Binary Code Extraction and Interface Identification for Security Applications

Juan Caballero[§†] Noah M. Johnson[†] Stephen McCamant[†] Dawn Song[†]
[†]UC Berkeley [§]Carnegie Mellon University

Abstract

Binary code reuse is the process of automatically identifying the interface and extracting the instructions and data dependencies of a code fragment from an executable program, so that it is self-contained and can be reused by external code. Binary code reuse is useful for a number of security applications, including reusing the proprietary cryptographic or unpacking functions from a malware sample and for rewriting a network dialog. In this paper we conduct the first systematic study of automated binary code reuse and its security applications.

The main challenge in binary code reuse is understanding the code fragment's interface. We propose a novel technique to identify the prototype of an undocumented code fragment directly from the program's binary, without access to source code or symbol information. Further, we must also extract the code itself from the binary so that it is self-contained and can be easily reused in another program. We design and implement a tool that uses a combination of dynamic and static analysis to automatically identify the prototype and extract the instructions of an assembly function into a form that can be reused by other C code. The extracted function can be run independently of the rest of the program's functionality and shared with other users.

We apply our approach to scenarios that include extracting the encryption and decryption routines from malware samples, and show that these routines can be reused by a network proxy to decrypt encrypted traffic on the network. This allows the network proxy to rewrite the malware's encrypted traffic by combining the extracted encryption and decryption functions with the session keys and the protocol grammar. We also show that we can reuse a code fragment from an unpacking function for the unpacking routine for a different sample of the same family, even if the code fragment is not a complete function.

1 Introduction

Often a security analyst wishes to reuse a code fragment that is available in a program's binary, what we call *binary code reuse*. For example, a piece of malware may use proprietary compression and encryption algorithms to encode the data that it sends over the network and a security analyst may be interested in reusing those functions to decode the network messages. Further, the analyst may be interested in building a network proxy that can monitor and modify the malware's compressed and encrypted protocol on the network. Also, for dialog rewriting [24, 29] if some field of a network protocol is changed, other dependant fields such as length or checksum fields may need to be updated. If those fields use proprietary or complex encodings, the encoding functions can be extracted and deployed in the network proxy so that the rewritten message is correctly formatted. Another application is the creation of static unpacking tools for a class of malware samples [17]. Currently, creating a static unpacker is a slow, manual process. Frameworks have emerged to speed the process [15], but a faster approach would be to extract the unpacking function from the malware sample and reuse it as a static unpacker.

At the core of these and other security applications is binary code reuse, an important problem for which current solutions are highly manual [1, 5, 6, 30]. In this paper we conduct the first systematic study of *automatic binary code reuse*, which can be defined as the process of automatically identifying the interface and extracting the instructions and data dependencies of a code fragment from an executable program, so that it is self-contained and can be reused by external code. Reusing binary code is useful because for many programs, such as commercial-off-the-shelf applications and malware, source code is not available. It is also challenging because binary code is not designed to be reusable even if the source code it has been generated from is. The main challenge of binary code reuse is to interface with the code fragment that we want to reuse. The code fragment may not have a function prototype available, for example because it was intended only for internal use, or it may

not even correspond to a function in the first place. Identifying a prototype for the binary code fragment enables reuse of the code by generating and passing appropriate inputs. In addition, we want to extract the code fragment itself, i.e., its instructions and data dependencies, so that it is self-contained and can be reused by other code, independently of the rest of the functionality in the program. The self-contained code fragment can easily be shared with other users and can be statically instrumented or rewritten, for profiling or to enforce a safety policy on its memory accesses if it is untrusted. To summarize, binary code reuse encompasses two tasks: identifying the interface of the code fragment and extracting the instructions and data dependencies of the code fragment so that it is self-contained.

Scope. Not all binary code can be reused. To reuse a binary code fragment, the fragment should have a clean interface and be designed to perform a specific well-contained task, mostly independent of the remaining code in the program. In this paper we mostly focus on reusing binary code fragments that correspond to functions at the source code level, what we call *assembly functions*, because in structured programming a function is the base unit of source code reuse. Functions are usually designed to perform an independent, well-contained task and have a well-defined interface, the *function prototype*. In addition, we show that a code fragment that does not correspond to a complete assembly function, but has a clean interface and performs a well-contained task, can also be reused.

Reusing an arbitrary assembly function can be extremely challenging because the function interface can be convoluted and the function can have complex side effects. Our approach handles common side effects such as an assembly function modifying one of its parameters or accessing a global variable, and also handles calls to internal and standard library functions. But we exclude functions with a variable-length argument list or functions that are passed recursive structures such as trees. We refer the reader to Section 2.3 for a more detailed description of the problem's scope. An important class of functions that we extract in this paper are *transformation functions*, which include encryption and decryption, compression and decompression, code packing and unpacking, checksums, and generally any function that encodes data. Such functions are usually well-contained, have clean interfaces, limited side effects, are interesting for many security applications, and can be identified using previously proposed techniques [24, 46].

Approach. The main challenge in assembly function reuse is identifying the interface of an assembly function and generating a prototype for it so that it can be reused by other source code. This is challenging because the *function parameters* that comprise the prototype are not explicitly defined in the binary code and also because they need to be expressed using variables and types, which do not exist in the binary code. Our approach uses dynamic analysis to extract a parameter abstraction at the binary level (an *assembly parameter*) and then translate the assembly parameters into the formal parameters in the function's prototype. To extract assembly parameters from a given execution trace, our approach first identifies the inputs and outputs for each function run, splits them into assembly parameters, identifies important attributes such as the parameter type (input, output, input-output) and the parameter location (register, stack, table), and finally combines this information across multiple function runs.

To extract the function's body, i.e., the instructions that constitute the assembly function, we use the observation that for reusing a binary code fragment a user often has no need to understand its inner workings. For example, a security analyst may want to reuse the proprietary cipher used by some malware, together with the session keys, to decrypt some data, without worrying about how the proprietary cipher works. For these applications, complex reverse-engineering or decompilation methods are not necessary to recover the function's body as source code. We can leverage the support of current C compilers for inline assembly [3, 11] and generate a function with a C prototype but an inline assembly body. To extract the function's body we use a combination of static and dynamic analysis that includes hybrid disassembly [41], symbolic execution [35], jump table identification [28], and type inference techniques.

Because the extracted binary code runs in the same address space as a program that uses it, the same security concerns apply to it as to an untrusted third-party library: a malicious extracted function might attempt to call other functions in memory or overwrite the application's data. If such attacks are a risk, an isolation mechanism is needed to limit what the extracted code can do. In this work we process the extracted code with a software-based fault isolation (SFI) tool to insert runtime checks that prevent the extracted code fragment from writing or jumping outside designated memory regions (separate from the rest of the program). We use PittSFIeld [39], an SFI implementation for x86 assembly code that enforces jump alignment to avoid overlapping instructions and includes a separate safety verifier.

We design and implement BCR, a tool that extracts code fragments from program binaries and wraps them in a C prototype, so they can be reused by other C code. We use BCR to extract the encryption and decryption routines used by two spam botnets: MegaD and Kraken. We show that these routines, together with appropriate session keys, can be reused by a network proxy to de-

crypt encrypted traffic on the network. Further, we show that the network proxy can also rewrite the malware's encrypted traffic by combining the extracted encryption and decryption functions with the session keys and the protocol grammar. To show that we can reuse code fragments that are not complete functions as long as the code fragments have a clean interface, we also extract the unpacking functions from two samples of Zbot, a trojan, and use an unpacking fragment from one sample as part of the routine to unpack the other sample.

Other applications. In addition to the applications that we examine in this paper, binary code reuse is useful for many other applications. For example, it can be used to automatically describe the interface of undocumented functions. It often happens that malware uses undocumented functions from the Windows API, which are not described in the public documentation [10]. Projects to manually document such functions [16] could benefit from our approach to automatically identify the interface of a binary code fragment. Extracted functions could also be useful in the development of programs that interoperate with other proprietary interfaces or file formats, by allowing the mixture of code extracted from previous implementations with re-implemented replacements and new functionality. Another application is to determine whether two pieces of binary code are functionally equivalent, for example to determine whether a vulnerability has been fixed in the most recent version. Recent work has addressed this issue at the source code level by fuzzing both pieces of source code and comparing the input-output pairs [34], but how to interface with a binary code fragment to perform such fuzzing is an open problem. Finally, a security analyst may want to fuzz a well-contained, security-sensitive function independently of the program state in which it is used.

Contributions.

