Structured Binary Editing with a CFG Transformation Algebra

Andrew R. Bernat and Barton P. Miller
Computer Sciences Department, University of Wisconsin
Madison, WI, USA
Email: {bernat,bart}@cs.wisc.edu

***Abstract*—Binary modification allows users to alter existing code or inject new code into programs without requiring source code, symbols, or debugging information. It is critically important that such modification not accidentally create a *structurally invalid* binary that has illegal control flow or executes invalid instructions. Unfortunately, current modification tools do not make this guarantee, instead relying on the user to manually ensure the modified binary is valid. In addition, they fail to provide high-level abstractions of the binary (e.g., functions), instead requiring the user to have a deep understanding of the idiosyncrasies of the instruction set and the behavior of the program. We present *structured binary editing*, which allows users to modify a program binary by modifying its control flow graph (CFG). We define an algebra of CFG transformations that is closed under a *CFG validity* constraint, thus ensuring that users can arbitrarily compose these transformations while preserving structural validity. We have implemented structured binary editing in the Dyninst binary analysis and instrumentation framework, creating a high-level binary modification toolkit. We demonstrate the usefulness of our approach by creating a hot patching tool that closes three vulnerabilities in a running, unmodified Apache HTTPD server without interrupting the server's execution.**

***Index Terms*—binary modification; binary editing; graph transformations; binary validity**

I. Introduction

Binaries are commonly held to be an execute-only program form: rigid, lacking in clear structure, complex to extend and difficult to modify. Today, this is not the case. With the proper techniques, derived from a formal understanding of the characteristics of program binaries, a binary can by extended and modified up to and during execution of the program. In turn, binary modification presents several benefits when compared to source-level or compile-time modification. Binary modification does not require source code, debugging information, or any other information that is not directly required to execute the program. For understanding the basic functionality of the code, such as for program testing [1] or dynamic patching [2], the program or its libraries may only be available as binaries. For understanding the performance characteristics of the code, such as for optimization [3] or performance analysis [4], it is useful to include the effects of compilation and linking on the program. Finally, for understanding the security characteristics of the code, such as for attack detection [5], behavior monitoring [6], cyberforensics [7], [8], and controlling execution [9], the binary is frequently the only program form available.

Current binary modification approaches require users to modify the program by replacing individual instructions [3], [10]. Such approaches have three significant flaws. First, they are not *safe*. They do not protect the user from accidentally creating a *structurally invalid* binary that has illegal control flow or executes invalid instructions; instead, the user must manually ensure each change they make is valid. Second, they do not provide an *abstract*, high-level model of the binary. Instead, they represent the binary only as instructions, which may hide aspects of program behavior that are obvious at the basic block or function level. Third, they are not *platform independent*. Instead, they require the user to thoroughly understand both the idiosyncrasies of the instruction set and the behavior of the program they are modifying.

In this paper, we introduce *structured binary editing*, in which users modify a program binary by manipulating its control flow graph (CFG). The core of our approach is an algebra of CFG transformations that preserve *validity* of the CFG and thus ensure the resulting CFG can be instantiated into a new binary that has corresponding control flow. This algebra is closed under validity, and thus these transformations can be composed to provide powerful binary modification capabilities without creating a structurally invalid program binary; furthermore, no instruction-level understanding is necessary.

This paper makes the following contributions:

A definition of CFG validity: A CFG is a digraph whose nodes represent basic blocks and edges represent the control flow relationship between those blocks. A graph is a *valid* CFG if it represents a structurally valid program. A CFG that has been derived from a structurally valid program to be implicitly valid; however, unconstrained transformation of such a graph does not necessarily preserve validity. We define CFG validity as a constraint over each element in the CFG definition, such that any graph that satisfies this constraint represents a structurally valid program binary.

An algebra of valid CFG transformations: A graph transformation replaces a particular subgraph of an input graph G with a replacement graph [11], [12]. We define a CFG transformation algebra that is closed under our validity constraint, and provide examples of transformations from this algebra (e.g., redirecting an edge or cloning a function). These transformations are purposefully simple and localized to avoid unexpected side-effects on program behavior. We demonstrate that this algebra is closed, and thus our transformations can be

composed to perform complex modifications of the program.

One particularly challenging operation consists of inserting new code into the program. To ensure we preserve validity, we assume that all modification of the program is performed by transforming the CFG. Inserting arbitrary code may allow users to break this assumption. Instead, we define *insertion* transformations that insert control flow operations, such as conditional branches or calls. In addition, we allow users to insert *code snippets*, which are single-entry, single-exit regions of code. Once inserted, snippets can be transformed in the same way as the original CFG. This provides users with the capability to construct more complex code sequences (e.g., code that makes calls or ends in a conditional branch) by combining snippets with CFG transformations.

An indirect control flow validity analysis: Program binaries frequently use *indirect* control flow to targets that are calculated at runtime. Examples of indirect control flow include jump tables, virtual method tables, or function pointers. These constructs present a challenge, since we cannot determine if an indirect control flow instruction is valid by only examining the CFG. To address this problem, we define a dataflow analysis that determines if the destinations of an indirect control flow instruction will be affected by a CFG transformation. If the destinations will be changed, we attempt to determine the new destinations and update the CFG. If our analysis fails, we provide a spectrum of responses, from inserting runtime monitoring code to disallowing the transformation.

We have implemented our structured binary editing approach in the PatchAPI component [13] of the Dyninst binary analysis and instrumentation framework [14], leveraging Dyninst's binary analysis capabilities to construct our initial CFG. Dyninst previously only supported binary instrumentation and whole-function replacement, and thus our binary modification work is a significant extension of Dyninst's capabilities. Finally, we demonstrate the usefulness of our approach by implementing a hot patching tool that we use to apply security bug fixes to a running Apache web server without interrupting its execution.

II. Related Work

Our approach of editing a binary by transforming its CFG is derived from previous work in both binary and compile-time code modification. In this section, we summarize these approaches and relate them to our work. We also describe other binary modification approaches that do not use the CFG, as well as binary instrumentation approaches that allow users to inject new code into the program.

Our work extends concepts introduced by the LANCET [15] tool of the Diablo binary modification toolkit [16]. LANCET allows users to modify a binary by freely modifying its CFG, such as by adding or removing edges or inserting new basic blocks. However, LANCET does not constrain these modifications and thus does not enforce any concept of CFG or binary validity, instead relying on the user to ensure that all modifications are safe. We extend the ideas introduced by LANCET, and address the validity issue with our concept of CFG validity. Finally, the Diablo toolkit requires input binaries to be specially prepared with a customized compiler toolchain. In contrast, our approach works on arbitrary binaries.

