbol information, files contain additional data such as header information and relocation information. We are in the process of creating object representations of this information.

5. UNSTRIP - A CASE STUDY

Executables and libraries may contain a symbol table that identifies objects (e.g., functions, sections, and variables) within the binary. However, much of this information is not necessary for the proper execution of a program. Stripped binaries have had varying amounts of their symbol table removed; this may range from the removal of symbols representing internal functions to the removal of the entire table. This operation is performed for reasons ranging from resisting analysis and reverse engineering to reducing disk space requirements. Common examples of stripped binaries include malicious programs, proprietary programs, and system utilities and libraries. In addition to removing symbol table information, the information may be present but incorrect; this often occurs due to compiler bugs.

The information present in the symbol table is useful both for analysts and automated analysis techniques. Analysts benefit from the names given to functions and variables within the program; a common obfuscation technique is to render these names meaningless [cite]. Tools often require symbol table information to identify the locations of functions within the binary; without this information, the capabilities of the tools are limited [cite].

As an example of the power and flexibility of the SymtabAPI, we have created a tool, called **unstrip**, that regenerates removed symbol table information in binaries. It does so by combining our stripped binary parsing techniques[9] with the symbol table creation capabilities of the SymtabAPI. By building on the pre-existing capabilities of the DyninstAPI, we were able to write unstrip in one programmer-day; the tool consists of 221 lines of code.

5.1. KEY FACTORS

Implementing unstrip was a simple process because we were able to leverage both the symbol table editing capabilities of the SymtabAPI and the analysis capabilities of the DyninstAPI. In particular, the following key factors were critical to the effort:

- *Interactivity* - SymtabAPI provides a powerful functionality to users whereby they can incrementally update existing symbol information or add new symbols found as part of analysis to the library. For unstrip, we were able to create and add new symbols as they were found by analysis. The tool is completely automated, and names functions by the address of their entry point. It would be straightforward to create an interactive version that allowed a user to provide their own additional names for functions.
- *Exportability* - While several tools are capable of extracting function information from stripped binaries, to our knowledge there are no tools that can produce a new binary that includes this information for use by other tools. We perform this operation by using the SymtabAPI's capability to generate a new symbol table; this new symbol table is then included in the final binary.
- *Sophisticated Analysis* - The DyninstAPI performs sophisticated binary analysis to enable its dynamic instrumentation techniques [9]. These techniques identify functions within stripped binaries by combining static control flow analysis and heuristic function identification methods. A critical part of the deconstruction effort is to separate these analysis techniques into a separate library. The unstrip tool currently uses the public DyninstAPI interface, although future versions will use the code analysis library directly.

5.2. METHOD

The unstrip tool operates in the following phases:

1) Start the stripped binary under Dyninst. As part of its start-up phase, the DyninstAPI parses and analyzes a binary to build control flow graphs of all functions in the program. If we detect gaps in the parsed areas of the binary, we apply a variety of heuristic techniques to determine if functions reside within these gaps; if so, we add the newly parsed areas to the list of known functions.

2) Extract function information from Dyninst via the public API.

3) Create new symbols for all discovered functions, e.g. "func0x1000". We prepend a unique identifier to the entry address of the function to ensure that no symbols share a common name.

4) Insert this information back into the SymtabAPI.

5) Cause the SymtabAPI to generate a new binary that includes these symbols.

The unstrip tool can regenerate symbol table information for any format supported by the SymtabAPI, including ELF and XCOFF; support for writing the PE format used by Windows is under active development. It runs on any platform supported by the DyninstAPI, including Linux (IA-32 and IA-64), AIX (POWER), Solaris (SPARC), and Windows (IA-32).

5.3. Discussion

The code necessary for extracting function information from Dyninst took half a programmer-day to write and consisted of under 50 lines of code; the code necessary to create symbols and generate a new symbol table took an additional half programmer-day and brings the total lines of code to 221.

A key to the success of the DyninstAPI is its abstract, platform-independent interface. The unstrip tool leverages this interface; the operations performed by unstrip appear straightforward because the complexity has been hidden in the DyninstAPI and SymtabAPI.

6. Conclusions

The broad goal of this work is to deconstruct the Dyninst library into a suite of component libraries that provide program analysis and instrumentation capabilities for tool builders. Each library will provide a platform-independent interface to a specific tool need, such as parsing a binary. We have provided a general overview of this deconstruction and described the first of these libraries, the SymtabAPI symbol table parsing library. The SymtabAPI library provides an abstract, interactive symbol table parsing and update capability. The 1.0 version of the SymtabAPI library has been released with the version 5.1 release of Dyninst, and will assist in the creation and porting of binary tools. We also describe the tool **unstrip**, which regenerates stripped symbol tables. The unstrip tool allows tools which require symbol table information to be applied to stripped binaries.

Our ongoing work consists of extending the SymtabAPI, and developing interfaces for remaining components of Dyninst. We expect this effort to provide significant benefits to the tool community in the form of making tools more portable, encouraging common data interchange formats, and increasing code reuse.

The SymtabAPI 1.0 library can be found at *http://www.paradyn.org/html/symtab1.0-features.html.*

7. References

[1] Ball, T. and Larus, J.R. "Optimally Profiling and Tracing Programs", *Nineteenth ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, USA, 1992.

[2] Bernat, A. and Miller, B.P., "Incremental Call-Path Profiling", *Concurrency: Practice and Experience*, to appear, 2006.

[3] Bruening, D., Garnett, T. and Amarasinghe, S. "An Infrastructure for Adaptive Dynamic Optimization", *First Annual International Symposium on Code Generation and Optimization*, San Francisco, California, USA, March 2003.