- We propose a novel technique to identify the interface of a binary code fragment directly from the program's binary, without access to its source code. The interface captures the inputs and outputs of the code fragment and provides a higher level parameter abstraction not available at the binary level.
- We design an approach to automatically extract a code fragment from a program binary so that the code fragment is self-contained and can be reused by an external C program. The extracted code fragment can be run independently of the rest of the program's functionality, can be easily instrumented, and can be shared with other users. We implement BCR, a tool that uses our approach to automatically extract an assembly function from a given program binary.
- We reuse the encryption and decryption routines used by two spam botnets in a network proxy that can rewrite their encrypted C&C traffic, when provided with the session keys and the C&C protocol grammar. In addition, we extract the unpacking function from a trojan horse program, and show that a code fragment belonging to that function can be reused by the unpacking function for a different sample from the same family. Finally, we apply software-based fault isolation [39] to the extracted functions to prevent them from writing or jumping outside their own isolated memory regions.

2 Overview and Problem Definition

In this section we give an overview of the binary code reuse problem, formally define it, outline the scope of our solution, and present an overview of our approach.

2.1 Overview

Binary code reuse comprises two tasks: 1) identifying the interface of the code fragment and formatting it as a prototype that can be invoked from other source code; and 2) extracting the instructions and data dependencies of the code fragment so that it is self-contained and can be reused independently of the rest of the program's functionality.

The main challenge in binary code reuse is identifying the interface of the code fragment, which specifies its inputs and outputs. This is challenging because binary code has memory and registers rather than named parameters, and has limited type and semantic information, which must be converted into a high level prototype. It is also challenging because the code fragment might have been created by any compiler or written by hand, thus few assumptions can be made about its calling convention. In addition, the extracted code fragment needs to be self-contained, which in turn implies that we need a recursive process that extracts any function called from inside the code fragment that we want to extract (and from inside those callees), and that we need to account for the possible side effects from the code fragment and its callees. For example, we need to identify and extract the data dependencies such as global variables and tables that the code fragment uses.

Previous work on binary code reuse is highly manual [5, 6, 30]. As far as we know we are the first ones to systematically study automatic binary code reuse. Our goal is to automate the whole process, with a focus on automatically identifying the code fragment's interface. There are two different representations for the extracted binary code: decompiled source code [5, 30] and assembly instructions [6, 30]. In this work we use inline as-

sembly with a C prototype because inline assembly is the most accurate representation of the code (it represents what gets executed) and because decompilation is not needed for binary code reuse. The use of inline assembly limits portability to the x86 architecture, and requires compiler support, but the x86 architecture is still by far the most important architecture in security applications, and commonly used compilers include rich support for inline assembly [3, 11].

To reuse a binary code fragment, the code should have a clean interface and be designed to perform a well-contained task, relatively independent of the remaining code in the program. Otherwise, if the extracted code interface is not clean or the code performs several intertwined tasks and the user is only interested in one of them, it becomes difficult to separate the relevant code and interface with it. In structured programming, the above characteristics are usually associated with functions, which are the basic unit of (source) code reuse in a program and reduce the development and maintenance costs of a program by making the code modular. The interface of a function is captured by its *function prototype*.

The source-level concept of a function may not be directly reflected at the binary code level, since functions at the source level can be inlined, split into non-contiguous binary code fragments, or can exit using jumps instead of return instructions (e.g., due to tail-call optimizations). Despite this blurring, it is possible to define an *assembly function* abstraction at the binary level for which an extracted prototype gives a clean interface when the underlying functionality is well modularized. Thus, we focus on identifying the interface and extracting the *body* of such function abstractions, the details of which we turn to next.

2.2 Problem Definition

To reuse functions from program binaries, we first need a function abstraction that captures our definition of what a function is in binary code.

Function abstraction. We define a *basic block* to be a sequence of instructions with one entry point and one exit point. Basic blocks are disjoint and partition the code in an executable. We define an *assembly function* to be a collection of basic blocks with a single *entry point*, which is the target of the instruction that transfers control from the external code into the assembly function code, and one or more *exit points*, which are instructions that transfer control to external code not belonging to the function. All code reachable from the entry point before reaching an exit point constitutes the body of the assembly function, except code reachable only through call instructions before corresponding return instructions, which is instead part of the called function. In other words, the body of a function is assumed to continue with the next instruction after a call instruction. An exit point can be a return or interrupt instruction. Our definition does not include assembly functions with multiple entry points, which we treat as multiple (partially overlapping) assembly functions, each including all code reachable from one entry point to any exit point.

If one assembly function jumps to another, this definition considers the blocks following the jump target to be part of the assembly function to extract. We can further extend our definition of an exit point to include jumps to the entry point of any other assembly function in the program's binary or in an external dynamic linked library (DLL). For this we need a list of entry points for other assembly functions, which can be given or approximated by considering any target of a call instruction to be an entry point.

Problem definition. The problem of assembly function reuse is defined as: given the binary of a program and the entry point of an assembly function in the binary, identify the interface and extract the instructions and data dependencies that belong to the assembly function so that it is self-contained and can be reused by external C code. The extracted function consists of both an inline assembly function with a C prototype and a header file containing the function's data dependencies. The problem definition when the code fragment is not an assembly function is the same, except that it requires the exit points to be given.

2.3 Scope

Reusing an arbitrary assembly function is extremely challenging because the function interface can be convoluted and the function can have complex side effects. To limit the scope of the problem we make the following assumptions about the function to be extracted:

- The function entry point is known. For transformation functions, we automatically discover them using a previously proposed technique that flags functions with a high ratio of arithmetic and bitwise operations [24, 46].
- Since our approach uses dynamic analysis, we assume that we can execute the function at least once. If some specific input is needed to reach the function, we assume we are provided with such input.
- The function has a fixed parameter list. Thus, we exclude functions with variable-length list of arguments such as `printf`.
- The function is not passed complex recursive structures such as lists or trees (pointers to single-level structures are supported).

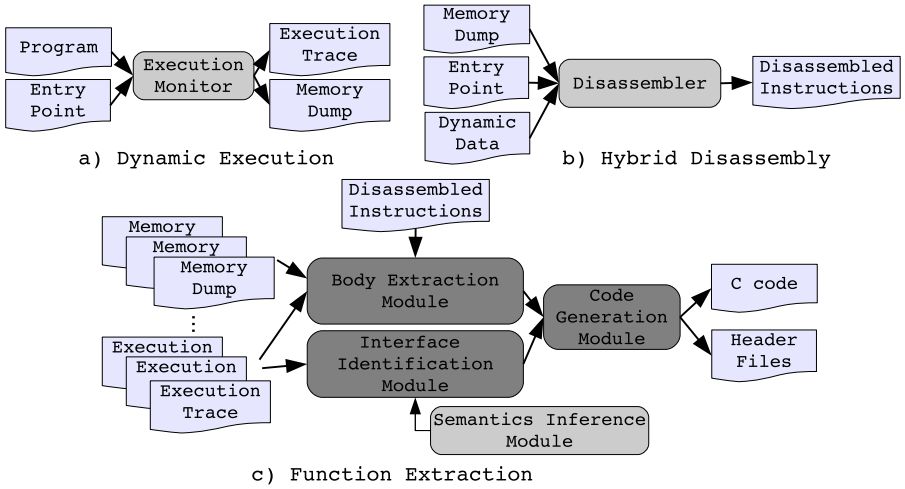

Figure 1. Our assembly function reuse approach. The core of our approach is the function extraction step implemented by BCR. The three dark gray modules in function extraction have been specifically designed for this work. The execution monitor, disassembler, and semantics inference module (light gray) are reused from previous systems.

- The function does not call system calls directly (e.g., through interrupt or `sysenter` instructions) but instead uses system calls only through well-known functions that are available in the target system where the function is reused (e.g., the standard C library, or the Windows API if the target system is Windows-based).
- The function contains no code that explicitly uses its own location. For example, the code should not check if it is loaded at a specific address or offset. This restriction excludes most self-modifying code. However, the function may still reference global addresses through standard position-independent-code and dynamic linking: relocatable and non-relocatable code are both supported.

An important class of functions that usually satisfy these constraints are *transformation functions*. Transformation functions include encryption and decryption, compression and decompression, code packing and unpacking, checksums, and generally any function that encodes given data in a different format. Such functions are usually well-contained, have clean interfaces, limited side effects, and are interesting for many security applications.

Handling obfuscation. Our approach can be applied to both benign code and malware. When applying it to malware we need to consider the obfuscation techniques that malware often uses. Common obfuscation techniques used to hamper static analysis such as binary packing, adding unnecessary instructions, or replacing calls with indirect jumps do not affect our hybrid disassembly approach because it uses dynamic analysis to complement static disassembly. However, a premise of our approach is that we can observe a sample's execution in our analysis environment (based on a system emulator). Thus, like other dynamic approaches, our approach can be evaded using techniques that detect virtualized or emulated environments [27]. In addition, an adversary may make its code hard to reuse, for example, by mixing unrelated functionality, adding unnecessary parameters, or inlining functions. We have not seen such obfuscations in our examples. We consider them another instance of code that is hard to reuse and may require automated or manual analysis on top of our techniques.