Structured binary editing is similar in may ways to compile-time program modification. Examples of compile-time program modification include LLVM [17] and SUIF [18]. These tools allow users to modify the intermediate representation (IR) used by the compiler. Since the IR is more expressive than the CFG, these approaches often provide more modification power while avoiding the validity concerns we address in this work. Unfortunately, these approaches cannot be applied to binaries without deriving an IR from the binary code. Unless the binary includes substantial additional semantic information, such derivation is difficult due to the complexities involved in pointer analysis.

Binary instrumentation frameworks focus on inserting new code into a binary program. This is typically done by allowing a user to annotate a program representation with new code. Such representations include a sequence of instructions (*software dynamic translation* or SDT instrumenters) or a CFG (*patch-based* instrumenters). Examples of SDT instrumenters include DynamoRIO [3], PIN [19], and Valgrind [10]; Dyninst is an example of a patch-based instrumenter [14]. These tools assume that instrumentation does not alter the program representation and thus do not address the validity concern addressed in this work. This assumption holds if instrumentation does not have any cumulative effect on the program (e.g., by saving and restoring registers); however, if instrumentation modifies process state the user must ensure that the resulting binary is valid.

Some binary instrumentation frameworks also allow the user to directly modify original code either at the instruction or function level. Instruction-level modification, as provided by DynamoRIO, Dyninst, and Valgrind, allows users to edit or remove individual instructions but does not provide any guarantee that the resulting program is valid. As a result, these tools allow users to accidentally introduce erroneous control transfers or corrupt address calculations used by indirect jumps. Function-level modification, as provided by Dyninst and PIN, allows users to replace entire functions with new code. However, this approach is not amenable to fine-grained modification within functions.

III. CFG

The control flow graph (CFG) is a familiar representation of the structure of a binary program (e.g., functions and basic blocks) and the relationship between these structures. We use a conventional definition of a CFG as a directed graph whose vertices represent basic blocks and whose edges represent control paths between blocks.

A. Control Flow Abstractions

Our CFG is based on four abstractions: the *interprocedural control flow graph* (consisting of *basic blocks* and *edges*) and *functions*. A CFG is a directed graph $G = (V, E, V_e, V_x, T)$, defined as follows; we show an example CFG in Figure 1:

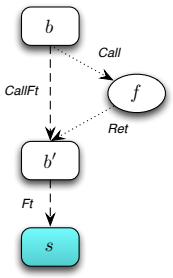


Fig. 1. An example CFG that demonstrates our notation. Blocks (b, b', s) are represented by boxes and edges as arrows that are labeled with their types. We shade snippet blocks, such as s, in blue. Dashed arrows represent *intra*procedural edges, while dotted arrows represent *inter*procedural edges. We summarize functions as ellipses (f).

- The set $V = B \cup \{v_\perp\}$ of vertices corresponding to basic blocks B and a *sink* $v_\perp$,
- The set $E \subseteq V \times V$ corresponding to control flow edges between blocks,
- The sets $V_e \subseteq V$ of entry vertices and $V_x \subseteq V$ of exit vertices, and
- The function $T : E \to \mathcal{T}$ that associates edges with *types*.

We define basic blocks in the conventional way as a consecutive sequence of instructions $b_i = \langle i_m, \ldots, i_n \rangle$ with a single entry instruction i_m and single exit instruction i_n; an instruction may belong to only one block. The in- and out-edges of b_i are denoted $In(b_i)$ and $Out(b_i)$. Unknown control flow is represented by an edge to a unique *sink* $v_\perp$ that contains no instructions and has no out-edges.

Edges are associated with a *type*, which is an element in the set $\mathcal{T} = (Dir, Ft, Cond, CondFt, Ind, Call, CallFt, Ret)$ that represents *direct, fallthrough, conditional taken, conditional fallthrough, indirect, call, call fallthrough*, and *return* edges. Call fallthrough edges link blocks ending with calls to their intraprocedural successors, and may be omitted if the callee does not return.

We define a *function* as the blocks reachable from an *entry block* traversing only intraprocedural edges. Formally, functions are subgraphs of the CFG $f_i = (V_i, E_i, v_i, X_i, T_i)$ where $V_i \subseteq V$, $E_i \subseteq E$, $v_i \in V_i$ is the entry block of the function, $X_i \subseteq V_i$ are the *exit blocks*, and T_i assigns only intraprocedural edge types.

Deriving a CFG from a binary is a difficult problem in its own right. We assume the presence of a parsing algorithm that can derive a CFG from the binary. The design of these algorithms is a challenge due to the presence of both indirect control flow that cannot be statically analyzed and data intermixed with code. We use the ParseAPI [20] recursive-traversal parser [21], [22] that follows statically determinable control flow to discover as much code as possible, and makes use of backwards slicing and heuristic techniques to identify the targets of indirect jumps (e.g., jump tables [23]) and functions that are only reached via indirect calls [24]. To our knowledge, this algorithm has never been published.

Address	Instruction		
61a901	`cmpl`	`$0x0,`	`%gs:0xc`
61a909	`je`	`61a90c`	
61a90b	`lock cmpxchg`	`%ecx,`	`0x31f4(%ebx)`
61a913	`jne`	`...`	

Fig. 2. Example of overlapping blocks from GNU libc on IA-32 compiled with GNU GCC 4.x. The first conditional branch skips past a locking prefix on the compare and exchange instruction; this results in two instructions that overlap. Execution converges at the second branch.

The CFG resulting from this parsing may be incomplete due to unknown indirect control flow, exceptions, and similar constructs. For conventional binaries, this incompleteness can be addressed by making conservative assumptions about such control flow, such as that any function may be the target of an indirect call, and that indirect branches are intraprocedural. In such cases, additional edges can be added to the CFG to represent these assumptions, and we may mark functions as unmodifiable if they contain unknown control flow. If the user is modifying a running process, we may augment our static parse with dynamic analysis techniques, such as those described by Roundy and Miller [25]. These techniques will always present a locally complete CFG.

