M.Sc. dissertation

Static data flow analysis and constraint solving to craft inputs for binary programs

Name:	Tim Blazytko
E-Mail:	tim.blazytko@rub.de
University:	Ruhr-Universität Bochum
Chair:	Chair of Systems Security
1. Referee:	Prof. Dr. Thorsten Holz
2. Referee:	Dipl.-Inform. Behrad Garmany
Submission date:	27th November 2015

Abstract

In the context of binary program analysis, input crafting describes the task of finding an input that directs the control flow from the user input to a defined location in the program. One known strategy to automatise input crafting is symbolic execution, which operates symbolically on program paths and derives path constraints that are solved by an SMT solver. This work describes a novel approach to input crafting, based on techniques of bounded model checking. The main idea is to unroll the control flow graph of a function up to a certain bound and transform it into a formula of first-order logic. An SMT solver decides the input crafting problem on the basis of this formula. For this purpose, data flow analysis and path selection are encoded as a logical problem. In this dissertation, the process of bounded model checking for input crafting on the binary level are derived and its feasibility, as well as its efficiency, are evaluated. Therefore, an architecture-independent framework for bounded model checking on the binary level has been implemented. The results show that this approach works efficiently for different architectures and for real-world binaries, such as *WPA supplicant*. In addition, it is demonstrated that SMT solvers exploit the structure of programs and solve the path selection problem with great efficiency. The main problems of this approach are the necessity to precisely model the program semantics, as well as a lesser degree of efficiency for nested memory writes. As a consequence, API functions must be included in the program semantics; function calls decrease the efficiency. Finally, improvements on the implementation level and further research on the basis of bounded model checking are discussed in the context of application security.

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Ich erkläre mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

Datum	Tim Blazytko

Contents

List of figures

List of tables

List of definitions

List of abbreviations

API	application programming interface
BMC	bounded model checking
CFG	control flow graph
CNF	conjunctive normal form
DAG	directed acyclic graph
DNF	disjunctive normal form
DF	dominance frontier
DFS	depth-first search
DPLL	Davis-Putnam-Logemann-Loveland
DU	definition-usage
ELF	Executable and Linkable Format
FOL	first-order logic
IL	intermediate language
IR	intermediate representation
LHS	left-hand side
LLBMC	low-level bounded model checker
LLVM	Low Level Virtual Machine
PE	Portable Executable
RHS	right-hand side
SAT	satisfiability
SCC	strongly connected component
SMT	satisfiability modulo theory
SSA	static single assignment

1 Introduction

This chapter introduces bounded model checking as a means to craft inputs for binary programs. After presenting the motivation, the related work will be discussed. Then, the research question as well as the contributions of this dissertation will be formulated. Finally, the dissertation structure will be outlined.

1.1 Motivation

Crafting inputs for binary programs is a common task in the field of reverse engineering and application security. In the area of software piracy, input crafting will be used for *keygenning*, in which a valid serial number is generated to activate a software application (cf. Section 11 [1]). In cryptography, a cryptosystem is insecure if it is possible to craft the key for a known plaintext/ciphertext pair (cf. Section 1.13.1 [2]). In the context of exploit development, input generation will be utilised to determine whether vulnerable code can be reached by user input [3].

In the last decade, methods of static program analysis and software verification were used to automatise this task in a broader context. *Data flow analysis* (cf. Section 9.2 [4]), *symbolic execution* [5] and *constraint analysis* [6] are three of those methods that are frequently utilised [3, 7, 8]. In the context of input crafting, the first method acquires information about the flow of user input in a control flow graph. Then, only the paths that are affected by the user input have to be considered. In dependence on the symbolic user input, the second method derives a set of logical constraints of a program path. These constraints will be resolved by a solver for *satisfiability modulo theories (SMT)* [9]. If the constraints can be satisfied, a valid user input will be returned.

The main limitation of these techniques is their hardness, in the general case. For instance, the satisfiability modulo theories are undecidable [10] and, in terms of symbolic execution, the amount of paths grows exponentially in the number of branches [11]. Nevertheless, heuristics for path selection [7] and the ability of SMT solvers to exploit the structure of problems [12] allow the application of those techniques on real-world programs.

Bounded model checking [13] is a technique that has not much gained attention in the contexts of reverse engineering or binary application security. In contrast to symbolic execution, which creates a logical formula of a path, bounded model checking generates a logical formula consisting of a control flow graph, which will be unrolled up to a certain bound, and properties that have to hold. This formula will be solved by an SMT solver, which will, if possible, return a satisfying assignment.

Since SMT solvers exploit the structure of problems, they may operate efficiently on data flow analysis and path selection problems. Combined with bounded model checking, the SMT solver itself may decide the input crafting problem. The goal of this dissertation is to determine the feasibility and efficiency of input crafting for binary programs, based on bounded model checking.

1.2 Related work

Symbolic execution was introduced by Boyer, Elspas and Levitt [14] in 1975 and has been used for program testing and program proving. In the context of program testing, symbolic execution has been utilised for input crafting, known as test case generation [7, 14, 15, 16, 17]. In 2008, a state-of-the-art symbolic execution engine, *KLEE* [7], was introduced, which operates on the intermediate presentation of *LLVM* [18] and presents heuristics for efficient path selection.

It is well-known that SMT solving is undecidable in general case and that the satisfiability problem is NP-complete [10]. In recent years, active research on efficient SMT solvers for various applied problems has been done [19, 20, 21].

As a consequence, symbolic execution and SMT solving will be used frequently in the context of application security. For instance, *AEG* [3] (2011), *Mayhem* [22] (2012) and *SAGE* [8] (2012) apply those techniques for automated exploit generation, both static and dynamic, as well as on source code and the binary level. *FuzzBALL* [23] (2011) utilises symbolic execution and SMT solving in the area of fuzzing. In addition, *Firmalice* [24] (2015), whose binary analysis platform has been released as *angr* [25], detects vulnerabilities in firmware. Finally, Vanegue, Heelan and Rolles [12] (2012) proved the power of SMT solvers for program analysis in the context of vulnerability checking, exploit generation, input crafting and cryptanalysis, amongst others.

In the field of hardware verification on the basis of SAT solvers, *bounded model checking* [26] (1999) has been derived from *model checking* [27] (1982) and, especially, from *symbolic model checking* [28] (1992). Armando, Mantovani and Platania [13] (2006) introduced bounded model checking on the basis of SMT solvers for software. Bounded model checking has also been used to verify C code [29] (2004). On the assembly level, it has been applied to the verification of embedded C programs [30] (2005), concurrent programs [31] (2005) and software for microcontrollers (2010 [32] and 2014 [33]). When applied to security, bounded model checking has been used for verifying and analysing security protocols (2004 [34] and 2008 [35]), web applications [36] (2004) and security properties of control flow graphs [37] (2001). Sinz, Falke and Merz [38] (2010) introduced the *LLBMC (low-level bounded model checker)* [39], which operates on the intermediate representation of LLVM and locates bugs as illegal memory access, integer overflows and others in C programs. However, bounded model checking has not been applied to binary application security by now.

1.3 Research question

To close the gap in research presented earlier, this dissertation examines whether the application of bounded model checking is feasible and efficient to craft inputs for binary programs. To provide an answer to this question, a process to apply bounded model checking to binary programs must be developed first. The main problem thereby lies in the logical encoding of the control and data flow. In addition, input crafting has to be expressed as a problem of bounded model checking. Furthermore, the research question inquires about the strengths and weaknesses as well as the limits of this approach.

1.4 Contributions

The contributions of this dissertation are twofold. First, a novel approach to craft inputs for binary programs, on the basis of bounded model checking, will be presented. The efficiency as well as the strengths and weaknesses of this approach will be discussed. Second, *Cylyx*, a cross-platform and architecture-independent binary analysis framework for bounded model checking will be introduced. Cylyx is the first publicly existing framework for bounded model checking on the binary level and it forms the basis for other use cases, such as vulnerability detection or deobfuscation.

1.5 Dissertation structure

Chapter 2 lays the theoretic foundations required for binary input crafting on the basis of bounded model checking. It describes graph-theoretic concepts, formally defines the input crafting problem and outlines the main ideas of symbolic execution and data flow analysis. Additionally, the theory and terminology behind SMT solvers and bounded model checking is introduced, as well as the binary analysis framework, Miasm, and its intermediate representation, Miasm IR. On the basis of this theory, the process of bounded model checking to craft inputs for binary programs is discussed in detail in Chapter 3: the general idea is derived, the assumptions and prerequisites are mentioned, the main steps – graph transformations, rewriting the intermediate representation and static single assignment – highlighted and, finally, the memory model, as well as the procedure of formula generation, explained. After that, Chapter 4 evaluates the introduced approach and answers the research question. To this end, tests will be applied in five case studies in order to scrutinise the advantages, disadvantages and robustness of the approach. Lastly, Chapter 5 summarises the results and indicates future work.

2 Foundations and concepts

This chapter lays the theoretical groundwork for input crafting via bounded model checking. For this purpose, graph-theoretic concepts will first be introduced, which facilitate the analysis of the control flow of assembly code. Subsequently, the input crafting problem will be defined. Then, symbolic execution – a common technique used for input crafting – will be discussed. Building on that, the main concepts that will be used in this work for input crafting are to be described; namely, they are static data flow analysis and constraint solving. For this, ideas of data flow analysis will be explained. Following this, satisfiability modulo theories, those behind the constraint solving method, will be elucidated. After clarifying the logical background, bounded model checking, a technique based on said theory, will be presented. Lastly, a brief introduction to the Miasm framework will be given, with special attention paid to its intermediate representation.

2.1 Concepts of graph theory

In this section, graph-theoretic concepts, with the focus on analysis of control flow graphs, will be explained. First, basic terminology of graph theory will be defined. Second, control flow graphs will be introduced. Third, loops in control flow graphs will be discussed. To provide consistency and simplicity, all definitions in this section will be adapted from standard definitions found in the literature of graph theory and compiler construction [4, 40, 41].

2.1.1 Basic terminology

A *directed graph* is a data structure which connects a set of nodes by arrows called oriented *edges*. An oriented edge from a node a to a node b indicates a relation from a to b (cf. Section 1.2 [40]).

Definition 2.1.1 (directed graph). *A directed graph or digraph is a pair $G = (V, E)$, where V is a non-empty finite set whose elements are called vertices or nodes and E is a finite set of ordered pairs of distinct vertices, which are called arcs or edges.*

For instance, let $G = (V, E)$ be a directed graph. Then, the set of vertices will be defined as $V = \{a, b, c, d, e, f, g\}$ and the set of edges $E = \{(a, b), (a, c), (b, d), (b, e), (d, f), (e, f), (e, b), (g, d)\}$. The directed graph is illustrated in Figure 2.1.

Hence, terminology describing traversals on directed graphs can be defined. Following, the terms *walk*, *path* and *cycle* will be distinguished (cf. Section 1.4 [40]).

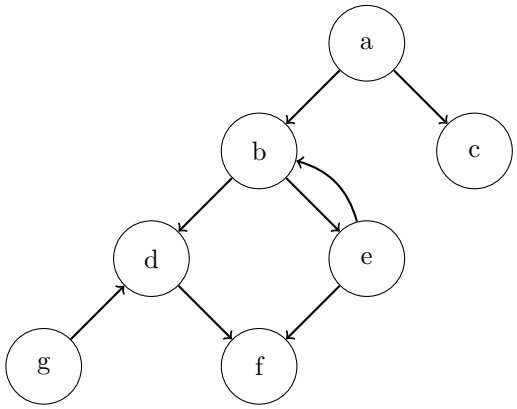

Figure 2.1: Directed graph with two heads and one cycle

Additionally, the concepts of *predecessors* and *successors* of nodes will be defined (cf. Section 1.2 [40] and Section 1.5 [40]). In both cases, the definitions are adapted from the literature.