2.4 Approach and System Architecture

Our assembly function reuse approach comprises three steps: *dynamic analysis*, *hybrid disassembly*, and *function extraction*. Figure 1 shows the three steps. In the dynamic analysis step the program is run inside the *execution monitor*, which is an emulator based on QEMU [13, 44] that can produce execution traces containing the executed instructions, the contents of the instructions' operands and optional taint information. The execution monitor tracks when execution reaches a given entry point and when it leaves the assembly function via an exit point. When an exit point is reached, the execution monitor produces a memory dump, i.e., a snapshot of the process memory address space. This step may be repeated to produce multiple execution traces and memory dumps.

In the hybrid disassembly step, BCR recovers the instructions comprising the function's body using a combination of static and dynamic analysis. It first tries to statically disassemble as much as possible from the memory dump starting at the function's entry point, using the IDA Pro commercial disassembler [4]. Then, it uses the information from the execution traces generated by the dynamic analysis to resolve indirect jumps and calls, and invokes the static disassembler to disassemble instructions at those locations. If the binary is not packed, static disassembly can be performed directly on the program binary, otherwise the memory dump produced during the dynamic analysis step is used. The hybrid disassembly step outputs the disassembled instructions belonging to the function body.

The core of our approach is the function extraction step. It is implemented by BCR and consists of three sub-steps. The *interface identification module* identifies the function's parameters and outputs (i.e., the function prototype). The *body extraction module* arranges the disassembled instructions into basic blocks, and rewrites addresses in jumps and table accesses to use labels. Finally, the *code generation module* takes as input the function prototype and the control flow graph of the assembly function, and produces as output the C files with the function and header files with its data dependencies. The interface identification module, the body extraction module, and the code generation module have been specifically designed and implemented in this work. The execution monitor [13,44], disassembler [4], and semantics inference module [24] are pre-existing tools. We detail the interface identification module in Section 3, and the body extraction module and code generation module in Section 4.

2.5 Running Example

Figure 2 shows our running example. At the top is the source code for the `encode` function, which reads `len` characters from buffer `src`, transforms them using the static table `enc_tbl`, and writes them to the `dst` buffer. Below it is the assembly function corresponding to the `encode` function, extracted by BCR from the program binary. The large boxes in the figure show the C prototype produced by the interface identification module, and the prologue and epilogue introduced by the code generation module. The smaller boxes show the additional elements in the body of the function that have been rewritten or modified to make the function stand-alone. The rest are the unmodified assembly instruction extracted by the body extraction module. Also produced, but omitted from the figure, is a header file defining a table called `tbl_004000000`, containing a memory dump of the original module.

```
char enc_tbl[256] = { 0x53, … ,0x9c };
int encode(char*src, char*dst, int len) {
  int i;
  if (!src || !dst) return -1;
  memset(dst, 0, len);
  for (i=0; i<len; ++i)
    dst[i] = enc_tbl[src[i]];
  return len;
}
```

```
static int func_00401000(
    void* buf0,  /* IN; STACK(0); FixLen(4); PTR */
    void* buf1,  /* IN-OUT; STACK(1); FixLen(4); PTR */
    data32_t buf0_len,  /*IN;STACK(2); FixLen(4); LEN */
) {
data32_t      retval_EAX;
  __asm__ __volatile__ (
"push   %[val0]\n\t"
"push   %[buf1]\n\t"
"push   %[buf0]\n\t"
"call   lbl00401000\n\t"
"jmp    lbl_func_00401000_ret\n\t"
"lbl00401000:\n\t"
"push   %%ebp\n\t"
"mov    %%esp,%%ebp\n\t"
"push   %%ecx\n\t"
"cmpl   $0x0,0x8(%%ebp)\n\t"
"je   lbl00401010\n\t"
 ...
"lbl00401015:\n\t"
"mov    0x10(%%ebp),%%eax\n\t"
"push   %%eax\n\t"
"push   $0x0\n\t"
"mov    0xc(%%ebp),%%ecx\n\t"
"push   %%ecx\n\t"
"call   memset\n\t"
"add    $0xc,%%esp\n\t"
"movl   $0x0,-0x4(%%ebp)\n\t"
"jmp    lbl00401039\n\t"
 ...
"lbl00401041:\n\t"
"mov    0x8(%%ebp),%%ecx\n\t"
"add    -0x4(%%ebp),%%ecx\n\t"
"movsbl (%%ecx),%%edx\n\t"
"mov    0xc(%%ebp),%%eax\n\t"
"add    -0x4(%%ebp),%%eax\n\t"
"mov    0x3018+tbl_00400000(%%edx),%%cl\n\t"
"mov    %%cl,(%%eax)\n\t"
"jmp    lbl00401030\n\t"
"lbl0040105a:\n\t"
"mov    0x10(%%ebp),%%eax\n\t"
"mov    %%ebp,%%esp\n\t"
"pop    %%ebp\n\t"
"ret    \n\t"
"lbl_func_00401000_ret:\n\t"
  : /* outputs */ "=a" (retval_EAX)
  : /* inputs */ [buf0] "mem" (buf0), [buf1] "mem" (buf1),
    [val0] "mem" (val0), [buf2] "c" (buf2)
  : /* clobber list */ "memory"
  );
  return retval_EAX;
}
```

Figure 2. Running example. At the top is the source code for the `encode` function and below the extracted version of the assembly function.

3 Function Prototype Identification

The goal of function prototype identification is to build a C function prototype for the assembly function so that it can be reused from other C code. The C prototype comprises the function's name and a list of its *formal parameters*. However, formal parameters do not directly appear at the binary code level, so BCR works with a binary-level abstraction, which we term an *assembly parameter* and describe next. At the same time, we collect some additional information, such as the parameter length or its semantics. This information does not directly appear in the prototype, but it is needed for interfacing with the extracted code.

The interface identification module identifies the assembly parameters using a dynamic analysis that takes as input the execution traces produced by the execution monitor. Thus, it can only extract parameters that have been used by the function in the executions captured in the given execution traces. To increase the coverage inside the function white-box exploration techniques that work directly on binaries can be used [23, 31]. In practice, we have seen that few traces are needed to capture the full prototype of the function and have not needed such exploration techniques.

In the remainder of this section we describe how to identify the prototype of an assembly function. The process for identifying the prototype of an arbitrary binary code fragment is analogous.

Parameter abstraction. An assembly parameter plays a role for an assembly function analogous to a formal parameter for a C function, specifying a location representing an input or output value. But instead of being referred to by a human-written name, assembly parameters are identified with a location in the machine state. To be specific, we define assembly parameters with five attributes:

1. The *parameter type* captures whether it is only an input to the function (`IN`), only an output from the function (`OUT`) or both (`IN-OUT`). An example of an `IN-OUT` parameter is a character buffer that the assembly function converts in-place to uppercase.
2. The *parameter location* describes how the code finds the parameter in the program's state. A parameter can be found on the stack, in a register, or at another location in memory. For stack parameters, the location records the fixed offset from the value of the stack pointer at the entry point; for a register, it specifies which register. Memory locations can be accessed using a fixed address or pointed by another pointer parameter, perhaps with an additional offset. BCR also specially classifies globals that are accessed as tables via indexing from a fixed starting address, recording the starting address and the offset.
3. The *parameter length* can be either fixed or variable. A variable length could be determined by the value of another length parameter, or the presence of a known delimiter (like a null character for a C-style string).
4. The *parameter semantics* indicates how its value is used. Parameters have pointer or length semantics if they are used to identify the location and size of other parameters, as previously described. Our parameter abstraction supports a number of semantic types related to system operations, such as IP addresses, timestamps, and filenames. An "unknown" type represents a parameter whose semantics have not been determined.
5. The *parameter value list* gives the values BCR has observed the parameter to take over all assembly function executions. This is especially useful if the parameter's semantics are otherwise unknown: a user can just supply a value that has been used frequently in the past.

Overview. The interface identification module performs three steps. For each assembly function execution, it identifies a list of assembly parameters used by the assembly function in that run (Section 3.1). Next, it combines the assembly parameters from multiple runs to identify missing parameters and generalizes the parameter attributes (Section 3.2). Finally, it identifies additional semantics by running the assembly function again in the execution monitor using the parameter information and a taint tracking analysis (Section 3.3). Later, in Section 4.2, we will explain how the code generation module translates the assembly parameters produced by the interface identification module into the formal parameters and outputs the C function prototype.