B. Complicating Cases

Several code structures commonly seen in binaries complicate the mapping of our control flow abstractions onto the binary code. Our goal with such structures is to hide their complexity from the user wherever possible. We describe how we handle two such cases: *overlapping basic blocks* and *overlapping functions*. In this section, we define these structures and how our abstractions map onto them.

Overlapping basic blocks occur when the same sequence of bytes disassembles to two distinct instruction sequences, both of which may be executed by the program. This situation occurs in variable-length architectures because instructions can overlap. One example of such code is shown in Figure 2, which uses a conditional branch to optionally skip a locking prefix on a compare and exchange instruction, resulting in two instructions that overlap. This sequence occurs in many GCC-compiled Linux libraries. Overlapping instruction sequences also commonly occur in obfuscated code. We represent each of these code sequences as distinct collections of basic blocks. This representation allows the user to treat the overlapping code sequences as logically disjoint sequences of instructions.

Overlapping functions occur when multiple functions include the same basic block. Optimizing compilers frequently share common code between functions (e.g., register restores or error code) to increase code density. A similar case occurs in code produced from languages that support multiple entry points into a function (e.g., the Fortran ENTRY statement). We represent this sharing in our interprocedural CFG by allowing multiple functions to contain a single block and define a CFG transformation in Section V-B that eliminates sharing. An alternative representation of such code is as a single function with a set of entry blocks. We chose instead to represent such code as a set of single-entry functions (one per entry point)

to keep a consistent function representation.

C. Inserted Code

Our approach uses the CFG as a mechanism for modifying the structure and execution of a program. However, transformation of the CFG only allows a user to modify or remove existing code, not insert new code. We wish to support such insertion while maximizing use of the CFG as a modification mechanism. Specifically, we wish to prevent inserted code from creating or destroying control flow paths, since those operations must be performed on the CFG instead. We do so by restricting inserted code to be single-entry, single-exit sequences; we call such sequences *code snippets*. This suffices for direct control flow; we discuss our approach for handling indirect control flow in Section VI. Once inserted, a snippet can be transformed as part of the original CFG. This provides users with the capability to construct more complex code sequences (e.g., code that makes calls or ends in a conditional branch) by combining snippets with CFG transformations.

Formally, a snippet s_j is a CFG $(V_j, E_j, v_j, x_j, T_j)$ with a single entry block v_j and exit block x_j; the exit block must have a single fallthrough edge which we link to the sink node. Code snippets may contain internal branching, but may not have explicit branches outside of the snippet. As a result, inserting a snippet does not create or destroy a control flow path. For simplicity, we presume that snippets are specified as a CFG; however, they could be provided as a buffer of assembly that is parsed at insertion time to create a CFG, or in a higher level language.

IV. Validity

Structured binary editing allows the user to modify a binary program while ensuring its structural validity is preserved. Instead of directly modifying the program, we derive a CFG from the binary, transform the CFG, and use the transformed CFG to instantiate a similarly modified program. By ensuring that all transformations preserve the validity of the CFG, we ensure the resulting program is structurally valid. In this section, we define structural validity of binary programs and validity of CFGs. Intuitively, a binary program is *structurally valid* if it will only execute valid instructions; structural validity does not guarantee correct execution or output. Similarly, a CFG is *valid* if it represents a structurally valid binary program and can thus be used to instantiate such a program.

A. Structural Validity

For our definition of structural validity, we represent a binary program as a tuple $P = (C, D)$ where $C = \langle i_0, \ldots, i_m \rangle$ is a sequence of instructions representing the binary code and D represents data. We say P is *structurally valid* if for all inputs, P will only execute instructions from C, and does not treat values from D as instructions or execute unallocated memory. We purposefully use a permissive model of validity that allows constructs such as code in data (as C and D may overlap) and overlapping instruction sequences, which is a common code obfuscation concept.

(a) Original CFG (b) Transformed CFG

Fig. 3. An example of an invalid CFG transformation. Figure (a) shows the original CFG, with $V = \{b_1, b_2\}, V_e = \{b_1\}, V_x = \{b_2\}, E = \{(b_1, b_2)\}$, and T mapping this edge to type Dir. Figure (b) shows the CFG with the edge (b_1, b_2) removed. The second CFG is invalid, as b_1 has no out-edges and is not an exit block, and cannot be instantiated as a structurally valid binary program. Thus, edge removal does not satisfy our constraints and is thus an invalid transformation.

We assume the input program P is valid. This may not be the case, either if P contains code that is modified at runtime by the remainder of the program or if it contains intentionally invalid instructions. In such cases, our approach will preserve the invalidity of the program.

B. CFG Validity

Unconstrained transformation of the CFG may result in a graph that no longer represents a structurally valid binary program. Consider the simple example shown in Figure 3, where a transformation has removed all of the out-edges of a block b_1. As a result, b_1 has no identified successors; since b_1 is not an exit node, then the graph does not represent a binary program and thus cannot be used to instantiate one.

We define CFG validity with *constraints* over each element in the CFG definition. Let $P = (C, D)$ be a structurally valid program as defined above. Then a control flow graph $CFG_P = (V, E, V_e, V_x, T)$ for P is valid if the following constraints hold:

- *Vertex constraint*: The vertex set $V = B \cup \{v_\perp\}$ is valid if every block $b \in B$ is valid by the definition of basic blocks and the sink $v_\perp$ is unique.
- *Edge constraint*: The edge set E is valid if no edge has the sink as a source and all non-exit blocks $b \in (V \setminus V_x)$ have at least one out-edge. That is, E is valid iff $\nexists e \in E$ s.t. $e = (v_\perp, b_i)$ for some block b_i and $\forall b_j \in (V \setminus V_x), \exists e \in E$ s.t. $e = (b_j, b_k)$ for some block b_k.
- *Entry constraint*: The entry set V_e is valid iff $V_e \subseteq V$, $|V_e| \geq 1$, and $v_\perp \notin V_e$.
- *Exit constraint*: The exit set V_x is valid iff $V_x \subseteq V$ and $v_\perp \notin V_x$. Unlike the entry set, the exit set may be empty if the program never terminates.
- *Type constraint*: The type function T is valid if all edges have a type, and the out-edges of each block b are labeled in a way that corresponds to the appropriate architecture.