Definition 2.1.2 (walk, path, cycle). *A (directed) walk in a directed graph G is a sequence of vertices $(v_0, v_1, \ldots, v_n)$ along the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n)$. A walk of distinct v_i is said to be a (directed) path. A (directed) cycle is a walk where $v_0 = v_n$ and where $\{v_1, v_2, \ldots, v_{n-1}\}$ are distinct.*

Definition 2.1.3 (predecessor, successor, reachable, head, tail). *For an edge $e = (v_i, v_j)$ in a directed graph $G = (V, E)$, v_i will be called the direct predecessor of v_j and v_j the direct successor of v_i. If there exists a walk from a vertex v_i to a vertex v_j and $(v_j, v_k) \in E$, then v_i will be said to be a predecessor of v_k and v_k to be a successor of v_i. The set of predecessors of a vertex v will be referred to as* $\mathrm{PRED}(v)$*; the set of successors of a vertex v will be referred to as* $\mathrm{SUCC}(v)$*. Equally, the set of direct predecessors of a vertex v will be denoted as* $\mathrm{DPRED}(v)$ *and the set of direct successors* $\mathrm{DSUCC}(v)$*. If a vertex v_j is a successor of a vertex v_i, then v_j is reachable from v_i. Additionally, a vertex is reachable from itself. A vertex with no predecessors will be called head or root, a vertex with no successors will be called tail or leaf.*

Regarding Figure 2.1, (a, b, e, b, d, f) is a walk, but not a path; (a, b, e, f) is a path. (b, e) is a cycle with no nodes between start and end. The sets of predecessors of c and f are $\mathrm{PRED}(c) = \{a\}$ and $\mathrm{PRED}(f) = \{a, b, d, e, g\}$; the direct predecessors of f are d and a, $\mathrm{DPRED}(f) = \{a, d\}$. The nodes a and g are heads of the graph because they do not have any predecessors; c and f are tails of the graph because they do not have any successors. There exists a path from a to c, but not from g to c. Therefore, c is reachable from a, but not from g.

Informally, a part of a directed graph can also be a directed graph. A *subgraph* of a directed graph is a graph whose vertices and edges are subsets of the sets of vertices and edges of the directed graph (cf. Section 1.2 [40]).

Definition 2.1.4 (subgraph). *A directed graph $G' = (V', E')$ is said to be a subgraph of a directed graph $G = (V, E)$, denoted as $G' \subseteq G$, if the following holds:*

1. $V' \subseteq V$.
2. $E' \subseteq E$.
3. $\forall e = (v_1, v_2) \in E' : v_1 \in V' \wedge v_2 \in V'$.

The third characteristic means that every node which is part of an edge in the set of edges must also be in the set of nodes. For instance, let the set of vertices be $V' = \{a, b, c\}$ and let the set of edges be $E' = \{(a, b), (a, c)\}$. Then, $G' = (V', E')$ is a subgraph of G, the graph in Figure 2.1. However, let the set of vertices be $E^* = \{(a, b), (b, d)\}$. In that case, $G^* = (V', E^*)$ is not a subgraph of G because $d \notin V'$.

As of now, *strongly connected components* can be introduced. In a strongly connected component, every node is reachable from every other node (cf. Section 1.5 [40]) [42].

Definition 2.1.5 (strongly connected component (SCC)). *A strongly connected component or SCC of a directed graph $G = (V, E)$ is a subgraph $G' = (V', E')$ of G in which paths from v_i to v_j and from v_j to v_i exist $\forall v_i, v_j \in V'$ with $v_i \neq v_j$. A strongly connected component is maximal if no vertices or edges from G can be added to G' while remaining strongly connected.*

To illustrate, each of the nodes a, c, d, f and g from the graph in Figure 2.1 is strongly connected to itself, while $G' = (V', E')$, with $V' = \{b, e\}$ and $E' = \{(b, e), (e, b)\}$, is a strongly connected component between b and e. Since no other edge or node can be included, these strongly connected components are maximal.

On the basis of Definition 2.1.1, in a directed graph, an edge is a pair of distinct nodes. This property implies that an edge from a node to itself cannot exist; *self-loops* are not possible. (However, this exclusion does not restrict handling of self-loops in control flow graphs, as Section 2.1.5 will show.) As a result, a *directed acyclic graph* can be defined as follows [43].

Definition 2.1.6 (directed acyclic graph (DAG)). *A directed acyclic graph or DAG $G = (V, E)$ is a directed graph with no directed cycles. The number of vertices $|V| = n$ is equal to the number of strongly connected components in G. In other words, each vertex in G is only strongly connected to itself.*

The graph G illustrated in Figure 2.1 is not acyclic because the nodes b and e are strongly connected. The subgraph $G' = (V', E')$, with $V' = \{d, f, g\}$ and $E' = \{(g, d), (d, f)\}$, is a directed acyclic graph since every node d, f and g is only strongly connected to itself.

2.1.2 Depth-first traversal

After the basic terminology required for control flow analysis has been defined, one algorithmic concept called *depth-first search (DFS)* or *depth-first traversal*, which is the fundament for a multitude of graph analysis algorithms [43, 41, 42, 44, 45], will be introduced. DFS is an algorithm that explores a directed graph, starting from a given node v, until every node that is reachable from v has been visited (cf. Section 4.1 [40]).

The iterative algorithm in Algorithm 2.1.1 has the complexity $\mathcal{O}(|V| + |E|)$. Given a node v, DFS marks the node as visited and pushes the direct successors of v onto a stack. Succeeding, v will be set to the last element on the stack and the previous step will be repeated. If the stack is empty, all reachable nodes from the given node v have been visited. The result is a list of nodes sorted by the order in which they have been visited.

Algorithm 2.1.1: Depth-first search (DFS)

```
   Data: directed graph G = (V, E), start node v ∈ V
   Result: list of nodes T = [v_0, v_1, ..., v_n]
 1 S ← EmptyStack
 2 D ← ∅
 3 T ← EmptyList
 4 PUSH(S, v)
 5 while S ≠ EmptyStack do
 6 |   v ← POP(S)
 7 |   if v ∉ D then
   |   |   /* mark v as visited                                  */
 8 |   |   D ← D ∪ {v}
   |   |   /* add v to the ordered list of visited nodes         */
 9 |   |   APPEND(T, v)
   |   |   /* push the direct successors of v onto the stack     */
10 |   |   foreach s ∈ DSUCC(v) do
11 |   |   |   PUSH(S, s)
12 |   |   end
13 |   end
14 end
```

For the graph in Figure 2.1, DFS$(a) = [a, c, b, e, f, d]$ is one possible DFS traversal, starting at node a; DFS$(a) = [a, b, d, f, e, c]$ is also possible. The traversal DFS$(a) = [a, b, c, e, f, d]$ is impossible because after b has been visited, one of its direct successors, d or e, as well as all the nodes reachable from those have to be visited before c can be visited.

2.1.3 Control flow graphs

Since the graph-theoretic terms and their characteristics have been defined, they will be adapted and extended in the context of compiler construction [4] and static program analysis [46]. Working on a sequence of program statements, the construction of control flow graphs based on those statements facilitates analysis of control and data flow; therefore, it supports reasoning about the behaviour of the sequence of program statements. Thereby, an order of execution of a sequence of program statements will be informally defined as *control flow*.

Definition 2.1.7 (control flow). *The control flow of a sequence of program statements is the order in which the statements will be executed or evaluated.*

A sequence of program statements can be partitioned into *basic blocks*. A program statement is an entry of a basic block if that statement is at the beginning of a sequence of program statements, if it is a target of a jump statement or if it immediately follows a jump statement (cf. Section 8.4.1 [4]). Building on that, a *control flow graph* is a directed graph with basic blocks as nodes and two additional nodes, *Entry* and *Exit* [41].

Definition 2.1.8 (basic block, control flow graph (CFG)). *A basic block is a sequence of program statements that is entered at the first and exited at the last statement. A control flow graph or CFG is a directed graph where nodes represent basic blocks and edges represent transfer of control between basic blocks. Besides the basic blocks, a control flow graph has two additional nodes called Entry and Exit. There is an edge from Entry to any basic block where statements can enter the sequence of program statements; there is an edge from any basic block where statements can leave the sequence of program statements to Exit.*

Defining a control flow graph with an *Entry* and an *Exit* node is a simplification without loss of generality. If a sequence of program statements has several ending points, the associated basic blocks will have edges to the node called *Exit*; the same holds for several starting points. For instance, if only one certain starting point s is of interest, then the edges from the node *Entry* to the other basic blocks can be removed (excluded the edges to the basic block that is associated with s).

In the control flow graph illustrated in Figure 2.2, *Entry* has one outgoing edge to a and *Exit* one incoming edge from f. If the control flow graph represents a sequence of program statements defining a function, then a can be interpreted as the start and f as the end of the function. For the purposes of illustration, in this example, the basic blocks contain labels instead of program statements. This serves as a visualisation for the definitions in the next sections.

Since a control flow graph is a graph of basic blocks and a basic block is a sequence of program statements, a control flow graph represents the control flow of program statements. A basic block b with two direct successors requires a conditional statement

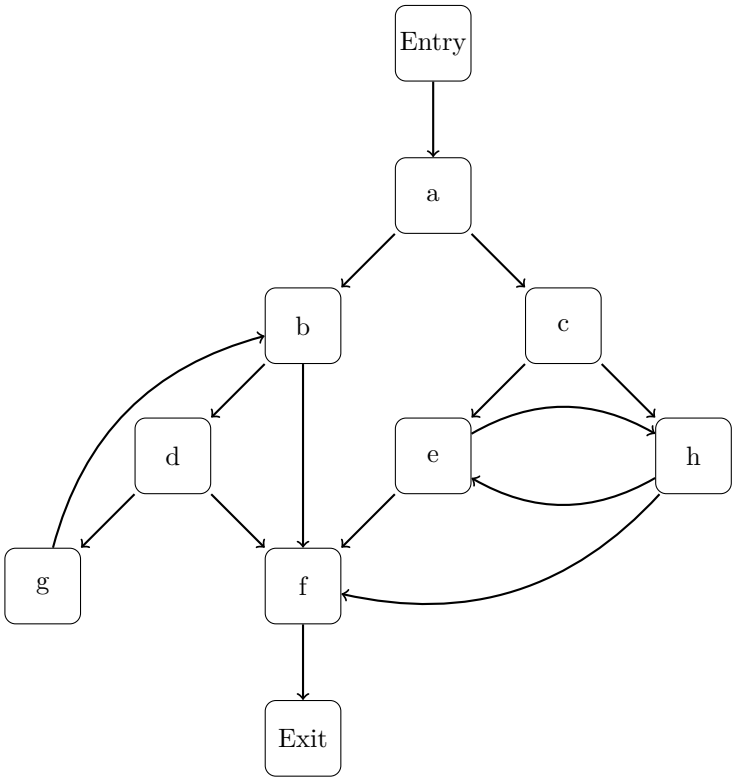

Figure 2.2: Control flow graph with one reducible and one irreducible loop

that defines whether the control flow will be directed to the one or the other direct successor of b. This condition will be called *jump condition*.

Definition 2.1.9 (jump condition). *Let $G = (V, E)$ be a control flow graph. If a basic block $b \in V$ has two direct successors $s_1, s_2 \in V$, then the last conditional program statement in b will be said to be the jump condition of b. Depending on the jump condition j, the conditional statement $j \,?\, s_1 : s_2$ directs the control flow to s_1, if j is true, otherwise to s_2.*

A jump condition can be either true or false. It may depend on complex statements; for instance, $x + y + z == x \cdot y$ as well as $x < y$ can be jump conditions. The last program statements in the basic blocks a, b, c, d, e and h in Figure 2.2 are conditional statements which rely on a jump condition. For example, let the jump condition for b be $x < y$. Then, the conditional statement could be $x < y \,?\, d : f$; if $x < y$ is true, then the control flow will be directed to d, otherwise to f.

Lastly, the concept of a walk or path can be adapted to control flow graphs. An *execution path* will be defined as a path that directs the control flow from *Entry* to a basic block b in a control flow graph. In contrast to Definition 2.1.2, an execution path will be defined as a walk in a control flow graph; an executed path may be

cyclic. Additionally, several execution paths may exist between two basic blocks. However, there exists only one execution path at a specific point in time, since an execution path represents an instance in which a sequence of program statements is evaluated.

Definition 2.1.10 (execution path). *Let $G = (V, E)$ be a control flow graph. An execution path p is a walk in the control flow graph that starts in Entry and directs the control flow in the sequence of program statements represented in the control flow graph along p. If there are different possible execution paths, only one of them can be taken at a specific point in time.*

In the illustrated control flow graph, $[Entry, a, c, e, f]$ and $[Entry, a, c, e, h, e, h, e, h, f]$ are two possible execution paths. It may be noted that there exists a cycle between e and h.

2.1.4 Dominance relations

In a control flow graph, the dominance relations between nodes are a strong property which enables techniques such as optimisation (cf. Section 10.4 [4]) or loop analysis (cf. Section 2.1.5). The dominance relations are also important for static single assignment, which will be introduced in Section 2.1.6. The definitions in this section are based on the work by Cytron et al. [41].