3.1 Identifying the Assembly Parameters from a Function Run

For each function run in the execution traces the interface identification module identifies the run's assembly parameters. Because there are no variables at the binary level (only registers and memory), this module introduces abstract variables (sometimes called A-locs [22]) as an abstraction over the machine-level view to represent concepts such as buffers and stack parameters. These variables must be sufficiently general to allow for rewriting: for instance, the addresses of global variables must be identified if the variable is to be relocated. A final challenge is that because the code being extracted might have been created by any compiler or written by hand, BCR must make as few assumptions as possible about its calling conventions.

In outline, our approach is that the interface identification module first identifies all the bytes in the program's state (in registers or memory) that are either an input or an output of the assembly function, which we call *input locations* and *output locations*, respectively. It then generalizes over those locations to recognize abstract locations and assembly parameters. To get the best combination of precision and efficiency, we use a combination of local detection of instruction idioms, and whole-program dataflow analysis using tainting and symbolic execution. In the remainder of this section we refer to an assembly parameter simply as "parameter" for brevity, and use the term "formal parameter" to refer to the parameters in the C function prototype. Next, we define a program location and what input and output locations are.

Program locations. We define a *program location* to be a one-byte-long storage unit in the program's state. We consider four types of locations: *memory locations*, *register locations*, *immediate locations*, and *constant locations*. Each memory byte is a memory location indexed by its address. Each byte in a register is a register location; for example, the 32-bit register EAX has four locations EAX(0) through EAX(3), two of which are also the registers AL and AH. An immediate location corresponds to a byte from an immediate in the code section of some module, indexed by the offset of the byte with respect to the beginning of the module. Constant locations play a similar role to immediate locations, but are the results of instructions whose outputs are always the same. For example, one common idiom is to XOR a register with itself (e.g., *xor %eax, %eax*), which sets the register to zero.

Input locations. We define an input location to be a register or memory location that is read by the function in the given run before it is written. Identifying the input locations from an execution trace is a dynamic dataflow-based counterpart to static live-variables dataflow analysis [40], where input locations correspond to variables live at function entry. Like the static analysis, the dynamic analysis conceptually proceeds backwards, marking locations as inputs if they are read, but marking the previous value of a location as dead if it is overwritten. (Since we are interested only in liveness at function entrance, we can use a forward implementation.) The dynamic analysis is also simpler because only one execution path is considered, and the addresses in the trace can be used directly instead of conservative alias analysis. This basic determination of input locations is independent of the semantics of the location, but as we will explain later not all input locations will be treated as parameters (for instance, a function's return address will be excluded).

Output locations. We define an output location to be a register, memory, or constant location that is written by the extracted function and read by the code that executes after the function returns. Extending the analogy with compiler-style static analysis, this corresponds to the intersection of the reaching definitions of the function's code with the locations that are live in the subsequent code. Like static reaching definitions [40], it is computed in a single forward pass through the trace.

Our choice of requiring that values be read later is motivated by minimizing false positives (a false positive output location translates into an extra parameter in the C function prototype). This requirement can produce false negatives on a single run, if an output value is not used under some circumstances. However, our experience is that such false negatives can be well addressed by combining multiple function runs, so using a strict definition in this phase gives the best overall precision.

Approach. The input and output locations contain all locations belonging to the assembly parameters and globals used by the assembly function, without regard to calling conventions. In addition to identifying them, the interface identification module needs to classify the input and output locations into higher level abstractions representing parameters. Also, it needs to identify whether a parameter corresponds to a stack location, to a global, or is accessed using a pointer. The overall parameter identification process from one function run is summarized in Table 1 and described next.

For efficiency, the basic identification of parameters is a single forward pass that performs only local analysis of instructions in the trace. It starts at the entry point of one execution of a function, and uses one mode to analyze both the function and the functions it calls, without discerning between them (for instance, a location is counted as an input even if it is only read in a called function), and another mode to analyze the remainder of the trace after the function finishes. For each instruction, it identifies the locations the instruction reads and writes. For each location, it identifies the first and last times the location is read and written within the function, as well as the first time it is read or written after the function. Based on this information, a location is classified as an input location if it is read inside the function before being written inside the function, and as an output location if it is written in the function and then read outside the function before being written outside the function; observe that a location can be both an input and an output.

At the same time, the analysis identifies stack and table accesses by a local matching of machine code idioms. The ESP register is always considered to point to the stack. The EBP register is only considered to point to the stack if the difference between its value and that of ESP at function entrance is a small constant, to sup-

Step	Description
1	Identify stack and table accesses
2	Identify input and output locations
3	Remove unnecessary locations (e.g., saved registers, ESP, return address)
4	Identify input and input-output pointers by value
5	Split input locations into parameter instances using pointer, stack and table access information
6	Identify input parameter pointers by dataflow
7	Split output locations into parameter instances using pointer information
8	Identify output parameter pointers by dataflow

Table 1. Summary of parameter identification process for a function run.

port both code that uses it as a frame pointer and code that uses it as a general-purpose integer register. Then, a memory access is a stack access if it uses a stack register as a starting address and has a constant offset. On the other hand, a memory access is classified as a table access if its starting address is a constant and the offset is a non-stack register. The starting address and offset values in stack and table accesses are recorded for future use.

Excluding unnecessary input locations. The input locations given by the simple liveness-style definition above include several kinds of locations with bookkeeping roles in function calls which should not be considered parameters, so we next discuss how to exclude them. To exclude the return address, the interface identification module ignores any memory locations written by a call instruction or read by a return instruction during the function execution. To exclude the stack pointer, it ignores any access to ESP. When code calls functions in a dynamically linked library, it fetches the real entry point of the function from an export table, but we exclude such loads.

Most complex is the treatment of saved registers. For instance, we define a stack location to be used for saving the register EBX if the contents of EBX are first saved in that location with a push instruction, and later restored to EBX with a pop instruction. But the location is not a saved register location if the value is popped to a different register than it was pushed from, if the stack value is accessed before the pop, or if after the pop, the stack value is read before being overwritten. Conventionally, the stack is used to save certain registers designated by the calling convention if a called function modifies them, but our analysis is independent of the calling convention's designation: it simply excludes any location used only for saving a register.

Identifying pointers. A final building block in identifying parameters is to identify locations that hold pointers. The interface identification module uses a combination of two approaches for this task: an inexpensive value-based method that can be applied on all locations, and a more expensive dataflow-based method that works by creating a symbolic formula and is applied selectively. To detect a pointer by value, BCR simply checks each sequence of four consecutive input locations (pointers are four bytes on our 32-bit architecture) to see if their value forms an address of another input or output location. However, this simple approach can fail to detect some pointers (for instance, the address of a buffer that was only accessed with non-zero indexes), so we also implement a more sophisticated approach.

To identify more pointers, the interface identification module uses a symbolic execution approach using our Vine system [18] to analyze an indirect memory access. The input locations to the function are marked as symbolic variables, and the module computes a formula for the value of the effective address of the access in terms of them, using dynamic slicing [21]. It then performs algebraic simplifications and constant folding on the formula, and checks whether it has the form of a 32-bit input plus a constant. If so, the input locations are considered a pointer, and the constant an offset within the region the pointer points to. (The reverse situation of a constant starting address and a variable offset does not occur, because it would already have been classified as a global table.) Though precise, this symbolic execution is relatively expensive, so the interface identification module uses it only when needed, as we will describe next.

Identifying assembly parameters from input and output locations. Once the input and output locations have been identified and unnecessary locations removed, the interface identification module identifies input and input-output pointers by value as explained above. Then it uses the pointers, stack, and table accesses to classify the input and output locations into assembly parameters. Each parameter is a contiguous region in memory (or a register), but two distinct parameters may be adjacent in memory, so the key task is separating a contiguous region into parameters. The module considers a location to be the start of a new parameter if it is the start of a pointer, the address after the end of a pointer, or the location of a pointer, stack, or table access. With the information found so far, the interface identification module determines the parameter type, location, and value, and if the parameter has pointer semantics. The parameter length is provisionally set to the length seen on this run.

Then, the interface identification module attempts to further classify any parameters that are in memory but are not on the stack and are not known globals by applying the dataflow-based pointer identification analysis. Specifically, it checks whether the access to the starting location of the parameter was a pointer access; if so, it updates the type of the pointed-to parameter and the semantics of the pointer parameter accordingly. After classifying the input locations and pointers in this way, the module classifies the output locations similarly, and to identify and classify other pointers that point to them.