The type constraint is architecture-specific because each architecture has different control flow instructions with different characteristics; for example, PowerPC and ARM support conditional indirect branches while IA-32 only supports unconditional indirect branches. We represent valid edge types, or *signatures*, in disjunctive normal form, and we define

TABLE I
VALID SIGNATURES FOR IA-32/X86-64, POWERPC, AND ARM

IA-32/x86-64	PowerPC/ARM
(*Dir*) ∨	(*Dir*) ∨
(*Ft*) ∨	(*Ft*) ∨
(*Cond* ∧ *CondFt*) ∨	(*Cond* ∧ *CondFt*) ∨
(Ind^+) ∨	(Ind^+) ∨
(*Call*) ∨	(*Call*) ∨
(*Call* ∧ CallFt) ∨	(*Call* ∧ *CallFt*) ∨
(Ret^+)	(Ret^+) ∨
	(Ind^+ ∧ *CondFt*) ∨
	(*Call* ∧ *CondFt*) ∨
	(*Call* ∧ *CallFt* ∧ *CondFt*) ∨
	(Ret^+ ∧ *CondFt*)

signatures for IA-32/x86-64, PowerPC, and ARM in Table I. We have implemented support for IA-32/x86-64 and PowerPC; an ARM implementation is in the planning stage.

Let $Out(b) = \{e_i, \ldots, e_j\}$ be the out-edges of a block b, and T_b a sequence of types $\langle T(e_i), \ldots, T(e_j)\rangle$; we represent T_b as a sequence since multiple edges may have the same type (e.g., indirect branches). Each sequence T_b is valid if it satisfies the appropriate signature, and T is valid if $\forall b, T_b$ is valid.

As examples, let b be a block with two out-edges e_1, e_2. A type function T_1 that maps $e_1 \rightarrow Cond$ and $e_2 \rightarrow CondFt$ is valid on IA-32, x86-64, and PowerPC. A type function T_2 that maps $e_1 \rightarrow Call$ and $e_2 \rightarrow CondFt$ is valid on PowerPC or ARM but not on IA-32 or x86-64. Finally, a type function that maps $e_1 \rightarrow Cond$ and $e_2 \rightarrow Dir$ is not valid on either architecture since there is no control flow instruction that can produce both a conditional and direct out-edge.

V. CFG TRANSFORMATION ALGEBRA

The major contribution of this work is an algebra of graph transformations for editing the structure and control flow of a binary program. In this section, we define this algebra. We then describe three classes of transformations: *block*, *edge*, and *code insertion*, and provide examples of each class. This discussion assumes that each transformation has no other effect on the CFG; specifically, that indirect control flow is not changed. We discuss our approach to handling indirect control flow in Section VI.

Our structured binary editing algebra is a tuple $BinEdit = (GT, VC)$ where GT is a set of graph transformation *rules* and VC represents the validity constraint defined in Section IV-A. Briefly, a rule $r : L \rightarrow R$ replaces an instance of the subgraph L in a target graph G with the graph R; we represent these rules graphically, as is typical in graph transformation [11], [12]. A rule r is *valid* under the constraint VC if transforming an input graph that satisfies VC results in an output graph that satisfies VC. We show an example graph transformation to demonstrate our notation in Figure 4. Finally, we define composition of two valid rules to be valid.

The transformations in our algebra are purposefully designed to be simple in order to avoid unexpected side-effects on program behavior. We expect users to compose these simple transformations to both create more complex transformations

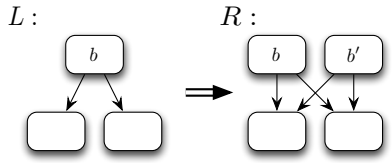
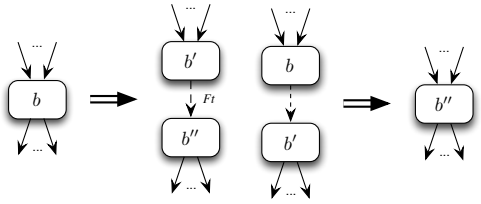
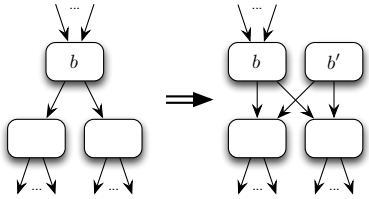


Fig. 4. An example graph transformation. The input subgraph L is on the left, and the replacement R is shown on the right, separated by a double arrow. We use solid arrows to indicate an edge that can be either intra- or inter-procedural, and omit type labels when they are not necessary. Edges to or from blocks not included in the transformation (such as the source edge(s) of b) are omitted for clarity.



(a) Block Splitting (b) Block Joining



(c) Block Cloning

Fig. 5. Example b lock transformations. Block splitting is shown in figure (a); the original block $b = (i_1, \ldots, l_n)$ is split into two blocks $b' = (i_1, \ldots, i_m)$ and $b'' = (i_{m+1}, \ldots, i_n)$. Block joining is shown in figure (b), where two blocks b, b' are combined into one block $b'' = b \cup b'$; note that b' may have no other in-edges except from b. Block cloning is shown in figure (c), where the block b is cloned creating a block b'. Note that cloning copies out-edges but not in-edges.

and to transform more of the CFG. We must ensure that such composition preserves structural validity. This is true by induction. Let T_1 and T_2 be two valid transformations in our algebra and G be a valid CFG. Then $G' = T_1(G)$ must be valid, and similarly $T_2(G') = T_2(T_1(G))$, which is by definition the composition of T_1 and T_2; therefore $T_1 \circ T_2$ is also a valid rule.

A. CFG Transformations

The first class of transformations alter the blocks (nodes) in the CFG. We define four block transformations: *block splitting*, *block joining*, *block cloning*, and *block removal*. These transformations are shown in Figure 5; for reasons of space, we omit block removal. Block splitting divides an existing block into two pieces and joins the resulting blocks with a fallthrough edge. Blocks must be split at an instruction boundary. Block joining reverses this operation. The first block must have a single intraprocedural out-edge that targets the second block; thus, this edge must be typed as either direct or fallthrough. The second block must have a single in-edge from the first block. Block cloning creates a copy of a particular

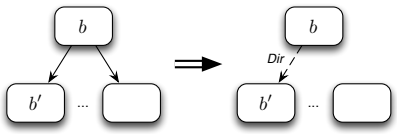
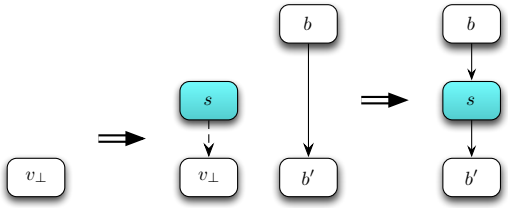
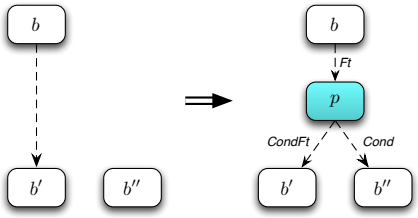


Fig. 6. Example edge transformation that replaces all out-edges of b with a single out-edge to b'. All blocks may have in-edges from blocks not included in the transformation; however, blocks with no in-edges are valid.