Definition 2.1.11 (domination, dominator, strict dominator). *Let G be a control flow graph. A node v_i dominates a node v_j, if every path from Entry to v_j passes through v_i. v_i is called the dominator of v_j. If a node v_i dominates a node v_j and $v_i \neq v_j$, then v_i strictly dominates v_j. The set of dominators of a node v will be denoted as* $\mathrm{DOM}(v)$*. If a node v_i dominates a node v_j, it will be denoted as $v_1 \,\mathrm{dom}\, v_2$; if a node v_i strictly dominates a node v_j, it will be denoted as $v_1 \,\mathrm{sdom}\, v_2$.*

For instance, in Figure 2.2, it is $\mathrm{DOM}(b) = \{Entry, a, b\}$, whereby b is dominated by itself, but strictly dominated by *Entry* and a. As well, f is dominated by itself and strictly dominated by *Entry* and a, $\mathrm{DOM}(f) = \{Entry, a, f\}$; *Exit* is dominated by itself and strictly dominated by *Entry*, a and f, $\mathrm{DOM}(Exit) = \{Entry, a, f, Exit\}$. Additionally, the dominators of h are *Entry*, a, c and h.

The closest strict dominator of a node v will be called the *immediate dominator*. Each node besides *Entry* has precisely one immediate dominator.

Definition 2.1.12 (immediate dominator). *Let G be a control flow graph. v_i is the immediate dominator of v_j if v_i is the nearest (direct) predecessor of v_j that strictly dominates v_j on all paths from Entry to v_j. The immediate dominator of a node v will be denoted as* $\mathrm{IDOM}(v)$*.*

To exemplify, it holds that $\mathrm{IDOM}(a) = Entry$, $\mathrm{IDOM}(f) = a$ and $\mathrm{IDOM}(h) = c$. The only node that does not have an immediate dominator is *Entry*, because *Entry* does not have any strict dominator.

The *dominator tree* of a control flow graph is a data structure that represents dominance relations in a graph. If there exists an edge from a node u to a node v in the dominator tree, then u is the immediate dominator of v [41].

Definition 2.1.13 (dominator tree). *Let $G = (V, E)$ be a control flow graph and $E' = \{(\text{IDOM}(v), v) | v \in V \backslash \{Entry\}\}$. Then $G' = (V, E')$ is called the dominator tree of G.*

As stated above, *Entry*, a and f strictly dominate *Exit*, whereby $\text{IDOM}(Exit) = f$, $\text{IDOM}(f) = a$ and $\text{IDOM}(a) = Entry$. Therefore, $[Entry, a, f, Exit]$ is a path in the dominator tree, as depicted in Figure 2.3. Throughout the course of the dissertation, the terms *predecessor* and *successor* will refer to control flow graphs; the terms *parent* or *ancestor* and *child* or *descendant* will refer to dominator trees.

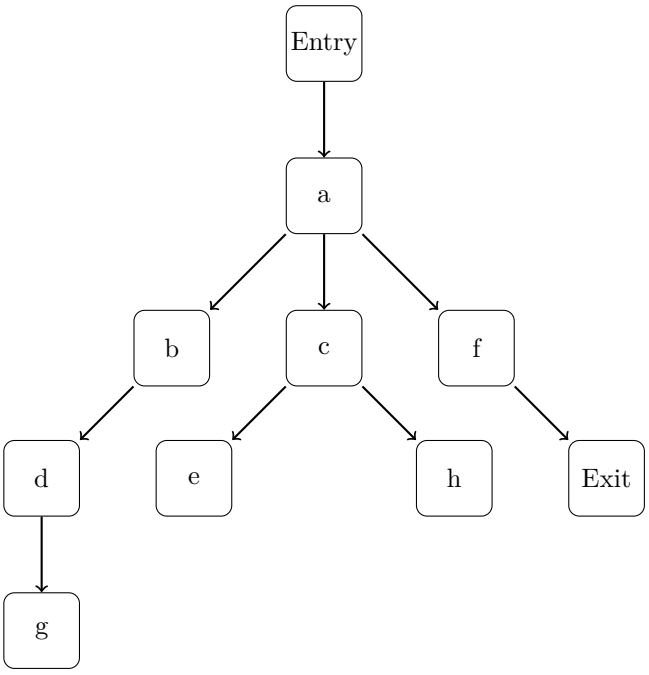

Figure 2.3: One dominator tree of the control flow graph in Figure 2.2

If a node v_j in a control flow graph G has more than one incoming edge, then there exists a *path convergence* of at least two paths in v_j. Let p_1 and p_2 be two paths, which converge first in v_j. Assume that p_1 is a path from *Entry* to *Exit*, p_2 is a path from a node v_i to *Exit* and $v_i \notin p_1$. Then, it will be said that v_j is in the *dominance frontier* of v_i.

Definition 2.1.14 (dominance frontier (DF)). *A node v_j in a control flow graph G is in the dominance frontier or DF of a node v_i if v_i dominates a predecessor of v_j and if v_i does not strictly dominate v_j. The dominance frontier*

of a node v_i will be denoted as $\mathrm{DF}(v_i)$*, whereby it will be formally defined as* $\mathrm{DF}(v_i) = \{v_j | \exists p \in \mathrm{PRED}(v_j) : v_i \,\mathrm{dom}\, p \wedge \neg(v_i \,\mathrm{sdom}\, v_j)\}$.

In Figure 2.2, c dominates the set $\{c, e, h\}$. There is a path $p_1 = [Entry, a, b, f, Exit]$, with $f \in p_1$ and $c \notin p_1$. There also exists path $p_2 = [Entry, a, c, e, f, Exit]$, with $c \in p_2$ and $f \in p_2$. f is also the first node, where p_1 and p_2 converge. In other words, c dominates e, a predecessor of f, but c does not strictly dominate f. Therefore, f is in the dominance frontier of c. Others examples are $\mathrm{DF}(d) = \{b, f\}$ and $\mathrm{DF}(g) = \{b\}$. Since a dominates every node besides *Entry*, there cannot be a path p from *Entry* to *Exit* with $a \notin p$. Thus, it is $\mathrm{DF}(a) = \emptyset$.

2.1.5 Reducible and irreducible loops

Since a control flow graph represents the control flow of program statements, informally, a cycle in a control flow graph represents a loop in a sequence of program statements. For that reason, within the framework of this dissertation, cycles in a control flow graph will be referred to as *loops*. The set of nodes in a loop will be called a *loop body*. If it is clear from the context, the terms loop and loop body will be used interchangeably. In relation to Definition 2.1.5, a cycle (loop) in a directed graph (control flow graph) is a strongly connected component because each node can be visited from any other node within the cycle (loop). In other words, a strongly connected component with at least two nodes is an *outermost loop* in the control flow graph [42].

Definition 2.1.15 (loop, loop body). *A cycle in a control flow graph will be called a loop. The set of nodes within a loop will be called a loop body.*

Definition 2.1.16 (outermost loop). *A maximal strongly connected component with at least two distinct nodes in a control flow graph G will be said to be an outermost loop.*

The term *outermost* does not reveal anything about the structure of the nodes within the strongly connected component. If l_2 is an outermost loop, there may exist a loop l_1 which is a proper subset of l_2. l_1 will be called an *inner loop*, l_2 an outer loop. In other words, loops can be *nested*.

Definition 2.1.17 (nested loops, inner loop, outer loop). *Let l_1, l_2 and l_3 be loops in a control flow graph G. If it holds that $l_1 \subset l_2 \subset l_3$, then l_1 will be called an inner loop of l_2 and l_3; l_2 will be said to be an inner loop of l_3, but an outer loop of l_1. The loops l_1, l_2 and l_3 are nested.*

Definition 2.1.18 (innermost loop). *A loop in a control flow graph that does not contain any further loops will be called the innermost loop.*

For instance, the control flow graph G in Figure 2.2 has two outermost loops – $l_1 = \{e, h\}$ and $l_2 = \{b, d, g\}$. Let there be an additional edge (g, d). Then, $l_3 = \{d, g\}$

is an *inner loop* and l_2 an *outer loop*. Additionally, since l_3 does not contain any further loop, it is the innermost loop.

The first node of a loop l that will be visited in a control flow graph G will be called a *loop header* or *loop entry*. Formally, a loop header is a loop node which has a direct predecessor outside of the loop. A loop may have several headers. If it has a single entry, the loop will be called *reducible*, otherwise it will be called *irreducible* [42].

Definition 2.1.19 (loop header). *Let l be a loop in a control flow graph $G = (V, E)$. If a node $v \in l$ has a direct predecessor p with $p \notin l$, then v is a header or entry of l. The set of entries of a loop l is defined as $H = \{v | \exists p \in \mathrm{DPRED}(v) : p \notin l \wedge v \in l\}$.*

Definition 2.1.20 (reducible loop, irreducible loop). *If a loop in a control flow graph has a single loop entry, it is a reducible loop; if a loop has multiple loop entries, then it is an irreducible loop.*

The two loops in Figure 2.2 are $l_1 = \{b, d, g\}$ and $l_2 = \{e, h\}$. The node $b \in l_1$ is the only node in l_1 that has a direct predecessor outside of l_1. Thus, l_1 has a single loop entry and is a reducible loop. In l_2, e and h have a direct predecessor outside of l_2. Therefore, l_2 has two loop headers and is an irreducible loop.

Since a reducible loop has a single entry point, the loop header dominates all nodes in the loop. Thus, there exists an edge from a node of the loop body to the dominator of the loop. Such an edge (v, u) from a node v to a dominator u of v is called a *back edge*. Due to the fact that $u \,\mathrm{dom}\, v$ implies the existence of a path from u to v, a back edge (v, u) implies the existence of a loop, which is dominated by u. The loop body consists of u and all the nodes which are able to reach u without walking through v. In other words, the loop body consists of all nodes that can be visited by performing a depth-first search from u to v on the reverse control flow graph. Such a loop will be defined as a *natural loop* (cf. Section 9.6.6 [4]).

Definition 2.1.21 (back edge). *Let $G = (V, E)$ be a control flow graph. If a node v_i dominates a node v_j, $v_i \,\mathrm{dom}\, v_j$, and there exists an edge $e = (v_j, v_i) \in E$, then e is said to be a back edge.*

Definition 2.1.22 (natural loop). *Let $G = (V, E)$ be a control flow graph and $e = (v_j, v_i) \in E$ a back edge. Then, v_j is be said to be the loop header of a natural loop, whose loop body consists of the set of nodes obtained from a depth-first search from v_j to v_i on the reverse control flow graph $G' = (V, E')$, with $E' = \{(e_2, e_1) | (e_1, e_2) \in E\}$.*

As stated above, $l_1 = \{b, d, g\}$ is a reducible loop with the loop entry b. Since there is an edge from g to b and b is a dominator of g, $e = (g, b)$ is a back edge. The edges (b, d), (d, g) and (g, b) are part of the loop. Hence, the loop edges of the reverse control flow graph are (d, b), (g, d) and (b, g). Performing a depth-first search from g to b on the reverse edges, $\mathrm{DFS}(g) = [g, b, d]$ is one possible result. Thus, the set of those nodes build the loop body of the natural loop dominated by b.

For the sake of completeness, some properties of natural loops will be mentioned. If two natural loops have the same header, one is either properly contained within the other or they can be combined to form one loop. If two natural loops have distinct headers, they are either disjoint or nested. If a control flow graph only has natural loops, it is called *reducible* (cf. Section 9.6.6 [4]).

Definition 2.1.23 (reducible control flow graph). *A control flow graph is reducible if all of the loops in that control flow graph are reducible loops. Otherwise, the control flow graph is irreducible.*

The properties of reducible loops cannot be conferred to irreducible loops. Since there is no single loop entry, the dominance relations in the loop are different. One strategy, called *node splitting*, applies graph transformations such that irreducible loops will be transformed into reducible loops. The disadvantage of that approach is its exponential running time [45].

Last, as stated in Section 2.1.1, in relation to Definition 2.1.5 and Definition 2.1.6, the requirement that an edge in a directed graph has to be a tuple of distinct nodes excludes self-loops, by definition. This means that an edge (v, v) cannot exist for a node v in a control flow graph. Assume, it would be possible and such an edge could exist. Additionally, assume that this self-loop would not be part of any other loop. Then, this loop would not be detected as an outermost loop because it would only be strongly connected to itself. However, a self-loop is a natural loop, since a node dominates itself by definition and therefore, a back edge exists. In other words, if a control flow graph contains a self-loop, it can be detected.

2.1.6 Static single assignment

Static single assignment (SSA) is a representation of program statements that has two properties. First, each variable assignment is unique in that each use of a variable is defined by exactly one assignment. Second, variable assignments on different control flow paths can be distinguished in converging paths with Φ-functions. The concepts described in this section are based on the work by Cytron et al. [41] and Alfred V. Aho et al. (cf. Section 6.2.4 [4]). To begin, the terms *def-use chain* or *DU chain*, *LHS* and *RHS* have to be defined.