3.2 Combining Assembly Parameters from Multiple Function Runs

The set of assembly parameters identified from a single run may be incomplete, for instance if a parameter is used only in a limited way on a particular execution path, like `src` and `dst` in Figure 2. Therefore the interface identification module further improves its results by combining the information about parameters identified on multiple runs.

The final set of parameters identified is the union of the parameters identified over all runs, where parameters are considered the same if they have the same parameter location. When parameters with the same location differ in other attributes between runs, those attributes are merged as follows:

- The parameter type generalizes to input-output if it was input in some runs and output in others.
- The parameter length generalizes to variable-length if it was fixed-length in some runs and variable-length in others, or if it had differing lengths across runs.
- The parameter semantics generalizes to any non-unknown value if it was a known value in some runs and unknown in others (e.g., a parameter is considered a pointer if it was identified to be a pointer at least once, even if it was considered unknown on runs when it was NULL). On the other hand, the semantics are replaced with unknown if they had conflicting non-unknown values on different runs.
- The parameter value list is the union of all the observed values.

3.3 Identifying Parameter Semantics

In addition to the declared type of a parameter included in the C prototype, it is also common (e.g., in MSDN documentation [10]) to provide additional information in text or a comment that explains how the parameter is used; what we refer to as its *semantics*. For instance, one `int` parameter might hold the length of a buffer, while another is an IP address. We next describe the techniques the interface identification module uses to identify such parameter semantics.

Two kinds of semantics that occur frequently in transformation functions as part of specifying other input and output parameters are pointers and lengths. As described above, the parameter identification process finds pointer parameters at the same time it identifies the parameters they point to. To identify length parameters, their targets, as well as variable-length parameters that use a delimiter to mark the end of the parameter (e.g., null-terminated strings), we leverage previously proposed protocol reverse engineering techniques [26, 47] based on taint tracking.

The interface identification module also builds on taint tracking to detect semantics related to system operations such as IP addresses, timestamps, ports, and filenames, using a kind of type inference [24]. Certain well-known functions take inputs or produce outputs of a particular type, so BCR uses taint tracking to propagate these types to the target function (the one being extracted) if an output of a well-known function is used as an input to the target function, or an output of the target function is an input to a well-known function. For instance, the argument to the `inet_ntoa` function is an IP address, so an output parameter that is used to derive that argument must itself be an IP address. Conversely, if an input parameter is based on the return value of `RtlGetLastWin32Error`, it must be an error code. Currently, BCR supports 20 semantics, the 18 semantics defined in [24], plus "pointer" and "unknown". A similar approach can be used at the instruction level to select a more specific C type (such as `float` rather than `int`) [33].

Taint-tracking-based semantics inference takes advantage of the execution monitor's support for function hooks, which are instrumentation code executed just before and/or just after the execution of a chosen function. Hooks added after the execution of well-known functions and the target function taint their outputs, and hooks before their execution check if their inputs are tainted. Because such hooks can only be added to the target function after its parameters have been identified, semantics inference requires an extra run of the function in the execution monitor.

4 Function Body Extraction and C Code Generation

In this section we first present how the body extraction module extracts the instructions that form the body of an assembly function, and then describe how the code generation module produces a C function with an inline-assembly body from the output of the interface identification module and the body extraction module. The

key challenges are disassembling all relevant instructions from the possibly stripped binary and adjusting the extracted code if it uses a calling convention different from what the C code expects. For brevity, we use "C function" to refer to a function with a C prototype and an inline-assembly body.

4.1 Function Body Extraction

Extracting the function body is a recursive process that starts by extracting the body of the given function and then recursively extracts the body of each of the functions that are descendants of this function in the function call graph. The body extraction module classifies descendant functions into two categories: *well-known functions* that may be available in the system where the C function is going to be recompiled, e.g., functions in the standard C library or in the Windows Native API, and the rest, which we term *internal functions*. The body extraction module extracts the body of the given function and all internal descendant functions. As an optimization, it avoids extracting well-known functions. This increases portability: for example if a function from a Windows executable uses `strcpy` from the standard C library, it can be recompiled in a Linux system making a call to the local `strcpy` function. In other cases, portability is not possible because the function may not have a direct replacement in the target OS (e.g., there is no direct replacement in Linux for NtReadFile), so this optimization is not performed. For instance, in our running example, shown in Figure 2, the `encode` function calls `memset`; since it is part of the C library, it is skipped.

Hybrid disassembly. The body extraction module uses *hybrid disassembly* that combines static disassembly from the program binary or a memory dump with dynamic information from execution traces [41].

In detail, the body extraction module supports three modes of operation: purely static, purely dynamic, and hybrid disassembly. In purely static mode, the body extraction module statically disassembles the code starting at the given function entry point, using the IDA Pro [4] commercial disassembler. If the program binary is not packed, then disassembly is performed directly on the executable. For packed binaries disassembly is performed on the memory dump taken by the execution monitor at the exit point of the function. It is important to take the dump at the end of the function's execution to maximize the code pages present in the dump, as pages may not be loaded into memory till they are used. Purely static disassembly provides good code coverage but may not be able to disassemble code reachable through indirect jumps or calls, or memory dumps if the instructions are re-packed after being executed.

In purely dynamic mode, the body extraction module extracts only instructions belonging to the function and its descendants that appear in the given execution traces. This mode has low code coverage but has no trouble dealing with packed executables, or indirect jumps or calls.

In hybrid disassembly mode, the body extraction module combines both static disassembly with dynamic information from the execution traces to obtain the best of both modes. We have found that hybrid disassembly works best and have set it to be the default mode of operation. For hybrid disassembly, the body extraction module first uses static disassembly starting at the given function entry point. In the presence of indirection, the static disassembler may miss instructions because it can not resolve the instructions' targets. Thus, the body extraction module collects the targets of all indirect jumps and calls seen in the execution traces and directs the static disassembler to continue disassembling at those addresses. For example, in Figure 2, the call to the memset function was originally a direct call to a stub that used an indirect jump into `memset`'s entry point in a dynamic library. The body extraction module resolves the target of the jump and uses the information about exported functions provided by the execution monitor to determine that the function is the standard `memset`. In addition, the body extraction module uses a dataflow-based approach to statically identify the targets of jump tables, another class of indirect jumps often used to implement switch statements [28].

There exist some situations where static disassembly may not be possible even from a memory dump, for instance if a program re-packs or deletes instructions right after executing them: the code may be gone by the time a dump is taken. In such a situation hybrid disassembly smoothly falls back to be equivalent to purely dynamic mode. To summarize, hybrid disassembly uses static disassembly when possible and incorporates additional dynamic information when it encounters indirection or packed memory dumps. For each function, hybrid disassembly stores the disassembled basic blocks, and recovers the control flow graph.

Rewriting call/jumps to use labels. Once the C function is recompiled it will almost certainly be placed at a different address, so the body extraction module needs to make the code relocatable. To enable this, it inserts a label at the beginning of each basic block. Then, it rewrites the targets of jump and call instructions to use these labels. If the target of a jump instruction has not been recovered by the hybrid disassembly, it is rewritten to use a unique missing block label that exits the function with a special error condition. Figure 2 uses small boxes to highlight the inserted block labels and the rewritten call/jump instructions. Rewriting the call/jump

instructions to use labels also enables a user or a subsequent tool (like the SFI tool discussed in Section 5.5) to instrument the function or alter its behavior by inserting new instructions in the body.

Rewriting global and table accesses. The extracted C function is composed of a C file with the assembly function and a header file. The header file contains a memory dump of the module containing the function to extract, taken at the function's exit point on a given run. The body extraction module rewrites instructions that access global variables or tables to point to the corresponding offsets in the memory dump array. This way the extracted function can access table offsets that have not been seen in the execution traces. In our running example, the header file is not shown for brevity, but the array with the contents from the memory dump is called `tbl_004000000` and the instruction that accesses `enc_tbl` has been rewritten to use the label `0x3018+tbl_00400000` which is the first byte of `enc_tbl` in the memory dump. The memory dump is taken at the function's exit point, but if the interface identification module discovers any input parameters that are accessed using a fixed address and modified inside the function, e.g., a global table that is updated by the function, it ensures that the parameter values on function entry are copied into the dump, so that they are correct when the function is invoked again.

An alternative approach would be to create separate C arrays and variables for each global parameter, which would reduce the space requirements for the extracted function. Though this would work well for scalar global variables, it would be difficult to infer the correct size for tables, since the binary does not contain bounds for individual variables, and code compiled from C often does not even have bounds checks. (An intermediate approach would be to estimate the size of a table by multiplying the largest observed offset by a safety factor; this would be appropriate if it could be assumed that testing covered at least a uniform fraction of the entries in each table.)