(a) Raw Insertion (b) Edge Insertion



(c) Predicate Insertion

Fig. 7. Insertion transformations, with the inserted snippet represented by blocks shaded in blue. Raw insertion is shown in figure (a); the added snippet s is not connected to the original CFG. Edge insertion is shown in figure (b); a new snippet s is inserted between b and b'. Predicate insertion is shown in figure (c); in the original graph b and b' are connected by a direct or fallthrough edge, and we introduce a conditional edge to b'' while converting the original edge to a conditional fallthrough. The block p represents the predicate that selects which edge is taken at runtime.

```
1  Algorithm: Function Cloning
   input : A graph CFG and function f = (V_f, E_f, v_f, X_f, T_f, L_f)
   output: A modified graph CFG' and new function f'
2  CFG' ← CFG, V'_f ← ∅, E'_f ← ∅, v'_f ← v⊥, X'_f ← ∅;
3  for each block b ∈ V_f do
4      if b ≠ v⊥ then
5          b' ← BlockClone(b, CFG');
6      else
7          b' ← v⊥;
8      end if
9      V'_f ← V'_f ∪ {b'};
10     if b = v_f then
11         v'_f ← b';
12     end if
13     if b ∈ X_f then
14         X'_f ← X'_f ∪ {b'};
15     end if
16 end for
17 for each block b' ∈ V'_f do
18     for each out-edge e' = (b', b_t) ∈ Out(b') do
19         E'_f ← E'_f ∪ {e'};
20         if e' is intraprocedural then
21             b'_t ← the clone of b_t;
22             EdgeRedirect(e', b'_t, CFG');
23         end if
24     end for
25 end for
```

Fig. 8. An algorithm for cloning a function f into a function f'. As with block cloning, the entry of f' has no interprocedural in-edges. This algorithm constructs the blocks, edges, entry block, and exit blocks of f'; the construction of the type and context labeling functions are omitted. We begin by cloning each block $b \in V_f$ and constructing V'_f, v'_f and X'_f (lines 3-16). After each block has been cloned, we iterate over each and redirect all intraprocedural out-edges to the corresponding cloned blocks (lines 17-25).

block, including all of that block's out-edges (but not its in-edges). Block removal deletes a block from the graph; the removed block must have no in-edges.

The second class of transformations alter the edges in the CFG. Since these transformations involve no new code, they can only alter or remove existing execution paths; we discuss creating new paths below. We define two edge transformations: *edge redirection* and *edge collapsing*. Edge redirection changes the target of an edge; the source is left unmodified. Edge collapsing replaces all out-edges of a block with a single edge to a selected target block that is typed as *direct*, as shown in Figure 6. One obvious transformation that we do not support is edge *deletion*, which removes an edge from the graph. Applying such a transformation may result in an invalid CFG, since it does not provide an alternative execution path (Figure 3). Similarly, modifying the source of an edge is invalid in our algebra since it may result in an invalid CFG.

The third class of transformations inserts new code into the CFG. We must constrain such insertion to ensure that the resulting CFG is valid; we do this by constraining the transformations that can be used to insert code. In this work we define five insertion transformations: *raw insertion*, *edge insertion*, *predicate insertion*, *call insertion*, and *return redirection*. Examples of these transformations are shown in Figures 7. Raw insertion simply inserts a provided snippet into the CFG, but performs no other operations; as a result, the user must use other transformations to connect its entry and exit with the original CFG. Edge insertion inserts a snippet along an existing single control flow edge; the snippet entry is linked to the source of the edge and the exit is linked to the target. Predicate insertion creates new execution paths by converting an existing direct or fallthrough edge to a conditional pair. This transformation inserts a predicate that controls which edge is executed and a new conditional taken edge e_t, while converting the original edge to a conditional fallthrough e_{ft}. For simplicity, the predicate tests a single register or memory location: if the value is non-zero, e_t is taken; if zero, e_{ft} is taken instead. If more complex predicates are required, a user can create them by composing predicate insertion with edge insertion, and we rely on existing code-optimization techniques to generate efficient code for the desired result. Call insertion allows a user to interpose a function call along an edge; to preserve validity, the original edge must be typed as either direct or fallthrough. Finally, return redirection replaces the out-edges of a block with an immediate return to the caller.

B. Function Transformation Examples

The transformations above allow the user to manipulate the interprocedural CFG at the level of blocks and edges. These transformations can be composed to manipulate functions as well. We describe three examples of such operations: function *cloning*, *removal*, and *unsharing*. Function cloning creates a copy f' of a function f by cloning all blocks in f and redirecting edges. Note that the entry block v'_f of f' will have no interprocedural in-edges (as it was just created and thus is not reachable from outside of f') but may have intraprocedural in-edges from other blocks in f'. We define an algorithm for function cloning in Figure 8. Function removal destroys a function f and blocks that belong only to f; for this transformation to be legal, the function entry block can have no interprocedural in-edges. For reasons of space we omit the formal description of this algorithm.

Finally, function unsharing eliminates sharing of blocks between functions (as described in Section III-B). We do this by cloning all shared basic blocks; we do not copy the sink node if it is shared to since the sink node must be unique. This algorithm is similar to function cloning, but has two key differences. First, function unsharing only copies shared blocks rather than all blocks. Second, it does not create a new function, instead copying blocks within the original.

VI. Indirect Control Validity

In the previous sections, we described an approach to modifying a binary program by transforming its CFG. This approach made the simplifying assumption that these transformations had no effect on the remainder of the CFG. This simplification does not hold if a program uses indirect branches or calls that use addresses determined at runtime. In this section, we describe how transforming the CFG may alter indirect control flow and describe an approach for safely determining when this may occur.