Definition 2.1.24 (def-use (DU) chain, left-hand side (LHS), right-hand side (RHS)). *In a sequence of program statements, an assignment of a variable v with a variable w has the form $v = w$. It will be said that v will be assigned to the value of w; v will be defined and w will be used. The definition of the assignment takes place on the left-hand side or LHS, the use of the assignment on the right-hand side or RHS. In other words, a variable v will be defined on the LHS and used on the RHS. The definition of a variable v and its further uses will be called the definition-use, def-use or DU chain of v. A DU chain of a variable v ends with the reassignment of v.*

In Figure 2.4a, the variable x will be defined in the statements $x = a+7$ and $x = a+b$; it will be used in the statements $y = x \cdot a$, $y = x - b$ and $z = x + y$. Since x will be defined twice on the LHS, there exist two DU chains for the variable x.

$$\begin{aligned} x &= a + 7 \\ y &= x \cdot a \\ y &= x - b \\ x &= a + b \\ z &= x + y \end{aligned}$$

(a) Program statements

$$\begin{aligned} x_0 &= a + 7 \\ y_0 &= x_0 \cdot a \\ y_1 &= x_0 - b \\ x_1 &= a + b \\ z_0 &= x_1 + y_1 \end{aligned}$$

(b) Program statements in SSA form

Figure 2.4: Program statements and their SSA representation

After the transformation of a sequence of program statements into SSA form, each variable definition of a variable v on the LHS will be unique; each variable use of v on the RHS will be the last definition of v. For instance, Figure 2.4b shows the SSA transformation of the program statements in Figure 2.4a. For the two definitions of x, two new variables, x_0 and x_1, will be created. Similarly, the last definition of x – x_0 or x_1 – will be used instead of x on the RHS.

The advantage of the first property of SSA – each variable assignment is unique – is the creation of a link from the beginning to the end in the sequence of program statements. This enables, inter alia, optimisations as follows: After assignment, y_0 will never be used on the RHS; instead, y_1 will be assigned in the following statement. That means that y_0 has no influence on any statement in the sequence of program statements and can be removed. A second example of this advantage is the resolving of dependencies. For instance, the variable z_0 depends on x_1 and y_1. Further, y_1 depends on x_0 and b, whereby x_0 depends on a and a constant; x_1 depends on a and b. In other words, z_0 can be resolved to $z_0 = x_1 + y_1 = \cdots = a + b + a + 7 - b$, which can be simplified to $z_0 = 2 \cdot a + 7$.

Relying only on the first property of SSA, a control flow graph cannot be transformed into SSA form. As stated in Section 2.1.3, if a basic block b_1 has two outgoing edges, the control flow depends on a jump condition; the jump condition affects the control flow. The flow of control will be directed either to the one or the other direct successor of b_1. Two different execution paths of the program statements are possible - p_1, if the jump condition of b_1 is true, and p_2, if the jump condition is false. Then, let v be a variable which will be defined in b_1 and modified in at least one of the execution paths. If p_1 and p_2 converge in a basic block b_2 and v will be used in b_2, then v cannot be assigned explicitly. For that reason, the concept of Φ-*functions* will be introduced.

Definition 2.1.25 (Φ-function). *In a control flow graph in SSA form, a Φ-function for a variable v is a function that combines all last definitions of v in converging*

execution paths and returns the definition of v that corresponds to the current execution path. An assignment has the form $v_i = \Phi(v_0, v_1, \ldots, v_n)$, with $0 < n < i$.

In general, Φ-functions are an abstract concept that combines the previous definitions of a variable for further analysis; they are not specified in a concrete or standardised way. Below, it will be illustrated how that concept can be used.

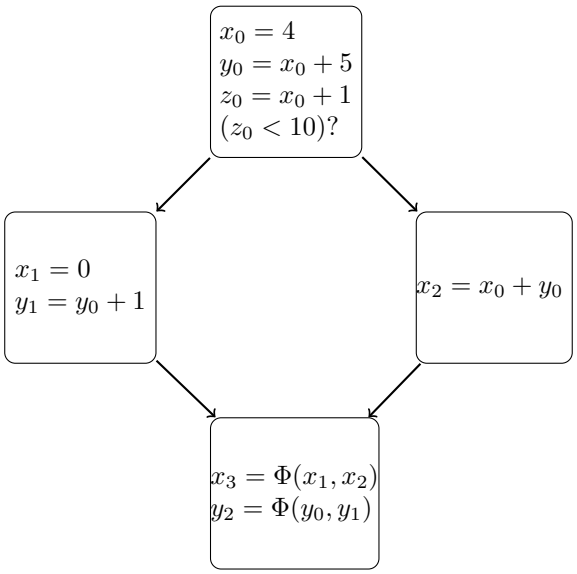

Figure 2.5: Φ-functions in a control flow graph

Figure 2.5 shows a control flow graph in SSA form. There are variable definitions for x, y and z. In the first basic block, the last definitions in SSA form are x_0, y_0 and z_0. There exists a jump condition $z_0 < 10$. The left successor redefines x and y; the right successor redefines x. Both execution paths converge in a block b. If b or a successor of b will use a variable x or y, it cannot be said which is the latest variable definition. For that reason, new variables – x_3 and y_2 – will be assigned for x and y in b; these will be used in the control flow starting in b. The variables are defined with Φ-functions; those functions signal that one of the two paths is the current execution path. For instance, if the right execution path will be taken, then x_3 will be assigned to the value of x_2 and y_2 to the value of y_0.

Another example will be given for an analysis based on SSA and the abstract concept of Φ-functions. For the control flow graph in Figure 2.5, it can be said which execution path will be taken. Let the control flow be directed to the left successor of the first basic block, if the jump condition $z_0 < 10$ is true; otherwise, it will be directed to the right successor. If $x_3 == x_2$ is true, then the right execution path will be taken. That implies that $z_0 < 10$ must be false. z_0 can be resolved to $z_0 = x_0 + 1 = 4 + 1 = 5$. In other words, $5 < 10$ cannot be true. Therefore, the left execution path is the only possible path; it holds $x_3 = x_1$ and $y_2 = y_1$.

2.2 Input crafting problem

The *input crafting problem* describes the task of finding an input such that certain statements in a sequence of program statements will be executed. In the literature, the problem has been phrased in different ways [47, 48]. Based on the definitions of the previous sections, the input crafting problem can be defined in a graph-theoretic manner as the search for a user input that directs the control flow from a basic block a to a basic block b in the control flow graph.

Definition 2.2.1 (input crafting problem). *Let $G = (V, E)$ be a control flow graph and $b_i, b_j \in V$ be two basic blocks. Additionally, let v be a variable which is defined by user input in b_i. The input crafting problem asks if there exists an input for v that directs the control flow from b_i to b_j in the control flow graph.*

2.3 Symbolic execution

Symbolic execution describes a technique which evaluates program paths in a symbolic manner, instead of one that is concrete. Working on a control flow graph, certain variables will be defined to be symbolic. Then, the program statements in the basic blocks will be calculated symbolically – equalities and inequalities will be generated in dependence on the symbolic variables and the semantics of the program statements. Those equalities and inequalities will be called *constraints* [14, 48].

Definition 2.3.1 (constraints). *For a sequence of program statements, constraints are equalities and inequalities derived from the semantics of the program statements.*

The jump condition of a basic block will also be evaluated symbolically. Let the last conditional statement of a basic block b, whose direct successors are s_1 and s_2 and whose jump condition is j, be $j \,?\, s_1 : s_2$. If the next basic block that will be executed symbolically is s_1, then the constraint will be defined such that j is true. Otherwise, j has to be false. In other words, the constraints have to be derived along an execution path. The constraints of an execution path are said to be the *path conditions*. To receive a concrete solution for the path conditions, the constraints have to be solved; if there exists a solution, the path conditions can be *satisfied*. Constraint solving will be discussed in Section 2.5.

Definition 2.3.2 (path conditions). *Let G be a control flow graph. For a sequence of program statements, the path conditions or path constraints are the constraints derived from an execution path in the control flow graph.*

Symbolic execution operates on execution paths. For instance, if that technique will be used for input crafting and there exist several possible execution paths, then the possible execution paths will be evaluated iteratively. For each evaluated path, it will be checked if the path conditions can be satisfied. If they can be satisfied, then

an input is known and the input crafting problem is decided. Since the number of possible execution paths grows exponentially with the number of branches, exploring every path is infeasible. Such exponential growth will be referred to as *path explosion*; the problem to decide which path should be explored first is known as *path selection problem* [49, 5].

Definition 2.3.3 (path explosion, path selection problem). *Let G be a control flow graph. The exponential growth of execution paths in the number of branches will be called path explosion. The path selection problem asks which direct successor of a basic block b should be visited first.*

2.4 Static data flow analysis

The term *data flow analysis* describes a set of techniques as acquire information about the flow of data in the control flow graph (cf. Section 9.2 [4]). The word *static* refers to the characteristic that the sequence of program statements will not be executed; contrary to that, *dynamic data flow analysis* operates on an execution path after a sequence of program statements has been executed [50].

Techniques as *reaching definitions*, *live variable analysis*, *dead variable analysis* or *available expressions* pertain to a set of a multitude of techniques in the context of data flow analysis (cf. Chapter 2 [51]). Those techniques are beyond the scope of this work and will not be covered. Instead, certain ideas of data flow analysis will be illustrated.

Data flow analysis on a control flow graph can be performed on the level of basic blocks. For each basic block, it is possible to define inputs for variables that enter the basic block and outputs for variables that leave the basic block (cf. Section 9.2.3 [4]). The inputs of a basic block b are the union of outputs of its direct predecessors. In contrast, the outputs of b are defined as the union of the variables that will be defined in b and of the inputs which have not been modified in b.

For instance, in Figure 2.5, let the basic block b_0 be the basic block which is the root of the graph, b_1 be the left direct successor of b_0, b_2 be the right direct successor and, lastly, b_3 be the remaining basic block. b_0 defines the variables x_0, y_0 and z_0. Therefore, $out_0 = \{x_0, y_0, z_0\}$ is the set of outputs. The inputs of b_1 and b_2 are equal to the outputs of b_0, $in_1 = in_2 = out_0$. The outputs of b_1 are defined as $out_1 = \{x_1, y_1, z_0\}$ because of the redefinitions of x and y. Equally, $out_2 = \{x_2, y_0, z_0\}$ are the outputs of b_2. Furthermore, the inputs of b_4 are the union of the incoming outputs – $in_4 = out_1 \cup out_2 = \{x_1, x_2, y_0, y_1, z_0\}$. At last, since x and y have been redefined, the outputs of b_4 are $out_4 = \{x_3, y_2, z_0\}$. The SSA representation already includes concepts of data flow analysis. For instance, the Φ-functions in b_4 are defined to represent the last variable definition of an execution path, as stated in Section 2.1.6.

2.5 Satisfiability modulo theories

In this section, the theoretical and practical foundations for constraint solving on the basis of SMT solvers will be acquired. For that, the basic principles of first-order logic have to be mentioned. On the other hand, handling these principles continuously in a formal manner is beyond the scope of this work and unnecessary. Therefore, the following sections will laid out as informally as possible, while still retaining the required level of formality.

First, basic terminology will be presented. Then, the functionality of SAT solvers will be described such that the power of SMT solvers can be demonstrated. Finally, two common theories will be described – the theory of bit vectors and the theory of arrays.

2.5.1 Preliminaries

The *first-order logic (FOL)* consists of a set of variables, logical symbols (for instance, $\wedge$, $\vee$, $\neg$, $\exists$ and $\forall$), nonlogical symbols (e.g., functions) and a syntax, which defines how formulas are constructed. The first-order logic without quantifiers ($\forall$ and $\exists$) and nonlogical symbols will be referred to as *propositional logic*. A *formula* is a well-formed sentence of first-order logic, with respect to the syntax. It is *satisfiable* if there exists an *interpretation* for which the formula is true; otherwise, it is unsatisfiable (cf. Section 1.4 [19]). In the context of this work, the word *model* will be used instead of 'interpretation'. Additionally, in the context of first-order logic, if a formula is true, it will be denoted as being equal to 1; if it is false, it will be denoted as being equal to 0.

Definition 2.5.1 (formula, satisfiable, unsatisfiable). *A well-formed sentence φ of first-order logic will be called a formula. If there exists a structure for which a formula φ is true, then φ is satisfiable; otherwise, it is unsatisfiable. The structure that is satisfiable for φ will be referred to as interpretation or model.*

$\varphi = (a \vee b) \wedge (a \vee \neg b)$ is a formula of first-order logic. $\varphi = 1$ holds for $a = 1$ and $b = 0$; φ is satisfiable and $m = [a = 1, b = 0]$ is a model that satisfies φ.

The simplest formulas of first-order logic are known as *atoms* or *predicates*. The term *literal* refers to a predicate or its negation. Predicates that are connected through logical symbols can be represented, amongst others, in *conjunctive normal form* and in *disjunctive normal form (DNF)*. (cf. Section 1.3 [19]).