4.2 C Code Generation

The code generation module writes the output C files using the information provided by the interface identification module and the body extraction module. To encode the function body the code generation module uses GCC's inline assembly feature [3]. It wraps the function body in an assembly block and then puts the assembly block inside a function definition with a C function prototype, as shown in Figure 2. In addition it creates a C header file containing the memory dump as an array. Though our current implementation is just for GCC, the inline assembly features of Visual C/C++ [11] would also be sufficient for our purposes. In fact, some of the Visual C/C++ features, such as "naked" inline assembly functions, for which the compiler does not generate a prologue or epilogue, could simplify our processing.

The assembly block contains the assembly instructions in AT&T syntax, and the list of inputs, outputs, and clobbered registers. These are filled using the parameter information provided by the interface identification module. When GCC compiles the function, it will add prologue and epilogue code that affects the stack layout, so even if the extracted function originally used a standard calling convention, it would not find the stack parameters where it expects. To overcome this problem, the code generation module inserts wrapper code at the beginning of the function that reads the parameters from the C prototype (as inputs to the assembly block), puts them in the stack or register locations expected by the extracted function, and calls the extracted entry point. After the call instruction it inserts a jump to the end of the function so that the epilogue inserted by GCC is executed. The second box in Figure 2 shows this wrapper.

The C prototype comprises the function name and the formal parameters of the function. The function name is based on its entry point (`func_00401000` in the running example), and each parameter's C type is based on its size and whether it is a pointer. Input and input-output parameters located in the stack or registers appear first, with stack parameters appearing in order of increasing offset (this means that if the extracted function used the most common C calling convention, their order will match the original source). For each output parameter returned using a register, the code generation module adds an additional pointer formal parameter at the end of the C prototype and uses the outputs list in the assembly block to let GCC know that the register needs to be copied to the pointed-to location. Additionally, for output global or table parameters the code generation module adds a C variable corresponding to the start address of the global or table in the memory dump. This makes the function's side effects available to other C code.

Each formal parameter is also annotated with a comment that gives information about the attribute values for the corresponding assembly parameter such as the parameter type and its semantics. These are useful for a user that wants to reuse the function. In addition, it prints the most common value seen for each parameter during the multiple executions along with the percentage of executions where the parameter showed that value. This allows the user to select a value for the parameter when the parameter semantics are unknown. The function prototype is shown in the first box in Figure 2.

5 Evaluation

This section describes the experiments we have performed to demonstrate that our binary code reuse approach and implementation is effective for security applications such as rewriting encrypted malware network traffic and static unpacking, that non-function fragments can be extracted to give useful functions, and that extracted functions can be used safely even though they come from an untrusted source.

5.1 Rewriting MegaD's C&C Protocol

MegaD is a prevalent spam botnet that accounted for 35.4% of all spam in the Internet in a December 2008 study [8], and still accounts for 9% as of September 2009 [9]. Recent work reverse-engineers MegaD's proprietary, encrypted, C&C protocol [24], and demonstrates rewriting messages on the host by modifying a buffer before encryption. In this section we show that our assembly function reuse approach enables the same C&C rewriting on a network proxy, by extracting the bot's key generation and encryption functions.

Function extraction. MegaD's C&C protocol is protected using a proprietary encryption algorithm, and the bot contains functions for block encryption, block decryption, and a common key generator. We identify the entry points of the three functions using previously proposed techniques that flag functions with a high ratio of arithmetic and bitwise operations [24, 46].

First, we use BCR to automatically extract the key generation function. The identified prototype shows that the function has two parameters and uses two global tables. The first parameter points to an output buffer where the function writes the generated key. The second parameter is a pointer to an 8 byte buffer containing the seed from which the key is generated. Thus, the function generates the encryption key from the given seed and the two tables in the binary. Other attributes show that all calls to the key generation function use the same "abcdefgh" seed, and that the two tables are not modified by the function.

Although the entry points for the block encryption and decryption functions are different, the first instruction in the block decryption function jumps to the entry point of the block encryption function, so here we describe just the encryption function. The prototype extracted by BCR has 3 parameters and uses 6 global tables. The first parameter points to an input buffer containing a key (as produced by the key generation function). The other two parameters are pointers to the same 8 byte input-output buffer that on entry contains the unencrypted data and on exit contains the encrypted data.

The technique to automatically detect transformation functions identifies the functions with highest ratio of arithmetic and bitwise operations, which for block ciphers is usually the functions that process a single block. To encrypt or decrypt an arbitrary message, we would like a function that encrypts or decrypts arbitrary length data. Thus, when using this technique, after BCR extracts the detected transformation functions, we instruct it to extract their parent functions as well. Then, we compare the prototype of each detected function with the one of the parent. If the parent's prototype is similar but accepts variable-length data, e.g., it has a length parameter, then we keep the parent function, otherwise we manually write a wrapper for the block function.

For MegaD, the parent of the block encryption function has additional parameters, because it performs other tasks such as setting up the network and parsing the message. It contains no single loop that performs decryption of a variable-length buffer; instead, decryption is interleaved with parsing. Since we are not interested in the parent function's other functionality, we write our own wrapper for the block encryption function.

To verify that the extracted encryption/decryption function works correctly, we augment a grammar for the unencrypted MegaD C&C protocol, reported in earlier work [24], to use the extracted decryption function. This augmented grammar serves as input to the BinPac parser shipped with the Bro intrusion detection system [42]. Using the augmented grammar, Bro successfully parses all the encrypted MegaD C&C messages found in our network traces.

Network-based C&C rewriting. To perform network rewriting we must deploy the encryption/decryption function, as well as the session keys, in a network proxy. Such a proxy will only be effective if the functions and keys match those in the bots, so to estimate the rate at which they change we repeated our analysis with an older MegaD sample. According to malware analysis online services [14, 19], our primary sample was first seen in the wild in December 2008, and our older one in February 2008. Although there are differences between both samples, such as the older sample using TCP port 80 instead of 443 for its C&C, the parser, using the decryption function and keys extracted from the December sample, is able to successfully parse the C&C messages from the February sample. In addition, we extract the key generation and encryption functions from the February sample and compare them with the ones from the December sample. Although there are syntactic differences, the versions are functionally equivalent, producing the same outputs on more than a billion randomly generated inputs. Thus we conclude that the relevant algorithms and keys, including the session key, have been unchanged during the time span of our samples.

To show how our assembly function reuse approach enables live rewriting on the network, we build a network proxy that is able to decrypt, parse, modify and reencrypt MegaD C&C messages that it sees on the network. To test the proxy we reproduce an experiment from [24], but perform rewriting on the network rather than on the host. The experiment proceeds as follows. We run a live MegaD bot in a virtual environment that filters all outgoing SMTP connections, for containment purposes.

To start, suppose that no proxy is in use. The C&C server sends a command to the bot ordering it to test its ability to send spam by connecting to a test mail server. Because the virtual environment blocks SMTP, the bot sends a reply to the C&C server indicating that it cannot send spam, and afterwards no more spam-related messages are received.

Next, we repeat the experiment adding a network proxy that acts as a man-in-the-middle on traffic between the C&C server and the bot. For each message sent by the bot, the proxy decrypts it and checks if it is a message that it needs to rewrite. When the bot sends the message indicating that it has no SMTP capability, the proxy, instead of relaying it to the C&C server, creates a different message indicating that the SMTP test was successful, encrypts it, and sends it to the C&C server instead. (It would not be sufficient for the proxy to replay a previous encrypted success message, because the message also includes a nonce value selected by the C&C server at the beginning of each dialog.) With the proxy in place, the bot keeps receiving spam-related messages, including a spam template and lists of addresses to spam, though it is unable to actually send spam.

5.2 Rewriting Kraken's C&C Protocol

Kraken is a spam botnet that was discovered on April 2008 and has been thoroughly analyzed [2,5,6,12]. Previous analysis uncovered that Kraken (versions 315 and 316) uses a proprietary cipher to encrypt its C&C protocol and that the encryption keys are randomly generated by each bot and prepended to the encrypted message sent over the network [5, 12]. Researchers have manually reverse-engineered the decryption function used by Kraken and provided code to replicate it [5]. In this paper, we extract Kraken's decryption function using our automatic approach and verify that our extracted function is functionally equivalent to the one manually extracted in previous work. Specifically, when testing the manually and automatically extracted function on millions of random inputs, we find their outputs are always the same. In addition, we extract the corresponding encryption function and a checksum function, used by the bot to verify the integrity of the network messages.