Programs rely on indirect control flow for language features such as multi-way branches, function pointers, or virtual functions. At the binary level, these constructs are similar: the program first calculates an address and then uses an *indirect control transfer* (ICT) to branch or call to this destination. These address calculations may be altered by a transformation of the CFG, such as by inserting new code that changes intermediate values used by the calculation. Such an altered calculation may change the possible destinations of the corresponding ICT. This, in turn, may result in an invalid CFG that does not properly represent these new destinations. The transformations presented in Section V do not take this case into account, instead assuming that the only control flow effects on the CFG are those explicitly represented in the transformation.

This problem is compounded by the difficulty of statically determining indirect control flow. Compiler-emitted address calculations (e.g., jump tables) frequently can be successfully analyzed and their destinations represented in the CFG [23]. However, more complex calculations, such as those used for function pointers or in optimized code, are either expensive to analyze [26] or defeat analysis entirely. We label such ICTs in our code with edges to the sink node, representing statically unknown control flow.

We identify which ICTs are affected by a particular CFG transformation and how they are affected as follows. We use static analysis to identify which ICTs are affected; clearly, if an ICT is not affected by a transformation then its destinations will not be changed. If the ICT is affected, we attempt to statically determine its new destination set. If each such destination is valid, we update the CFG to match; if a destination is invalid we inform the user and invalidate the transformation. Finally, if we fail to resolve the possible destinations via static analysis, there is a spectrum of responses: we could invalidate the transformation, allow the transformation but insert runtime monitoring code, or allow the transformation and warn the user of the CFG possibly becoming invalid. We allow the user to select which response they prefer, which allows a user to be conservative if they want while allowing transformations that the user may know is valid even if our analysis fails.

We define a destination to be valid if it targets the entry of a known basic block (for an indirect jump) or known function (for an indirect call). This is purposefully a restricted definition following our policy to minimize the possible secondary effects of a CFG transformation. More relaxed policies are possible; for example, we could define a destination to be valid if it targeted a known instruction, and implicitly split a block at such a targeted instruction.

Our analysis is performed over the transformed CFG, which may include inserted code snippets as well as original code. This analysis can be simplified if the user provides semantic *summaries* of inserted code snippets; however, such summaries are not required. When we encounter a snippet during our analysis, we apply a summary if it is available and continue instead of having to analyze the entire snippet. This simplification biases towards cases where inserted code has no cumulative effect on the address calculation (e.g., instrumentation) but proving this with analysis of the inserted code would be expensive. Specifically, if a snippet has no cumulative effect, we can omit its effects entirely and quickly determine that the ICT was not affected.

A. Identifying Affected Indirect Jumps

Intuitively, an ICT is affected if a CFG transformation interferes with the address calculation preceding the ICT. Let i_j be such an ICT in a function f in the original CFG, t a transformation, and f' be the resulting transformed function with corresponding ICT i'_j. We define the address calculation corresponding to i_j as the *backwards slice* ac_j, bounded at function entry [27]; we discuss our selection of this bound below. Clearly, if t has no effect on ac_j then the ICT i_j will also be unaffected; similarly, if t alters ac_j then i_j is affected.

This is essentially a dataflow analysis problem, and thus it is insufficient to see if ac_j is directly affected by t. Instead, we calculate the new backwards slice ac'_j from i'_j in f'. If $ac_j \equiv ac'_j$, then no modification of the address calculation has occurred and thus $i_j \equiv i'_j$; otherwise, we conclude i_j

is affected and perform further analysis to determine its new destinations.

We bound the backwards slice at unanalyzable memory accesses or the entry of the function. Our current implementation can resolve stack accesses or accesses to fixed memory locations; all others, such as a pointer-based heap reference, terminate the slice. If we reach an unanalyzable memory access, we make the conservative assumption that the ICT was affected by the transformation. If we reach the entry of the function we can safely end the slice since an intraprocedural transformation will not alter the function's inputs.

B. Identifying New Destinations

Once we have identified the ICT i_j that is affected by the transformation t, we must determine its new possible destination addresses; let this be a set of addresses $D = \{a_m, \ldots, a_n\}$. We say i_j is *valid* if $\forall a_k \in D$, a_k is the entry of a block b_k. We analyze D as follows. First, we use symbolic evaluation to convert the slice ac_j into a symbolic representation $eval_j$ [28]. We then attempt to evaluate $eval_j$ to determine its possible outputs. If this evaluation succeeds, we examine each possible destination to determine if it is valid; if all destinations are valid, we update the out-edges of the ICT in the CFG and inform the user of each changed edge. If a destination is invalid, we invalidate the transformation. Finally, if the evaluation fails, we perform one of three actions depending on the user's preference for handling failed analysis. The first option invalidates the transformation; this option is conservative, but may invalidate legal transformations. The second option inserts runtime monitoring code that raises an exception if the indirect branch targets an invalid destination; this option allows possibly invalid transformations, but may result in unexpected runtime failures. The third option removes all out-edges from the ICT and replaces them with a single edge to the sink node; this option is maximally permissive, but may allow invalid transformations and thus may be dangerous.

C. Preserving Other Dataflow Characteristics

The above problem can be cast as detecting when a transformation of the CFG affects a dataflow property of the program; in this case, the values generated by an address calculation. The same approach that we use to address this problem can be applied to detecting other common errors that can also be cast as dataflow characteristics, such as determining whether a transformation will cause the program to use an uninitialized value or access unallocated memory. We intend to investigate these other common and frequently unintended errors in future work using this analysis framework.

VII. Evaluation

We evaluated our structured binary editing approach by applying it to dynamically patching, or *hot patching*, an Apache web server. Hot patching is complicated by the differences in binaries generated by different compilers, compiler versions, or optimization levels. Ideally, a hot patching tool should be independent of a particular compilation of the code, rather than needing to be specialized for every possible binary. Thus, the method used for code identification, construction, and removal should be related to structural characteristics rather than a particular instruction sequence. For example, the location where local variables are stored may vary, or instructions may be reordered by different compilers.

Current binary modification toolkits are not well-suited to performing hot patching. DynamoRIO [3] and Valgrind [10] can perform instruction-level modification and thus can replace code. However, they do not identify the structural characteristics helpful for identifying where to apply a patch. Furthermore, they require the user to manually construct the replacement code. Therefore, tools based on these toolkits would be limited to particular versions of the binary to be patched. PIN [19] provides function-level modification, but otherwise suffers the same problems as DynamoRIO and Valgrind. Dyninst [14] represents the binary as a CFG, thus easing identification, and provides a high-level language for constructing the replacement code sequence. However, like PIN, Dyninst only supports function-level replacement. Our structured binary editing technique addresses this lack by providing CFG modification.