Definition 2.5.2 (conjunctive normal form (CNF), disjunctive normal form (DNF)). *A formula of first-order logic is in conjunctive normal form or CNF, if it is a conjunction of disjunctions. If a formula is a disjunction of conjunctions, it is in disjunctive normal form or DNF.*

To exemplify, the formula $(a \vee b) \wedge (\neg a) \wedge (a \vee c)$ is in conjunctive normal form; $(a \wedge b) \vee (\neg a) \vee (a \wedge c)$ is in disjunctive normal form. In these formulas, a, b and c are the predicates; the literals of a are a and $\neg a$.

2.5.2 SAT solver

Before satisfiability modulo theories and SMT solvers can be described, the satisfiability problem and the function of SAT solvers will be presented. These are basic principles for the next section. The *satisfiability problem (SAT)* describes the problem if a formula is satisfiable.

Definition 2.5.3 (SAT problem). *The satisfiability problem SAT asks if a formula φ of the propositional logic is satisfiable or unsatisfiable.*

In general, SAT is an NP-complete problem (cf. Section 2.5 [19]). Nonetheless, *SAT solvers* are known to perform adequately on many real-world problems.

Definition 2.5.4 (SAT solver). *A SAT solver is a program that decides if a formula φ of propositional logic is satisfiable or unsatisfiable. If it is satisfiable, it returns* `SAT` *and a model; otherwise, it returns* `UNSAT`*.*

'Modern SAT solvers can solve many real-life CNF formulas with hundreds of thousands or even millions of variables in a reasonable amount of time' (Chap. 2, p. 27 [19]). The reason for this is the way SAT solvers work. SAT solvers based on the *Davis-Putnam-Logemann-Loveland (DPLL) algorithm* are able to learn when they run into a conflict. A *conflict* is an assignment of the variables in the formula φ, such that φ becomes false. Then, a SAT solver creates a *conflict clause* c and tries to solve the new formula $\varphi' = \varphi \wedge c$. In a formula in conjunctive normal form, all disjunctions have to be true. A SAT solver randomly picks a variable in a disjunction and sets it to true or false. Next, the other variables in this disjunction will be set such that the disjunction becomes true. A conflict occurs if a disjunction cannot be true with the current variable assignments. While performing an exhaustive search on binary trees, the ability to learn conflict clauses facilitates a SAT solver to prune the search space significantly. In other words, SAT solvers are efficient for use in many real-world problems because they are able to exploit the structure of those problems (cf. Section 2.2 [19]).

2.5.3 SMT solver

Informally, *satisfiability modulo theories (SMT)* can be described as an extension of SAT. An SMT formula φ is an instance of SAT in which the predicates of φ are not limited to propositional predicates; instead, they can be predicates over (combined) theories, such as the theory of integers (cf. Section 3.3.3 [52]), the theory of reals (cf. Section 3.4.1 [52]), the theory of arrays (cf. Section 3.6 [52]) and the

theory of bit vectors (cf. Section 6.1 [19]). In contrast to SAT formulas, real-world problems can be modelled more expressive in SMT formulas (cf. Section 1 [53]). The *satisfiability modulo theory problem* describes the problem if an SMT formula is satisfiable (cf. Section 3 [53]).

Definition 2.5.5 (satisfiability modulo theory problem). *Let φ be a formula of propositional logic and of a theory T. The satisfiability modulo theory problem for the theory T asks if there exists a model of T that also satisfies φ.*

As stated in Section 2.5.2, SAT is NP-complete. Satisfiability modulo theories are, in the worst case scenario, undecidable. *Quantifier-free* fragments of theories – the fragments of theories without $\forall$ and $\exists$ – are NP-complete, in many cases. For instance, this holds for the theory of bit vectors and the theory of arrays. The quantifier-free fragments are sufficient for modelling many real-world problems [10]. As well as SAT solvers, *SMT solvers* are known to perform adequately on many real-world problems.

Definition 2.5.6 (SMT solver). *An SMT solver for a theory T is a program that decides if a formula φ of propositional logic and of the theory T is satisfiable or unsatisfiable. It returns* `SAT` *and a model, if it is satisfiable; otherwise it returns* `UNSAT`*.*

SMT solvers depend on SAT solvers. There are two common techniques how an SMT solver interacts with a SAT solver – eager and lazy SMT techniques. The *eager* technique translates the fragments of SMT formulas in non-propositional logic into fragments in propositional logic and solves the transformed formula with a SAT solver. In the *lazy* approach, a SAT solver interacts with a theory solver for a theory T. If the SAT solver generates a model, it queries the theory solver whether this model is also a model for T. If this is not the case, the SAT solver creates – as described in Section 2.5.2 – a conflict clause, generates a new formula and re-starts. Otherwise, the SAT solver generates a model that is also satisfiable in T. Then, the SMT solver returns `SAT` and a model m. In general, modern SMT solvers are build upon the lazy approach (cf. Section 3.2 [53]).

Many modern SMT solvers support the *SMT-LIB* standard [54], which defines a common interface for SMT formulas. These formulas can be solved with different SMT solvers. In the following sections, the basic ideas of the theory of bit vectors and of the theory of arrays will be illustrated informally. In the SMT-LIB standard, the theory of quantifier-free bit vectors is known as QF_BV; the combined theory of quantifier-free formulas over the theory of bit vectors and arrays is known as QF_ABV [55].

2.5.4 Theory of bit vectors

In this section, the basic concept of the theory of bit vectors and their operations will be given. The formulas in the theory of bit vectors will be assumed to be

quantifier-free and be represented in *QF_BV* [56]. A *fixed-sized bit vector* is an array of bits of the length l.

Definition 2.5.7 (bit vector). *A bit vector b of the length l is a tuple $b = (b_0, \ldots, b_{l-1})$, with $b_i \in \{0, 1\}$.*

Arithmetic and bitwise operations can be formally defined in the theory of bit vectors (cf. Section 6.1 [19], Section 1.3 [57] and Section 2.4 [58]). Therefore, arithmetic operations, such as addition, subtraction, multiplication or division as well as bitwise operations as &, |, ¬, ⊕, << and >>, can be applied to bit vectors. Since bit vectors have a fixed-size, they may overflow. To exemplify, bit vectors of the length $l = 8$ represent the numbers from 0 to 255.

2.5.5 Theory of arrays

The *theory of arrays* models the behaviour of arrays used in programming languages (cf. Section 7.1 [19]). Since arrays fulfil certain characteristics, these have to be modelled properly. To point out those properties, the theory of arrays will be formally introduced. The following definitions are based on the work by Bradley and Manna (cf. Section 3.6 [52]) as well as by Falke, Sinz and Merz [59].

Definition 2.5.8 (theory of arrays). *The theory of arrays has the signature $\Sigma_E : \{\cdot[\cdot], \cdot(\cdot \triangleleft \cdot), =\}$. $a[i]$ or $\text{read}(a, i)$ represent the read operation and $a(i \triangleleft v)$ or $\text{write}(a, i, v)$ the write operation; the operator for equality is represented by $=$.*

The read and write operations are defined by

- *read:* $\text{ARRAY} \times \text{INDEX} \to \text{ELEMENT}$ *and*
- *write:* $\text{ARRAY} \times \text{INDEX} \times \text{ELEMENT} \to \text{ARRAY}$.

The axioms of the theory of arrays are

1. *the axioms of equality (symmetry, transitivity and reflexivity),*
2. $\forall a, i, j : i = j \to \text{read}(a, i) = \text{read}(a, j)$ *(array congruence),*
3. $\forall a, v, i, j : i = j \to \text{read}(\text{write}(a, i, v), j) = v$ *(read-over-write 1) and*
4. $\forall a, v, i, j : \text{read}(\text{write}(a, i, v), j) = \text{read}(a, j)$ *(read-over-write 2).*

The theory of arrays consists of the equality operator $=$ and two functions – *read* and *write*. The equality operator, in combination with the axioms of equality, state that a is equal to c if a is equal to b and if b is equal to c, for the arrays a, b and c.

An array a and an index i are the inputs of the read operation; the output is a value v. It will be said that the value v will be read from an array a at an index i – denoted as $v = \text{read}(a, i)$ or $v = a[i]$.

On the contrary, an array a, an index i and a value v are the inputs of the write operation; the output will be a new array a', which encodes logically the previous write operation. It will be said that the value v will be written into an array a at an index i – denoted as $a' = \text{write}(a, i, v)$. The theory of arrays works in a functional manner (cf. Section 3.6 [52]); for this reason, a new array will be returned that encodes the previous write access.

For instance, let a be an array, i and j indices and, last, v and w values. Additionally, let $i \neq j$ and $v \neq w$. Then, two write operations into a can be represented as $a' = \text{write}(a, i, v)$ and $a'' = \text{write}(a', i, w) = \text{write}(\text{write}(a, i, v), i, w)$. In other words, it holds that $v == a''[i]$ and $w == a''[j]$ are true.

These properties are modelled by the axioms. The axiom called *array congruence* defines that two read operations are equal if the same index will be read; $i = j$ implies that $a[i] == a[j]$ is true. The third axiom, *read-over-write 1*, states that if two indices are equal ($i = j$), then the value that will be read at an index j in the array a is identical to the preceding write of the value v into the array a at an index i. To illustrate, presuming the write operation $a' = write(a, j, v)$, the formula $v == a'[i]$ will be true if $i = j$. In contrast, if $i \neq j$, then $v == a'[i]$ will be false, but $v == a'[j]$ be true, since the array indices are different. This characteristic is described by the fourth axiom, *read-over-write 2*.

2.6 Bounded model checking

Bounded model checking (BMC) is a technique used in the context of hardware and software verification. The approach of SMT-based bounded model checking is to prove a property p of a transition system M, which is unrolled up to a specified bound k. Then, an SMT solver will decide a formula consisting of the negation of the property p and the k times unrolled transition system M. If the formula is satisfiable, the property is proven to be wrong [13]. In the following, the application of BMC will be limited to software; the definitions and concepts will be based on the work by Sinz, Falke and Merz [38].

The main idea of BMC is to create a transition system of a sequence of program statements. This transition system will be unrolled k times and translated into an SMT formula. Unwinding this system with a bound k connotes that loops in the sequence of program statements will be unrolled k times. As a result, the unrolled transition system of the sequence of program statements can be represented as a directed acyclic graph. Checking properties for an unbounded transition system of a program is undecidable because of its Turing completeness. On the other hand, checking properties for a bounded transition system is decidable, since such a system is a finite state machine; properties will be checked for finite program runs.

For an SMT formula *prog*, which represents an unrolled transition system of a sequence of program statements, properties of the program statements will be applied

as preconditions and postconditions. *Preconditions* describe properties that will be *assumed* to be fulfilled; *postconditions* describe properties that must be fulfilled.

Definition 2.6.1 (precondition, postcondition). *Let prog be an SMT formula of an unrolled transition system of a sequence of program statements. An SMT formula precondition encodes an assumption that must hold for prog; an SMT formula postcondition encodes the property or assertion that must hold.*

Let *prog* be an SMT formula of a sequence of program statements. In addition, let *preconditions* be an SMT formula of a conjunction of assumptions that are assumed to hold for certain program statements. Last, let *postconditions* be a conjunction of properties that have to be proven. Then, the formula $\varphi = preconditions \wedge prog \wedge \neg postconditions$ will be decided by an SMT solver. If φ is unsatisfiable, then it is proven that the properties hold in the k times unrolled transition system, which is represented by *prog* as well as by the corresponding preconditions; in other words, the properties have been proven for a limited amount of runs of the sequence of program statements. If φ is satisfiable, then the returned model is a counterexample, which proves that the asserted properties do not hold.

2.7 Miasm's intermediate representation

Miasm [60] is an architecture-independent binary reverse engineering framework, which is licenced under GPLv2 [61] and written in Python. It supports, amongst others, the architectures *x86* [62], *x86_64* [62], *ARMv7* [63], *AArch64* [64] and *MIPS32* [65]; for each architecture, it uses 'its own disassembler [...] and instruction semantic[s]' [60]. Additionally, Miasm facilitates the analysis of directed graphs, symbolic execution and code emulation, which is based on just-in-time compilation build upon *TinyCC* [66]. The symbolic execution operates on the level of basic blocks; in relation to the code emulation, functions of the *application programming interface (API)* can be emulated. In particular, Miasm uses its own *intermediate language (IL)* or *intermediate representation (IR)* called *Miasm IR*, which includes a translation into SMT formulas for the SMT solver *Z3* [20]. In the following, the Miasm IR will be introduced. If not stated otherwise, the description of Miasm IR will be based on the work by Fabrice Desclaux [67] and the related source code [68].