Similarly to the MegaD experiment described in Section 5.1, we build a network proxy that uses the extracted encryption, decryption, and checksum functions, as well as the protocol grammar, and use it to rewrite a C&C message to falsify the result of an SMTP capability check. Unfortunately (for our purposes), none of our Kraken samples connects to a live C&C server on the Internet. Thus, to verify that the message rewriting works we use a previously published Kraken parser [7]. The rewritten message parses correctly and has the STMP flag correctly modified (set to one).

5.3 Reusing Binary Code that is not an Assembly Function

Next, we show that our approach enables reusing a binary code fragment that does not correspond to a complete assembly function, but has a clean interface and performs an independent task. We extract unpacking code from two versions of a trojan horse program *Zbot* used primarily to steal banking and financial information [20]. Zbot uses two nested layers of packing. The samples, provided to us by an external researcher, represent a typical task in the course of malware analysis: they have already had one layer of packing removed, and we have been provided the entry points for a second, more complex, unpacking routine.

The function prototype extracted by BCR is identical for both functions. It contains two pointer parameters: the ESI register points to an input-output buffer containing packed data as input and a count of the number of bytes unpacked as output, while the EDI register points to an output buffer for unpacked data. Since ESI and EDI are not used for parameter passing in any of the standard x86 calling conventions, this suggests these functions were originally written in assembly code.

Although the prototypes are the same, the unpacking functions are not functionally equivalent; they both consist of two distinct loops, and we find that extracting these loops separately captures more natural functional units. Examining the extracted function bodies, we find that both consist of two loops that are separated by `pusha` and `popa` instructions that save and restore processor state. Each loop makes its own pass over the packed data, with the first pass applying a simpler deciphering by subtracting a hardcoded key, and the second pass performing a more complex instruction-by-instruction unpacking. After extracting the two loops into separate functions, we verify that the differences between the versions are only in the first loop: the extracted version of the second loop can be reused across the sample versions. This highlights the fact that as long as a binary code fragment has a clean interface and performs a well-separated task, it can be reused even if it

does not correspond to a complete function in the original machine code.

5.4 Quantitative Summary of Function Extraction

Table 2 summarizes the extraction results for all functions mentioned in Section 5.1 through Section 5.3 and some additional functions that we extract from the OpenSSL library for evaluation purposes. The *General* section of the table shows the number of function runs in the execution traces used as input to the function extraction step, and the total time needed to extract the function. The *Code Extraction* section has the number of instructions in each extracted function, the number of missed blocks and the number of indirect call and jump instructions. The *Parameter Identification* section shows the number of parameters in the C function prototype and the number of false positives (e.g., unnecessary parameters in the prototype) and false negatives (e.g., missing parameters in the prototype). For the OpenSSL functions, the false positives and negatives are measured by comparison with the original C source code. For the malware samples, no source is available, so we compare with our best manual analysis and (for Kraken) with other reported results.

The results show that a small number of executions is enough to extract the complete function without missing blocks or parameters. For samples without indirect jumps or calls, static disassembly recovers all basic blocks. For the samples with indirection, the dynamic information resolves the indirection and enables the static disassembler to find all the instructions in the function body. The Kraken checksum and MegaD encrypt samples are significantly slower to extract than the other samples. This is because they have larger number of invocations of the dataflow-based pointer analysis technique, which dominates the running time. The parameter identification results show that no parameters are missed: some runs do not identify all parameters, but combining multiple executions (Section 3.2) gives complete results. For the functions from OpenSSL, the parameters include fields in a context structure that is passed to the functions via a pointer. There are two false positives in the Kraken functions (i.e., extra parameters are identified), both of which are output parameters reported as returned in the ECX register. These are caused by a compiler optimization (performed by the Microsoft compiler, for instance) that replaces the instruction `sub $4,%esp` to reserve a location on the stack with the more compact instruction `push %ecx`, which has the same effect on the stack pointer and also copies a value from ECX that will later be overwritten. When this idiom occurs in the code following an extracted function that uses ECX internally, the interface identification module incorrectly identifies ECX as a function output. Note that false positive parameters are not a serious problem for usability: extra outputs can simply be ignored, and extra inputs do not change the extracted function's execution.

5.5 Software-based Fault Isolation

If the extracted functions are to be used in a security-sensitive application, there is a danger that a malicious extracted function could try to hijack or interfere with the operation of the application that calls it. To prevent this, we use software-based fault isolation (SFI) [45] as a lightweight mechanism to prevent the extracted code from writing to or calling locations in the rest of the application. SFI creates separate "sandbox" data and code regions for the extracted function, so that it can only write to its data region and it can only jump within its code region. SFI works by adding checks just before each store or jump instruction, but the extracted code still runs in the same address space, so calls from the application are still simple and efficient.

Specifically, we postprocess our extracted malware functions using PittSFIeld, an implementation of SFI for x86 assembly code [39]. PittSFIeld adds new instructions for checks, and to enforce additional alignment constraints to avoid overlapping instructions. Thus, BCR's translation of jumps to use labels is necessary for it to work. PittSFIeld was previously implemented for use with the assembly code generated by GCC, so in order to work with assembly code that could be generated by other compilers or hand-written, we generalize it to save and restore the temporary register used in sandboxed operations, and to not assume that EBP is always a pointer to the stack. We also make corresponding changes to PittSFIeld's separate verification tool, so a user can check the safety of an extracted function without trusting the person who extracted it.

6 Related Work

This section compares our approach with the manual process it aims to replace, techniques for related extraction problems in other domains, and some other tasks that require similar algorithms.

Manual code extraction. Code extraction is a common activity in malware analysis, but it is usually performed manually [5, 6, 30]. While this process can give the analyst a deep understanding of the malicious functionality, it is also very time-consuming. Simple tool support can make some of the repetitive tasks more convenient [1], but existing approaches still require specialized skills.

	General		Code Extraction			Parameter Identification		
Function	**# Runs**	**Run time(sec)**	**# Insn.**	**# Missed blocks**	**# Indirect call/jump**	**# Param.**	**FP**	**FN**
MegaD keygen	4	3	320	0	0	3	0	0
MegaD encrypt	6	257	732	0	0	4	0	0
Kraken encrypt	2	16	66	0	0	7	1	0
Kraken decrypt	1	2	66	0	0	6	0	0
Kraken checksum	1	179	39	0	0	4	1	0
Zbot v1151	2	15	98	0	0	2	0	0
Zbot v1652	2	17	93	0	0	2	0	0
MD5_Init	6	2	10	0	0	1	0	0
MD5_Update	6	38	110	0	1	3	0	0
MD5_Final	7	31	67	0	3	2	0	0
SHA1_Init	1	8	11	0	0	1	0	0
SHA1_Update	1	36	110	0	1	3	0	0
SHA1_Final	2	36	76	0	3	2	0	0

Table 2. Evaluation results. At the top are the functions extracted during the end-to-end applications and at the bottom some additional functions extracted from the OpenSSL library.

Our approach allows this task to be automated, when all that is needed is to be able to execute the functionality in another context.

Input-output relationship extraction. A variant on the extraction problem is extracting the relationship between some given inputs and outputs of a computation. To extract such relationships, previous work has used symbolic execution [25,36] or dynamic binary slicing [36, 37]. When the functionality to be extracted is sufficiently simple, it can be represented by a single input-output symbolic formula. For instance, such input-output formulas can be used for protocol dialog replay [25], or as a malware signature [36]. However, a single formula is not a practical representation for more complex functionality that includes loops or other variant control-flow paths, or uses complex data structures.

Another alternative representation is a dynamic binary slice that captures the instructions needed to produce the output from the inputs in a given execution. Dynamic binary slices are usually generated by applying modified versions of dynamic program slicing techniques [21] on execution traces. For instance, Lanzi et al. [37] produce dynamic binary slices using a combination of backwards and forward slicing, and use them to analyze kernel malware. When it cannot extract an exact input-output symbolic formula, the malware modeling tool of Kolbitsch et al. [36] combines dynamic binary slicing with tainted scopes to capture control dependencies. There are two main differences between extracting input-output symbolic formulas or dynamic binary slices and binary code reuse. First, our problem is more difficult because the inputs and outputs must be inferred. Second, by using a combination of dynamic and static analysis to extract the body of the code fragment we achieve better coverage than purely dynamic techniques.

Other applications of interface extraction. Jiang and Su [34] investigate the problem of automatic interface extraction in C source code, to allow automated random testing for fragments with equivalent behavior. Their task of determining which variables constitute inputs and outputs of a fragment is related to the one we tackle in Section 3, but made easier by the availability of type information. Extracting the code itself is also easier because in their scenario code fragments are restricted by definition to contiguous statements.