We investigated hot patching security vulnerabilities in a running Apache HTTPD web server [29]. We selected Apache for three reasons. First, it is widely used. Second, security flaws, as well as the patches necessary to fix these flaws, are widely published and available. Third, as a long-running process, Apache is an excellent test of our ability to modify a running process without corrupting its structure. We constructed a single tool that dynamically patches three security vulnerabilities (CVE-2011-3368, CVE-2011-3607, and CVE-2012-0021 [30]) in a running Apache process.

This evaluation has three goals. First, we should be able to patch an unmodified, executing Apache HTTPD web server. Second, we should be able to patch versions of the same server as compiled on different systems, which requires identifying the locations to patch by structural characteristics (e.g., a subgraph of the CFG) rather than a literal sequence of instructions. Third, we should be able to prepare the patch from the corresponding source code patch instead of manually crafting it in assembly code. We accomplished these goals as follows. We prepared our target binaries by compiling the appropriate versions of Apache from source using default configuration options and several different versions of GNU GCC (4.1 through 4.6). We used different compiler versions to ensure that our location match did not depend on the particular idioms used by a single version of GCC. Finally, we wrote each patch in Dyninst's high-level AST language [14] instead of assembly code.

For conciseness, we describe how we created our hot patching tool for the CVE-2011-3368 vulnerability; the process for patching CVE-2011-3607 and CVE-2012-0021 was similar. This vulnerability allowed a user to use a carefully crafted URL to gain full internal network access from a DMZ web server. We began by examining the Apache source code and the published patch to gain an understanding of the

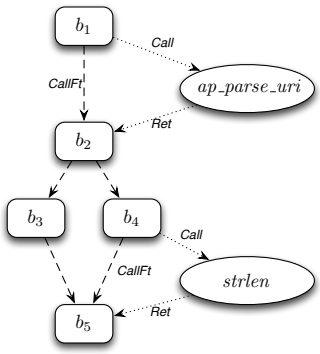


Fig. 9. CFG fingerprint for CVE-2011-3368.

```
  1  ap_parse_uri(r, uri);
  2
+ 3  if (r->method_number != M_CONNECT
+ 4      && !r->parsed_uri.scheme
+ 5      && uri[0] != '/'
+ 6      && !(uri[0] == '*' && uri[1] == '\0')) {
+ 7        r->args = NULL;
+ 8        r->hostname = NULL;
+ 9        r->status = HTTP_BAD_REQUEST;
+ 10       r->uri = apr_pstrdup(r->pool, uri);
+ 11 }
  12
  13 if (ll[0]) {
```

Fig. 10. Code listing for the CVE-2011-3368 security patch; the lines prefixed with a "+" are inserted. For clarity, this listing omits call to a logging function `ap_log_rerror`.

vulnerability and to locate where to apply our modifications; we show the patch file in Figure 10.

We then examined the CFG of the binary to identify the corresponding point in the binary that required patching. Interestingly, we found from an examination of the binary that the patched function (`read_request_line`) was not present in the server binary. This function is declared static in its source file, resulting in the compiler inlining it into its caller, `ap_read_request`. However, we were still able to identify its characteristic CFG and prepared the fingerprint shown in Figure 9. This fingerprint matched a portion of the `ap_read_request` function that had incorporated `read_request_line`. This fingerprint was unique in the binary; however, this may not be true for binaries built for other compilers. We designed our hot patching tool to exit without modifying the binary if it either fails to find the fingerprint or finds multiple matches, since this would indicate the fingerprint may have falsely identified the location to modify. We performed this step manually; however, we believe it could be partially automated by generating the expected CFG from source code and then manually refining the fingerprint.

Next, we converted the code added by the patch file into a Dyninst code snippet. We divided the new code into two sections: a *condition* that identified when server was being attacked and a *patch* that took corrective action. Both the condition and the patch required access to two local variables in `read_request_line`, *r* (a status structure) and *uri* (the input URI). Since Apache is compiled with debugging information, we were able to identify *r* and its fields. Debugging information for *uri* was not present. However, it is an argument to a function call to `ap_parse_uri` immediately before the patch location. We could not use the argument register directly because it was modified by the call; instead, we used Dyninst's dataflow analysis capabilities to identify the register that contained the authoritative value for *uri*. Once we had created AST nodes for these two variables, building the snippet was a straightforward operation of manually translating C code. As with the previous step, we generated the code snippet manually using the Dyninst DynC C-like language [31]

Finally, we constructed a CFG transformation that injected the snippet into the binary; we show the transformed CFG

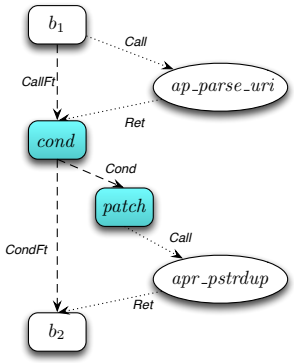


Fig. 11. The resulting CFG after our transformations were applied to insert the patch shown in Figure 10. User-inserted snippets are shown in blue (*pred* and *patch*), and automatically generated code in green (*cond* and *call*). The *pred* snippet corresponds to lines 3-6 and the *patch* snippet to lines 7-10; for clarity, we have omitted a logging call to `ap_log_rerror`. The *cond* block represents the conditional branch used to implement the **if** statement in line 3, and the *call* block represents the call to apr_pstrdup in line 10. We composed three types of transformations to create this CFG: interception insertion, predicate insertion, and call insertion.

subgraph in Figure 11. This injection was straightforward, as the patch did not contain complex control flow. We then verified that our tool closed the security flaw while not impacting the normal execution of the web server.

This case study demonstrates that our structured binary editing approach is capable of patching security vulnerabilities in an unmodified, executing Apache web server. We are able to use the same tool on Apache binaries compiled with several different compiler versions. This is due to our use of the CFG to identify the patch site and apply the patch; although the underlying instructions in each compiled version of Apache are different, their CFGs are the same.