Miasm IR is an architecture-independent intermediate representation and designed for binary program analysis and vulnerability research. It is explicit and based on modular expressions. There exist eight categories of expressions that represent integers, variables, assignments, conditional statements, memory access, operations, extraction of bits and composition of expressions. Expressions from different categories may be combined.

Definition 2.7.1 (Miasm IR). *The intermediate representation of Miasm, Miasm IR, consists of modular expressions. An expression may be part of another expression; expressions may be nested. There exist eight categories of expressions. These are*

1. *ExprInt, used to represent constants,*
2. *ExprId, used to represent variables,*
3. *ExprAff, used for assignments,*
4. *ExprCond, used for conditional statements,*
5. *ExprMem, used for memory access,*
6. *ExprOp, used for applying operators on expressions,*
7. *ExprSlice, used to extract bits from an expression and*
8. *ExprCompose, used to compose or concat expressions.*

The size of an expression is included in its definition. If two expressions, e_1 and e_2, will be combined, then the combined expression is conforming to the sizes of e_1 and e_2.

As a result of the compact but modular design, constants, variables and memory expressions can be distinguished. Only assignments modify other expressions. If a memory expression is on the left-hand side of an assignment, it is a memory write; if it is on the right-hand side of an assignment, it is a memory read. Conditional statements are of the form $cond\,?\,a : b$ – if *cond* is true, then a will be returned, otherwise b. In operations, operands must have the same size; one operator and at least one operand will be expected. Operations can be unary operations, such as $\neg a$ and $-a$, or binary operations, such as $a + b$ and $a \cdot b$, whereby a and b are expressions. It is also possible to define operations for a variable number of arguments. Then, it will be represented as a function, for instance $\gcd(a_1, \ldots, a_n)$, whereby $a_1, \ldots, a_n$ are expressions and gcd() is the operator. Last, expressions may be sliced and composed. For instance, a register *EAX* (represented as variable) can be expressed as the slice of the first 32 bits of the register *RAX*, which has a size of 64 bit. Contrary to that, two variables with a size of 32 bit can be composed to a variable with a size of 64 bit.

While assembly code is implicit, the intermediate representation is explicit – every assignment has exactly one effect. Each assembly instruction will be transformed into a list of expressions. For instance, `POP RBP` will be transformed into `[RSP = (RSP+0x8), RBP = @64[RSP]]`; `RSP` will be incremented by $0x8$ and `RBP` will be assigned to the memory value at the address that is stored in `RSP`. In Miasm IR, the list of expressions that are related to one assembly instruction will be evaluated simultaneously; the intermediate representation of an assembly instruction describes the input and output state of that instruction. The right-hand side of the assignments in the intermediate representation describes the input state, the left-hand side the output state. To put it differently, the order of the intermediate expressions is insignificant as long as they belong to the assembly instruction that is evaluated currently [69].

Lastly, a few characteristics of Miasm IR will be mentioned. Miasm IR's instruction pointer, `IRDst`, is always the last instruction of a basic block in the intermediate representation. A basic block in the assembly code may be translated into several basic blocks in the intermediate representation. Each function call is represented as a separate basic block in the intermediate representation [69, 70].

3 Bounded model checking for input crafting

Building on the preceding established theoretic foundations, this chapter describes the process as the input crafting problem can be addressed with bounded model checking (BMC). First, the general idea of using bounded model checking for input crafting will be introduced. After that, its individual steps will be outlined. The prerequisites, function discovery and API call handling, as well as the process of unrolling the transition system and rewriting the intermediate representation, will be discussed. Then, the most important step will be exposed in the context of SSA – encoding the control flow into a logical formula. Last, the memory model will be explained and the building of an SMT formula will be illustrated.

In contrast to the previous chapter, definitions in this chapter will be informal; they will be used to summarise concepts, instead of providing formal frameworks. The same holds for algorithms, which will be mentioned where necessary, but not discussed in detail.

3.1 Introduction

In this section, bounded model checking for binary input crafting will be introduced. First, the general idea will be derived. Second, the process of binary BMC will be exposed. Thereafter, a concrete framework that executes the described process will be presented. Last, the underlying assumptions for this task will be discussed.

3.1.1 General idea of BMC-based input crafting

As described in Section 2.3, symbolic execution is a common technique for input crafting. To locate a user input which directs the control flow to a specific basic block in the control flow graph, constraints will be derived by evaluating symbolically possible execution paths. If the constraints for a path can be satisfied, then the satisfying model represents an input with the required characteristic. One main limitation is the exponential growth of execution paths in the number of branches, known as the path explosion. On these grounds, the path selection problem leads to the question of which path should be evaluated first (cf. Definition 2.3.3).

To address path explosion and alleviate path selection, in relation to input crafting, the number of paths can be reduced by static data flow analysis. Only paths that are known to be influenced by user input have to be evaluated. In the worst case, static data flow analysis has to be performed on the entire control flow graph to acquire

this knowledge. In this case, the user input is defined in the first basic block of a function.

As stated in Section 2.5.3, SMT solvers are strong in deductive reasoning, especially when applied to real-world problems. On that account, it is feasible that SMT solvers are strong in performing data flow analysis; the question that arises is whether or not static data flow analysis can be encoded as a logical problem that will be decided by an SMT solver. Extending this question, it can be asked if it is possible to find a solution to the path selection problem with an SMT solver. Furthermore, the generalised question asks if it is operable to transform the entire control flow graph into a formula of first-order logic and decide the input crafting problem by solely relying on an SMT solver, which tracks the dependencies of the user input, chooses a path and returns the chosen path and a satisfying user input, if the problem is satisfiable.

This question can be formulated as an instance of bounded model checking. With respect to Section 2.6, let the control flow graph be the transition system of a sequence of program statements. Then, the input crafting problem can be described as the property that a defined basic block must be visited. To put it differently, an SMT solver that decides a formula consisting of the conjunction of the unrolled control flow graph and the mentioned property, which is encoded as postcondition, decides the input crafting problem.

3.1.2 Process of BMC-based input crafting

The idea to perform bounded model checking on the binary level is inspired by work by Sinz, Falke and Merz [38], who conduct bounded model checking based on the intermediate representation of *LLVM (Low Level Virtual Machine)* [18] for the purpose of discovering memory errors in programs written in C. Subsequently, the process of *bounded model checking for binary input crafting* will be outlined; the essential steps will be described explicitly in the succeeding sections.

Definition 3.1.1 (bounded model checking for binary input crafting). *Let a sequence of program statements be assembly code. Additionally, let all calls to functions, the assembly code and the control flow graph of those functions be known. Lastly, let both, the basic block b_1, which defines user input, and b_2, the basic block to which the control flow must be directed by user input, be known. Then, bounded model checking for binary input crafting consists of the following steps.*

1. *transformation from assembly into an intermediate representation*
2. *directed acyclic graph generation*
3. *SSA transformation*
4. *translation into an SMT formula prog*

5. *defining preconditions and postconditions*
6. *SMT solving of* $\varphi = preconditions \wedge prog \wedge postconditions$

The preconditions are optional and define assumptions that have to hold. The postconditions describe the property that b_2 *must be visited. A model for* φ *describes an execution path* p *from* b_1 *to* b_2 *and all constraints that define the control flow on the path* p*.*

As described in the previous section, bounded model checking operates on a control flow graph, in the context of this work. Therefore, the program statements – the assembly instructions – have to be derived by disassembling the relevant parts of the binary program. Further, a function discovery has to be performed. Since the control flow graph will be unrolled, function starts, ends and calls have to be known. The same holds for calls to API functions because those functions could influence the control flow and have to be considered and handled. For reasons described in Section 2.7, the derived assembly instructions will be translated into an intermediate representation. On this basis, the control flow graphs can be constructed for the intermediate representation. Therefore, it will be assumed that a control flow graph exists for each discovered function. In addition, it will be assumed that it is possible to associate a basic block in the intermediate representation with the corresponding basic block in the assembly code and vice versa.

Hence, after the control flow graphs of the functions are derived, the transition system will be unrolled k times. For that, loops will be unwinded k times and functions will be inlined. As a result, the unrolled transition system will be represented by a directed acyclic graph. Subsequently, the intermediate representation has to be rewritten, such that the program semantics will be preserved. The rewritten sequence of program statements represents all possible execution paths for the unrolled control flow graph. In other words, this sequence represents all possible paths of a finite program run.

In the next step, the directed acyclic graph will be prepared so that it can be translated into an SMT formula. For that, SSA will be applied to the graph. As a result, each assignment will be unique. Lastly, the control flow must be encoded into a logical formula. At this stage, the directed acyclic graph will be represented by a set of constraints in the intermediate representation. These constraints, as well as the encoding of the control flow, will be translated into SMT formulas and combined to one large formula in conjunctive normal form; the directed acyclic graph will be represented by an SMT formula *prog*.

Then, the input crafting problem will be encoded as the condition that a certain basic block must be visited. This will be handled by a variable *visit*, which is true when the block has been visited; otherwise, it is false. In relation to Section 2.6, an SMT formula is of the form $\varphi = preconditions \wedge prog \wedge \neg postconditions$. Therefore, the input crafting problem has to be defined as $postconditions = (visit == 0)$ in order

to receive a satisfying model for the input crafting problem. For the sake of simplicity, it will instead be encoded as $\varphi = preconditions \wedge prog \wedge postconditions$, whereby the postcondition will be defined as $postconditions = (visit == 1)$. Additionally, the variable *preconditions* may be set to model assumptions; to exemplify, it may be set to model restrictions on the user input.

Finally, the SMT solver decides φ. If it returns `UNSAT`, then there exists satisfying input for the chosen bound; the bound can be increased and the procedure repeated. Otherwise, a satisfying user input will be returned.

3.1.3 Framework

Cylyx is a cross-platform framework for bounded model checking on the binary level that has been designed and realised as essential part of this work. It is written in Python and implements the concepts described in the following sections. Before these concepts will be elucidated, some characteristics of the implementation will be stressed.

The emphasis of Cylyx is to realise the core concepts of bounded model checking. Thus, in relation to the previous section, the focuses are unwinding the transition system, generating the SMT formula and defining postconditions. In contrast, prerequisites, such as function discovery, are implemented fragmentarily since these tasks can be performed by more powerful tools as *IDA* [71]. Those tools are not utilised, by default, in order to minimise required dependencies. Therefore, Cylyx supports these prerequisites, but also allows the integration of other frameworks.

Based on Miasm, Cylyx uses Miasm's disassembler, graph analysis and, especially, Miasm's intermediate representation, Miasm IR, as well as its translation into SMT formulas (cf. Section 2.7). Building on this, Cylyx performs graph transformations, such as function inlining and loop unrolling, rewrites the instructions to represent the program semantics of the directed acyclic graph, transforms this graph into SSA and translates it into an SMT formula. Cylyx is architecture-independent due to its implementation of the core concepts of bounded model checking, as well as Miasm's ability to translate assembly code of different architectures into Miasm IR. In the context of this work, parts of Cylyx have been submitted to and are now included in Miasm. All contributions are listed in Appendix A. The submissions include graph algorithms, bug fixes and extensions of the translation into SMT formulas.

3.1.4 Assumptions

To craft inputs for assembly code, some prior knowledge of the assembly code will be assumed. This knowledge includes the basic block in which user input will be defined as well as the basic block to which the defined user input must direct the control flow. For instance, if a function f has three input parameters a, b and c, if this function will return 1 or 0 and if the input for a and b should be crafted such

that f will return 1, then knowledge of the assembly code and control flow graph of f will be presumed. In addition, it will be inferred to be known which basic block will return 1 and that a and b are defined by user input. If the parameter c also influences the control flow, it must be known as c is defined outside of f. This will be encoded as a precondition.

3.2 Prerequisites

This section describes the prerequisites needed to perform bounded model checking for binary programs and methods to cope with those prerequisites. First, the subject function discovery will be addressed; second, the detection and handling of API functions will be explained.

3.2.1 Function discovery

As described in Section 3.1.2, bounded model checking on the binary level requires a precise description of the program semantics; it requires every assembly instruction of the relevant parts in the binary. For this, in the case of function calls, functions that will be called have to be inlined (cf. Section 3.3.1). This necessitates knowledge of function calls, as well as knowledge of the control flow graphs and the assembly code of called functions. The process of detecting functions will be called *function discovery.*

Definition 3.2.1 (function discovery). *Let a sequence of program statements be disassembled assembly code. Then, function discovery describes the task of grouping assembly instructions such that they represent functions of high-level programming languages.*

It is well-known that distinguishing between code and data is undecidable. Therefore, in general, disassembling is undecidable (cf. Section 1.2.1 [72]). This implies that function discovery itself is undecidable, since it relies on disassembled code. In practice, functions can be detected with heuristics. For instance, IDA and *BYTEWEIGHT* [73] are known to implement powerful heuristics for function discovery.