Independently, Lin et al. [38] extract an interface to functionality in a benign program in order to add malicious functionality: for instance, to turn an email client into a spam-sending trojan horse. Because the functionality runs in its original context, their interface need not cover all the inputs and outputs of the code, only those relevant to a particular use. Using techniques similar to our output inference, they perform a side-effect analysis to determine whether a function's memory effects can be reverted to hide it from the rest of an execution.

Liveness analysis. The analyses that our tool performs to identify input and output variables are the dynamic analogues of static data-flow analyses performed by compilers, such as live variable and reaching definitions analysis [40]. Some of the same challenges we face have also been addressed in purely static tools such as link-time optimizers that, like our tool, must operate on binary code. For instance, link-time optimizers [32, 43] must also exclude saves of callee-saved registers from the results of naive liveness analysis.

Binary rewriting. Many of the techniques required for binary code reuse are used in binary rewriting and instrumentation applications. For instance, purely static disassembly provides insufficient coverage for even benign applications on Windows/x86 platforms, so state-of-the art rewriting tools require a hybrid of static and dynamic disassembly [41] much as we do. Cifuentes and Van Emmerik [28] introduced the technique we adopt for locating the jump table statements used to implement switch statements.

7 Conclusion

This paper performs the first systematic study of automatic binary code reuse, which we define as the process of automatically identifying the interface and extracting the instructions and data dependencies of a code fragment from an executable program, so that it is self-contained and can be reused by external code.

We have proposed a novel technique to identify the prototype of an undocumented code fragment directly from the program's binary, without access to its source code. We have designed an approach to automatically extract a code fragment from a program binary so that it is self-contained. The extracted code fragment can be run independently of the rest of the program's functionality in an external C program, and can be easily tested, instrumented, or shared with other users.

We have implemented BCR, a tool that uses our approach to automatically extract an assembly function from a program binary. We have used BCR to reuse the cryptographic routines used by two spam botnets in a network proxy that can rewrite the malware's C&C encrypted traffic. In addition, we have extracted an unpacking function from a trojan horse program, and have shown that a code fragment belonging to that function can be reused by the unpacking function for a different sample from the same family. Finally, we have applied software-based fault isolation techniques [39] to the extracted functions to ensure they can be used safely even though they come from an untrusted source.

8 Acknowledgements

We would like to specially thank Fabrice Desclaux for providing us with the unpacking samples and for all his help with this project. We also thank Cedric Blancher and Philippe Biondi for their help. Finally, we are grateful to Rolf Rolles for his valuable comments to improve this manuscript.

This work was performed while Juan Caballero was a visiting student researcher at University of California, Berkeley. This material is based upon work partially supported by the National Science Foundation under Grants No. 0311808, No. 0448452, No. 0627511, and CCF-0424422, by the Air Force Office of Scientific Research under Grant No. 22178970-4170, and by the Army Research Office under grant DAAD19-02-1-0389. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Air Force Office of Scientific Research, or the Army Research Office.

References

[1] Code snippet creator. http://sharemation.com/servil/idaplugs/csc-bin.zip.

[2] Dissecting the Kraken: An analysis of the kraken bonet's obfuscation techniques. http://www.securescience.net/blog/kraken_paper.pdf.

[3] GCC inline assembly HOWTO. http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html.

[4] The IDA Pro disassembler and debugger. http://www.hex-rays.com/idapro/.

[5] Kraken encryption algorithm. http://mnin.blogspot.com/2008/04/kraken-encryption-algorithm.html.

[6] Kraken is finally cracked. http://blog.threatexpert.com/2008/04/kraken-is-finally-cracked.html.

[7] Kraken wireshark dissector. http://www.mnin.org/data/kraken/cli-dissector.rar.

[8] Marshal8e6 security threats: Email and Web threats. http://www.marshal.com/newsimages/trace/Marshal8e6_TRACE_Report_Jan2009.pdf.

[9] Marshal8e6 spam statistics for week ending september 13, 2009. http://www.m86security.com/TRACE/spam_statistics.asp.

[10] Microsoft developer network. http://msdn.microsoft.com.

[11] MSDN: Inline assembler. http://msdn.microsoft.com/en-us/library/4ks26t93.aspx.

[12] Owning Kraken zombies, a detailed dissection. http://dvlabs.tippingpoint.com/blog/2008/04/28/owning-kraken-zombies.

[13] TEMU: The Bitblaze dynamic analysis component. http://bitblaze.cs.berkeley.edu/temu.html.

[14] ThreatExpert. http://threatexpert.com/reports.aspx.

[15] Titanengine. http://www.reversinglabs.com/products/TitanEngine.php.

[16] The undocumented functions: Microsoft windows nt/2k/xp/2003. http://undocumented.ntinternals.net/.

[17] The unpacker archive. http://www.woodmann.com/crackz/Tools/Unpckarc.zip.

[18] Vine: The Bitblaze static analysis component. http://bitblaze.cs.berkeley.edu/vine.html.

[19] Virustotal. http://www.virustotal.com/.

[20] Zbot. http://www.f-secure.com/v-descs/trojan-spy_w32_zbot_hs.shtml.

[21] H. Agrawal and J. R. Horgan. Dynamic program slicing. *ACM SIGPLAN Notices*, 25(6), June 1990.

[22] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *International Conference on Compiler Construction*, Barcelona, Spain, March 2004.

[23] J. Caballero, S. McCamant, A. Barth, and D. Song. Extracting models of security-sensitive operations using string-enhanced white-box exploration on binaries. Technical Report UCB/EECS-2009-36, EECS Department, University of California, Berkeley, March 2009.

[24] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *ACM Conference on Computer and Communications Security*, Chicago, IL, November 2009.

[25] J. Caballero and D. Song. Rosetta: Extracting protocol semantics using binary analysis with applications to protocol replay and NAT rewriting. Technical Report CMU-CyLab-07-014, Cylab, Carnegie Mellon University, October 2007.

[26] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *ACM Conference on Computer and Communications Security*, Alexandria, VA, October 2007.

[27] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *International Conference on Dependable Systems and Networks*, Anchorage, AK, June 2008.

[28] C. Cifuentes and M. V. Emmerik. Recovery of jump table case statements from binary code. In *International Workshop on Program Comprehension*, Pittsburgh, PA, May 1999.

[29] W. Cui, V. Paxson, N. C. Weaver, and R. H. Katz. Protocol-independent adaptive replay of application dialog. In *Network and Distributed System Security Symposium*, San Diego, CA, February 2006.

[30] F. Desclaux and K. Kortchinsky. Vanilla Skype part 1. In *ReCon*, 2006.

[31] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium*, San Diego, CA, February 2008.

[32] D. W. Goodwin. Interprocedural dataflow analysis in an executable optimizer. In *SIGPLAN Conference on Programming Language Design and Implementation*, Las Vegas, ND, June 1997.

[33] I. Guilfanov. A simple type system for program reengineering. In *Working Conference on Reverse Engineering*, Stuttgart, Germany, October 2001.

[34] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *International Symposium on Software Testing and Analysis*, Chicago, IL, July 2009.

[35] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

[36] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *USENIX Security Symposium*, Montréal, Canada, August 2009.

[37] A. Lanzi, M. Sharif, and W. Lee. K-Tracer: A system for extracting kernel malware behavior. In *Network and Distributed System Security Symposium*, San Diego, CA, February 2009.

[38] Z. Lin, X. Zhang, and D. Xu. Reuse-oriented camouflaging attack: Vulnerability detection and attack construction. Technical Report CERIAS-TR-2009-29, Purdue University, November 2009.

[39] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *USENIX Security Symposium*, Vancouver, Canada, July 2006.

[40] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Academic Press, 1997.

[41] S. Nanda, W. Li, L.-C. Lam, and T. Chiueh. BIRD: Binary interpretation using runtime disassembly. In *International Symposium on Code Generation and Optimization*, NY, March 2006.

[42] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: A yacc for writing application protocol parsers. In *Internet Measurement Conference*, Rio de Janeiro, Brazil, October 2006.

[43] B. Schwartz, S. Debray, G. Andrews, and M. Legendre. PLTO: A link-time optimizer for the Intel IA-32 architecture. In *Workshop on Binary Translation*, Barcelona, Spain, September 2001.

[44] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*, Hyderabad, India, December 2008. Keynote invited paper.

[45] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Symposium on Operating Systems Principles*, Asheville, NC, October 1993.

[46] Z. Wang, X. Jiang, W. Cui, and X. Wang. ReFormat: Automatic reverse engineering of encrypted messages. In *European Symposium on Research in Computer Security*, Saint-Malo, France, September 2009.

[47] G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *Network and Distributed System Security Symposium*, San Diego, CA, February 2008.