In all, our tool consisted of 498 lines of code, of which 93 were responsible for identifying the patch location, 187 were responsible for creating the snippet, and 120 were responsible for transforming the CFG; the remainder was utility code. The small size of our tool is due to two factors: the expressive power of our CFG transformations and the analysis capabilities of the Dyninst toolkit. We believe these sizes would be representative of other uses of structured binary editing.

VIII. Conclusion

We have presented structured binary editing, a technique that uses the CFG as a mechanism for binary program modification. Structured binary editing overcomes the weaknesses of instruction-level binary modification by ensuring the resulting binary is structurally valid. We have defined an algebra of valid CFG transformations and provided several example transformations. We have also described a slicing-based dataflow analysis that determines if a transformation will alter indirect control flow. We have implemented our approach in the Dyninst binary analysis and instrumentation framework, and created a prototype hot patching tool. We applied this tool to a running Apache web server to show that we can change local program behavior without causing undesired side effects.

Our prototype will be included in the next public release of the Dyninst project. In the meantime, code is available upon request. We intend to address the limitations of our current approach in future work. First, we wish to improve our indirect control transfer (ICT) analysis; this includes both our heuristics for bounding the analysis as well as our determination of the new destinations of the ICT. Second, we wish to supplement the code snippet summarization requirement with automated analysis of each snippet. Third, we will continue to build more complex transformations.

Acknowledgment

We thank the reviewers for their insightful comments and suggestions. This research funded in part by Department of Homeland Security grant FA8750-10-2-0030 (funded through AFRL), National Science Foundation grants OCI-1032341, OCI-1032732, and OCI-1127210, and Department of Energy grants DE-SC0004061 and DE-SC0002154.

References

[1] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *International Conference on Software Engineering (ICSE)*, Shanghai, China, May 2006.

[2] N. Loriant, M. Ségura-Devillechaise, and J. Menaud, "Server protection through dynamic patching," in *Pacific Rim International Symposium on Dependable Computing (PRDC)*, Changsha, Hunan, China, December 2005.

[3] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *First Annual International Symposium on Code Generation and Optimization*, San Francisco, CA, USA, March 2003.

[4] B. P. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, B. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn parallel performance measurement tool," *IEEE Computer*, vol. 28, no. 11, pp. 37–46, November 1995.

[5] J. Newsome, D. Brumley, D. Song, J. Chamcham, and X. Kovah, "Vulnerability-specific execution filtering for exploit prevention on commodity software," in *13th Annual Network and Distributed Systems Security Symposium*, San Diego, CA, USA, February 2006.

[6] W. Drewry and T. Ormandy, "Flayer: exposing application internals," in *Workshop on Offensive Technologies (WOOT)*, Boston, MA, USA, August 2007.

[7] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Security and Privacy (SP)*, Oakland, CA, USA, May 2007.

[8] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," in *Security and Privacy (SP)*, Oakland, CA, USA, March 2007.

[9] H. Pan, K. Asanović, R. Cohn, and C.-K. Luk, "Controlling program execution through binary instrumentation," *SIGARCH Computer Architecture News*, vol. 33, pp. 45–50, December 2005. [Online]. Available: http://doi.acm.org/10.1145/1127577.1127587

[10] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Programming Language Design and Implementation (PLDI)*, San Diego, CA, USA, June 2007.

[11] R. Heckel, "Graph transformation in a nutshell," in *Electronic Notes in Theoretical Computer Science*. Elsevier, 2006, pp. 187–198.

[12] M. Andries, G. Engels, A. Habel, B. Hoffmann, H. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer, "Graph transformation for specification and programming," *Science of Computer Programming*, vol. 34, pp. 1–54, 1996.

[13] Paradyn Project, "PatchAPI: An application program interface for binary patching," 2012.

[14] B. Buck and J. Hollingsworth, "An API for runtime code patching," *Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 317–329, Winter 2000.

[15] K. Van Put, B. De Sutter, M. Madou, B. De Bus, D. Chanet, K. Smits, and K. De Bosschere, "Lancet: A nifty code editing tool," in *Program Analysis for Software Tools and Engineering (PASTE)*, Lisbon, Portugal, September 2005.

[16] B. De Bus, B. De Sutter, L. Van Put, D. Chanet, and K. De Bosschere, "Link-time optimization of ARM binaries," in *ACM Conference on Languages, Compilers, and Tools (SIGPLAN/SIGBED)*, Washington, D.C., June 2004.

[17] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization (CGO)*, Palo Alto, CA, USA, March 2004.

[18] R. Wilson, R. French, C. Wilson, A. S., J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy, "SUIF: An infrastructure for research on parallelizing and optimizing compilers," *ACM SIGPLAN Notices*, vol. 29, 1994.

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

[20] Paradyn Project, "ParseAPI: An application program interface for binary parsing," 2012. [Online]. Available: http://www.paradyn.org/html/parse7.0.1-features.html

[21] C. Cifuentes and K. Gough, "Decompilation of binary programs," *Software - Practice & Experience*, vol. 25, no. 7, pp. 811–829, July 1995.

[22] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," in *IEEE 9th Working Conference on Reverse Engineering*, Richmond, VA, USA, October 2002.

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

[24] L. Harris and B. P. Miller, "Practical analysis of stripped binary code," in *Workshop on Binary Instrumentation and Applications*, St. Louis, MO, USA, September 2005.

[25] K. A. Roundy and B. P. Miller, "Hybrid analysis and control of malware binaries," in *Recent Advances in Intrusion Detection (RAID)*, Ottawa, Canada, September 2010.

[26] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *International Conference on Compiler Construction*, New York, NY, USA, April 2004.

[27] A. Kiss, J. Jasz, G. Lehotai, and T. Gyimothy, "Interprocedural static slicing of binary executables," in *Source Code Analysis and Manipulation*, Amsterdam, The Netherlands, September 2003.

[28] P. Saxena, P. Poosankam, S. McCamant, and D. Song, "Loop-extended symbolic execution on binary programs," in *18th International Symposium on Software Testing and Analysis (ISSTA)*, Chicago, IL, USA, March 2009.

[29] Apache Project, "Apache httpd daemon," November 2011. [Online]. Available: http://www.apache.org

[30] National Vulnerability Database, November 2011. [Online]. Available: http://web.nvd.nist.gov/

[31] Paradyn Project, "DynC: An instrumentation language for specifying snippets," 2012. [Online]. Available: http://www.paradyn.org/html/dync1.0-features.html