Function discovery performed by Cylyx relies on two assumptions; a function call jumps to the beginning of a function and there exists a return statement at the end of a function. The disassembled code between the beginning and the end of the function describe the semantics of the function. This basic approach works for unoptimised code without indirect function calls. For more powerful function detecting, IDA or *radare2* [74] can be used; the results have to be prepared such that they can be used by Cylyx. In other words, Cylyx's function discovery can be replaced by a more powerful one, if some manual work will be done.

3.2.2 API functions

Concerning the previous section, to describe precisely the program semantics, API functions must also be considered. Since API functions are often located outside of the binary and depend on the architecture, they cannot be included straightforwardly as functions within the same binary. Therefore, they have to be handled somehow. In the following sections, the detection of API calls will be described. Then, an interface for handling API functions will be characterised.

An *API call* is a function call that calls an API function. Often, API calls are calls to thunk functions, which jump into API functions. *Thunk functions* are functions that 'contain nothing but a single jump statement' (Chap. 20, p. 429 [75]).

Cylyx performs API call detection as follows. Firstly, it parses the imports of the binary. Then, it locates the thunk functions. (In the case of *ELF (Executable and Linking Format)* [76] files, it parses the binary. For *PE (Portable Executable)* [77] files, it relies on an IDA script which finds thunk functions. Afterwards, these functions have to be defined manually for Cylyx.) Lastly, a call to a thunk function which jumps into an API function will be detected as an API call.

As described above, Cylyx provides an interface for API call/function handling. For each API function, the handling has to be defined manually. Cylyx supports removing API calls from the control flow graph, inlining API functions and modelling API functions in an SMT formula. These will be discussed in the next sections; the third approach is not implemented, but considered as future work.

API functions that do not affect the control flow can be removed from the control flow graph. For instance, the functions `printf` or `puts` do not modify the control flow as long as their return value will not be used after the API call. In such a case, the program semantics do not depend on those functions. Since Miasm IR represents a function call as a separate basic block (cf. Section 2.7), this call block can be removed from the control flow graph.

The second approach, the inlining of API functions, is based on the idea that, if an API function can be represented as a single control flow graph without calls to other functions, then it can be treated as a function within the binary and be inlined (cf. Section 3.3.1). This method requires manual preparation, which has to be done for every API function of interest and for each architecture. Firstly, an API function and all functions that will be called within this function need to be disassembled. Secondly, these functions must be inlined such that the API function will be represented as one large control flow graph. Disassembling API functions connotes operating on optimised code, which may be multithreaded or may contain indirect function calls. Therefore, it cannot be applied to a large scale. However, if this has been done for a set of API functions, then those API functions can be inlined without further preparations. Cylyx includes scripts for facilitating this process, such as an IDA script for function discovery or a script to prepare API functions for inlining.

The last option for API call handling is not implemented. However, this approach is based on the idea to model the behaviour of an API function as an SMT formula, instead of building it on the basis of assembly code. This might enable an architecture-independent handling and may be more efficient than the second approach. For instance, Cylyx models low-level memory management in SMT formulas (cf. Section 3.6); Sinz, Falke and Merz [38] describe an approach for modelling `malloc` and `free`. Lastly, Vinod Ganapathy et al. [78] demonstrate an abstract framework for modelling API functions and apply it to `printf`.

3.3 Graph transformations

Until now, assembly code has been derived, function discovery has been performed and the handling of API functions has been defined. Thus, for every function, there exists a control flow graph that has been constructed from assembly code. In the next step, the assembly code will be translated into the intermediate representation of Miasm, Miasm IR. Henceforth, operations will be performed either on control flow graphs which have been constructed from the intermediate representation or on the intermediate representation itself.

This section describes the process of unwinding the transition system. Since a control flow graph represents the transition system, unrolling will be achieved by graph transformations. In the following, methods for function inlining, unrolling reducible loops and unrolling irreducible loops will be demonstrated. Then, a process which combines these methods will be described. As a result, the transition system will be a directed acyclic graph.

3.3.1 Function inlining

Function inlining describes the process of replacing function calls with the body of the called function. Since a function call will be represented as a separate basic block in Miasm IR (cf. Section 2.7), function inlining can be defined, informally, as replacing a function call block in a control flow graph with the control flow graph of the called function.

Definition 3.3.1 (function inlining, caller, callee). *Let $G = (V, E)$ be a control flow graph of a function f_1 and let a basic block $b \in V$ represent a function call to a function f_2. Additionally, let the control flow graph of f_2 be G'. If b will be replaced by the control flow graph of f_2 in G, then it will be said that f_2 will be inlined into f_1. E will be extended with edges from the direct predecessors of b to the heads of G' and with edges from the leaves of G' to the direct successors of b. f_1 will be referred to as the caller and f_2 will be referred to as the callee.*

This will be illustrated in Figure 3.1. In the control flow graph $G = (V, E)$ with $V = \{a, b, c\}$ and $E = \{(a, b), (b, c)\}$ of a function f_1, the basic block b calls a function

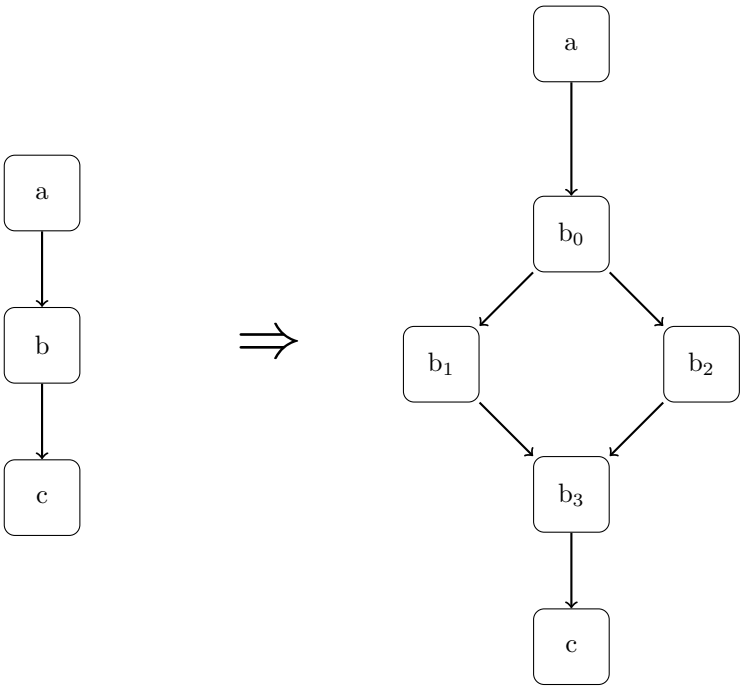

Figure 3.1: Replacing a function call with the callee's control flow graph

f_2; f_1 is the *caller* and f_2 the *callee*. The control flow graph of f_2 is $G' = (V', E')$ with $V' = \{b_0, b_1, b_2, b_3\}$ and $E' = \{(b_0, b_1), (b_0, b_2), (b_1, b_3), (b_2, b_3)\}$. b_0 is a head of G' and b_3 a leaf. To inline f_2 into f_1, the direct predecessors of b have to be connected with the heads of f_2 and the leaves of f_2 have to be connected with the direct successors of b. Therefore, the edges $e_1 = (a, b_0)$ and $e_2 = (b_3, c)$ will be added to E.

3.3.2 Unrolling reducible loops

Unrolling a loop up to a certain bound k characterises a graph transformation which eliminates a loop in a control flow graph without modifying the program semantics for k loop iterations. To put it differently, *loop unrolling* removes a loop in a control flow graph; however, the modified control flow graph represents k iterations of the loop that has been removed.

Definition 3.3.2 (loop unrolling). *Let G be a control flow graph for a sequence of program statements. It will be said that a loop l is unrolled k times, if G is transformed such that l no longer exists in G, but the program semantics are preserved for k iterations of l.*

As stated in Section 2.1.5, a reducible or natural loop is a loop with a single loop header. The loop header dominates every node in the loop. Therefore, the existence of a back edge – an edge from a node to a dominator of this node – implies the existence of a reducible loop. To unroll a reducible loop, the back edge has to be removed. Then, the nodes of the loop body will be duplicated k times for k loop iterations. In every case, the edges need to be set properly.

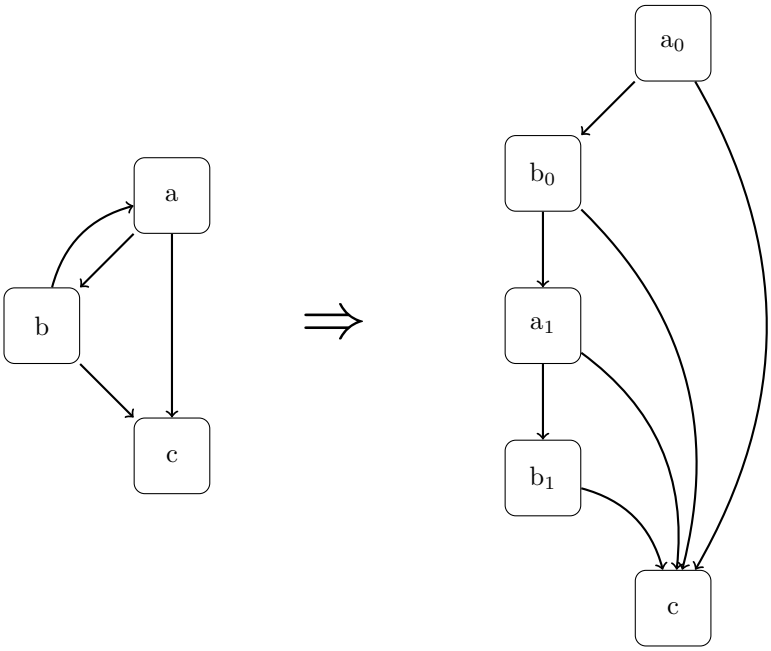

Figure 3.2: Unrolling a reducible loop $k = 2$ times

For instance, in Figure 3.2, the node a dominates b, the edge $e_b = (b, a)$ is a back edge and $l = \{a, b\}$ defines the loop body of the reducible loop l. Since the loop is unrolled $k = 2$ times, the nodes a_0 and b_0 represent the first loop iteration; a_1 and b_1 represent the second loop iteration. To preserve the program semantics, the loop edges are set properly for every iteration. The edge (a, c) is represented by (a_0, c) for the first and (a_1, c) for the second loop iteration. The same holds for (b, c). (b, a), the back edge, has been removed; instead, the next loop iteration is defined by (b_0, a_1). Since the walk $w = (a_0, b_0, a_1, b_1)$ already represents two loop iterations, there is no additional edge (b_1, c_2) for a third loop iteration. In other words, the loop condition no longer holds. This has to be considered when the intermediate representation is rewritten, as will be described in Section 3.4.

3.3.3 Unrolling irreducible loops

Contrary to reducible loops, irreducible loops are loops with multiple loop headers. Therefore, the dominance relations (cf. Section 2.1.4) for irreducible loops are different than those for reducible loops. As a result, the approach described in the previous section cannot be utilised for irreducible loops without difficulty.

As mentioned in Section 2.1.5, a method known as node splitting enables the transformation of irreducible loops into reducible loops in an exponential running time. Since the irreducible loops have to be unrolled, this approach is not required. Instead, for each loop header h, it will be assumed that h is the only loop header. Then, the dominance relations hold for the loop dominated by h; this loop can be unrolled with the method described in the previous section. Again, in relation to the program semantics, edges need to be set properly.