[4] Buck, B.R, and Hollingsworth, J.K., "An API for Runtime Code Patching", *Journal of High Performance Computing Applications* **14**, 4, Winter 2000.

[5] Cain, H., Miller, B.P., and Wylie, B. "A Callgraph-Based Search Strategy for Automated Performance Diagnosis", *Euro-Par 2000*, Munich, Germany, August 2000.

[6] Cifuentes, C. and Gough, K.J. "Decompilation of Binary Programs", *Software - Practice and Experience* 25(7):811-829, 1995.

[7] De Bus, B., De Sutter, B., Van Put, L., Chanet, D., and De Bosschere, K., "Link-time Optimization of ARM Binaries", *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools,* Washington DC, USA, June 2004.

[8] GNU Tools, http://www.gnu.org/software/binutils.

[9] Harris, L. and Miller, B.P., "Practical Analysis of Stripped Binary Code", *Workshop on Binary Instrumentation and Applications*, St. Louis, Missouri, USA, September 2005.

[10] Hollingsworth, J.K., Miller, B.P., Goncalves, M., Naim, O., Xu, Z., and Zheng, L. "MDL: A Language and Compiler for Dynamic Program Instrumentation", *Parallel Architectures and Compilation Techniques*, San Francisco, California, November 1997.

[11] IDA Pro, http://www.datarescue.com/idabase/overview.htm.

[12] Larus, J.R. and Schnarr, E., "EEL: Machine Independent Executable Editing", *ACM Conference on Programming Language Design and Implementation (PLDI),* La Jolla, California, USA, June 1995.

[13] Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., and Hazelwood, K. "Pin: building customized program analysis tools with dynamic instrumentation", *ACM Conference on Programming Language Design and Implementation (PLDI)*, Chicago, Illinois, USA, June 2005.

[14] Maebe, J., Ronsse, M. and De Bosschere, K. "DIOTA: Dynamic Instrumentation, Optimization and Transformation of Applications", *PACT'02: International Conference on Parallel Architectures and Compilation Techniques*, Charlottesville, Virginia, USA, 2002.

[15] Marathe, J., Mueller, F., Mohan, T., De Supinski, B., McKee, S., and Yoo, A. "METRIC: Tracking Down Inefficiencies in the Memory Hierarchy via Binary Rewriting", *International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, San Francisco, California, USA, 2003.

[16] Miller, B.P., Callaghan, M., Cargille, J., Hollingsworth, J., Irvin, R.B., Karavanic, K., Kunchithapadam K., and Newhall T., "The

Paradyn Parallel Performance Measurement Tool", *IEEE Computer* **28**, 11, November 1995: 37-46. Special issue on performance evaluation tools for parallel and distributed computer systems.

[17] Miller, B.P., Christodorescu, M., Iverson, R., Kosar, T., Mirgorodskii, A., and Popvici, F. "Playing Inside the Black Box: Using Dynamic Instrumentation to Create Security Holes", *Parallel Processing Letters* **11**, 2/3, June/September 2001.

[18] Mirgorodskiy, A., Maruyama, N., and Miller, B.P., "Problem Diagnosis in Large-Scale Computing Environments" *Supercomputing '06*, Tampa, Florida, USA, November 2006.

[19] Newsome, J., Brumley, D., and Song, D. "Vulnerability-Specific Execution Filtering for Exploit Prevention on Commodity Software", *13th Annual Network and Distributed System Security Symposium (NDSS '06)*, San Diego, California, USA, February 2006.

[20] Nethercote, N. and Seward, J. "Valgrind: A Program Supervision Framework", *Electronic Notes in Theoretical Computer Science,* 2003.

[21] O'Callahan, R., and Choi, J. "Hybrid Dynamic Race Detection", *ACM Symposium on the Principles and Practices of Parallel Programming (PPoPP)*, San Diego, California, USA, June 2003.

[22] Prasad, M. and Chiueh, T. "A Binary Rewriting Defense Against Stack Based Buffer Overflow Attacks", *USENIX Annual Technical Conference*, San Antonio, Texas, USA, June 2003.

[23] Romer, T., Voelker, G., Lee, D., Wolman, A., Wong, W., Levy, H. and Bershad, B. "Instrumentation and Optimization of Win32/Intel Executables Using Etch", *USENIX Windows NT Workshop*, Seattle, Washington, USA, 1997.

[24] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. "Eraser: A Dynamic Race Detector for Multithreaded Programs", *ACM Transactions on Computers* **15**, 4, 1997.

[25] Snavely, A., Gao, X., Leea, C., Carrington, L., Wolter, N., Labarta, J, Gimenez, J., and Jones, P. "Performance Modeling of HPC Applications", Dresden, Germany, August 2003.

[26] Srivastava, A. and Eustace, A., "ATOM: A System For Building Customized Program Analysis Tools", *ACM Conference on Programming Language Design and Implementation (PLDI)*, Orlando, Florida, USA, June 1994.

[27] Srivastava, A., Edwards, A. and Vo, H. "Vulcan: Binary Transformation in a Distributed Environment", Technical Report MSR-TR-2001-50, 2001.

[28] SymtabAPI Programming Guide, http://paradyn.org/symtabAPI/release-1.0/ProgGuide.html

[29] Tikir, M., and Hollingsworth, J. "Efficient Instrumentation for Code Coverage Testing", *International Symposium on Software Testing and Analysis (ISSTA)*, Rome, Italy, July 2002.

[30] Totalview, http://www.etnus.com.

[31] Xu, Z., Miller, B.P., and Naim, O. "Dynamic Instrumentation of Threaded Applications", *7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, Georgia, May 1999.