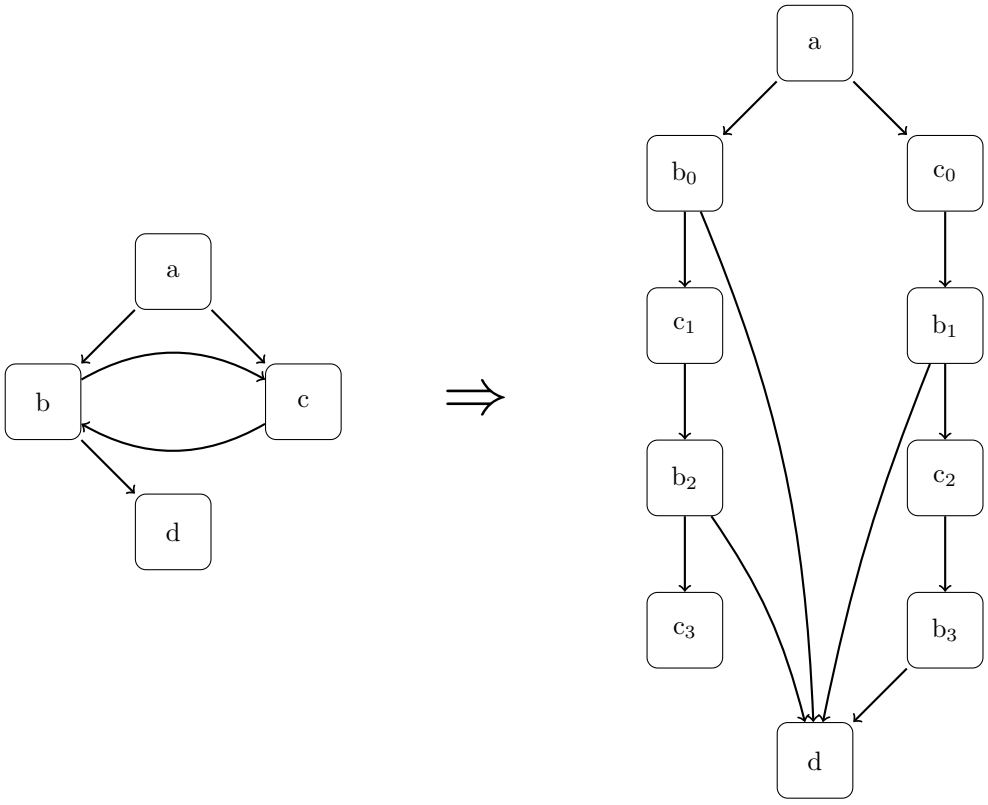

Figure 3.3: Unrolling an irreducible loop $k = 2$ times

To illustrate this concept, Figure 3.3 shows an irreducible loop between b and c; it holds neither $b \operatorname{dom} c$ nor $c \operatorname{dom} b$. Therefore, b and c are both loop headers of $l = \{b, c\}$. To unroll, first, assume that b is the loop header. Then, (c, b) is a back edge and b dominates c. As a result, l is a reducible loop that can be unrolled,

as described in the previous section. The edges (b_0, d) and (b_2, d) represent the edge (b, d); (b_0, c_1) and (b_2, c_3) represent the edges within the same loop iteration and (c_1, b_2) represents the back edge. Again, since (b_0, c_1, b_2, c_3) represents two loop iterations starting in b and since c has no other outgoing edges, the semantics for the loop starting in b are preserved. Next, assume that c is the loop header. Thus, (b, c) is the back edge and c dominates b. Thereby, again, l is a reducible loop and can be unrolled in the same way.

3.3.4 Procedure

In this section, a procedure which combines the previously described graph transformations to unroll a transition system will be outlined; as a result, the unrolled control flow graph will be a directed acyclic graph (cf. Definition 2.1.6). After that, Cylyx's approach will be presented. The graph algorithms used by Cylyx will only be mentioned, since they can be replaced by others with a similar running time and since they are out of the scope of this work.

First, to unroll a control flow graph with function calls k times, function inlining will be performed. Consequently, the resulting control flow graph does not contain any function calls; reducible and irreducible loops may exist. Therefore, irreducible loops will be unrolled k times. Since the dominance relations are not explicit in irreducible loops, they will be unrolled first. After that, each loop in the transformed control flow graph is reducible. Thus, reducible loops will be unrolled k times. Subsequently, the transformed graph is a directed acyclic graph without functions calls.

Cylyx extends the described approach for API functions. As stated in Section 3.2.2, API functions can be inlined or removed. First, defined API functions will be removed from the control flow graph. For this, the basic block which calls the API function (to be precise, the basic block that calls the thunk function, which jumps into the API function) will be removed; its direct predecessors will be connected with its direct successors. Second, defined API functions will be inlined. As of now, the control flow graph does not contain any API calls. Third, function inlining will be performed. As a result, the transformed control flow graph may only contain loops.

Loop detection will be performed by locating strongly connected components (based on an iterative version of Gabow's path-based algorithm [43]) with at least two distinct nodes. In this way, in relation to Section 2.1.5, outermost loops will be detected. Although outermost loops do not consider loop nesting, the program semantics will be preserved; if the unrolling bound k does not cover the required number of iterations of the innermost loop, k can be increased.

To localise irreducible loops, all back edges in the control flow graph will be detected and removed. (Cylyx includes algorithms used to locate back edges and natural loops, described by Alfred V. Aho et al. (cf. Section 9.6.6 [4]).) Since a back edge defines a natural loop, removing a back edge is equivalent to unrolling a reducible loop by $k = 1$. In other words, reducible loops will be eliminated in the control flow

graph. Therefore, the loop detection described above will only detect irreducible loops, which will be unrolled k times. Then, on the transformed control flow graph, the back edges will be re-added. Ultimately, the only possible loops are reducible loops, which will be detected and unrolled k times.

3.4 Rewriting the intermediate representation

Before the directed acyclic graph can be transformed into SSA, the unrolling of the transition system has to be finalised. So far, the intermediate representation and the control flow graphs of each function, as well as the directed acyclic graph of the unrolled transition system, are known. In the next step, the intermediate representation must be rewritten such that the program semantics will be represented by the transformed control flow graph. Additionally, the intermediate representation will be prepared for SSA transformation.

In order to rewrite the control flow, Cylyx performs a depth-first traversal on the directed acyclic graph, starting at the head. For each node, the corresponding basic block b will be analysed. If the direct successors of the current node in the directed acyclic graph differ from the direct successors of the basic block b, the last statement of the basic block, which directs the control flow, will be rewritten. Additionally, logical implications will be derived, which are dependent upon the jump condition. These implications will be used to encode the control flow graph, as will be described in Section 3.5.2. If a basic block corresponds to multiple nodes in the directed acyclic graph – for instance, if this basic block is part of an unrolled loop – then there will exist multiple rewritten copies of the basic block's intermediate representation, one for each corresponding node.

To exemplify, b is the current basic block and b_5 is the corresponding node in the directed acyclic graph. Additionally, let the last statement of b with the jump condition j be $j \,?\, c : d$, whereby c and d represent the direct successors of b in the control flow graph. Let b and c be part of the same loop, but not d. Then, the intermediate representation of b will be copied to create a new basic block b_5. The jump statement of b_5 will be rewritten to $j \,?\, c_5 : d$, since the direct successors of b_5 are c_5 and d. Additionally, logical implications will be derived. Assume that the instruction pointer IP will be set to depend on the jump statement, for instance $IP = j \,?\, c_5 : d$. Then, the implications $IP == c_5 \Rightarrow j == 1$ and $IP == d \Rightarrow j == 0$ will be derived; if c_5 is the next basic block that will be visited, then the jump condition must be true, otherwise, this condition must be false.

Similarly, let b be the same basic block and b_8 be the corresponding node in the directed acyclic graph. Let b_8 be the last loop iteration and (b, c) the back edge of this loop. Then, in relation to Section 3.3.2, b_8 will only have one direct successor, d, in the directed acyclic graph; however, c and d are the direct successors of b. For the jump statement of b, $IP = j \,?\, c : d$, the implications $IP == d$ and $IP == d \Rightarrow j == 0$

will be derived; since the back edge cannot be taken, the next block must be d and the jump condition has to be false.

Last, certain instructions will be rewritten in order to prepare for SSA transformation. For instance, as will be exposed in detail in Section 3.6, memory instructions will be rewritten such that memory access can be modelled as a load/store architecture. Then, the variable M represents global memory access and will be transformed into SSA.

3.5 Static single assignment

In this section, SSA transformation on a directed acyclic graph will be depicted. As a consequence of SSA, each variable definition is unique; the control flow graph will be transformed into a set of constraints. For this, one additional step has to be performed – the control flow has to be encoded. In the following, the construction of SSA and the encoding of the control flow will be outlined. Since constructing SSA is an intricate process, which is mostly based on algorithms described in standard literature, only the main ideas of this process will be mentioned [41]. Aside from that, the focus of this section rests upon encoding the control flow logically.

3.5.1 Construction

Constructing SSA relies strongly on graph-theoretic concepts, which are described in Section 2.1. As stated in Section 2.1.6, the two main tasks of SSA are renaming variables and inserting Φ-functions. Since variable renaming will also rename variables in Φ-functions, these functions must first be inserted. If not stated otherwise, the following steps are based on the work by Cytron et al. [41].

In relation to Definition 2.1.25, a Φ-function distinguishes the variable assignments of different execution paths in converging paths. Therefore, it is possible to insert a Φ-function for every variable in a basic block in which path convergence exists. However, Cylyx builds on an SSA construction, which is known as *minimal SSA*. In minimal SSA, the amount of Φ-functions is reduced since the insertion relies on dominance relations, especially on the dominance frontier (cf. Section 2.1.4). As a consequence of Definition 2.1.14, the dominance frontier of a basic block b defines the set of nearest basic blocks that are not dominated by b; a variable v, which is defined in b, may be redefined in the execution paths that do not depend on b. Therefore, a Φ-function for a variable v will be inserted into each basic block which is in the dominance frontier of a basic block b, for each defined variable v in b. Cylyx determines the dominance frontier of a node based on an algorithm by Cooper, Harvey and Kennedy [44].

In the next step, on the directed acyclic graph, the variables will be defined and the Φ-functions for the variables will be updated with the latest variable definitions.

In general, without delving into much detail, the worklist algorithm for variable renaming performs a depth-first traversal on the dominator tree (cf. Definition 2.1.13), starting at the head of the graph, and carries out the following steps. First, the variable definitions will be renamed for all Φ-functions in the current basic block. To exemplify, the variable definition $v = \phi(\dots)$ will be transformed into $v_1 = \phi(\dots)$. Second, for each assignment, all variable uses on the right-hand side will be renamed for non-Φ-functions. Third, all variable definitions on the left-hand side will be renamed for each assignment. For instance, the assignment $a = a + b$ will first be renamed to $a = a_0 + b_2$, then to $a_1 = a_0 + b_2$. Last, Φ-functions in the direct successors of the current basic block in the control flow graph will be updated on the right-hand side. For example, assume that v_3 is the last definition of a variable v. Then, if there exists a Φ-function for v in the direct successor of the current basic block, this Φ-function will be updated with v_3; $v = \phi(v_1, v_2)$ will be updated to $v = \phi(v_1, v_2, v_3)$.

Finally, some technical characteristics will be mentioned. Initially, variables in an assignment will first be updated on the right-hand side and then on the left-hand side, since SSA defines a def-use chain (cf. Definition 2.1.24) that starts with a variable definition of a variable v and ends with a reassignment of v. If an assignment $v = v + 1$ for a variable v will be transformed into SSA form, the right-hand side must be transformed first, since the definition of v on the left-hand side starts a new def-use chain for v. Second, Cylyx computes the dominator tree of a control flow graph by defining tree edges as edges from an immediate dominator of a node v to v. Therefore, Cylyx includes an algorithm which computes the immediate dominators of a directed graph. Third, the logical implications, which have been derived in the context of rewriting the intermediate representation (cf. Section 3.4), will also be transformed into SSA. Last, the final stage in the process of variable renaming – updating Φ-functions – will be extended and utilised to encode the control flow of the directed acyclic graph into logical formulas. This will be described in the following section.

3.5.2 Control flow encoding

This section describes how SSA can be used to encode the control flow logically such that it can be transformed into an SMT formula. First, the concept of Φ-functions will be extended. Then, it will be presented as these extensions will be generated. The following ideas are based on the work by Sinz, Falke and Merz [38].

As stated in Section 2.1.6, a Φ-function for a variable v returns the variable for the current execution path. In general, Φ-functions are not defined in a concrete manner. However, bounded model checking requires a concrete modelling of the program semantics. This will be achieved by extending the concept of Φ-functions with *edge conditions* that represent the current execution path.

Definition 3.5.1 (Φ-functions for bounded model checking, edge conditions). *Let $G = (V, E)$ be a directed acyclic graph. For each edge $e_i \in E$ there exists a condition c_i that can be true or false. If c_i is true, then e_i is in the current execution path. Otherwise, c_i is false. For a variable v, a Φ-function for bounded model checking is of the form*

$$v_j := \Phi((c_1, v_1), (c_2, v_2), \ldots, (c_n, v_n))$$

and can be translated into a statement of nested conditions, which has the form

$$v_j := c_1?v_1 : (c_2?v_2 : (\ldots (c_{n-1}?v_{n-1} : v_n) \ldots))$$

with $1 < n < j$.

To illustrate, in Figure 3.4, let the variables v_1, v_2 and v_3 be the last variable definitions of v in the corresponding execution paths labelled by the edge conditions c_1, c_2 and c_3. Then, the Φ-function $v_4 = \phi(v_1, v_2, v_3)$ for v in the basic block a will be extended to $v_4 = \phi((c_1, v_1), (c_2, v_2), (c_3, v_3))$. This can be translated into the logical formula $v_4 = c_1?v_1 : (c_2?v_2 : v_3)$. The edge conditions are mutually exclusive, since there exists only one execution path at a specific point in time.

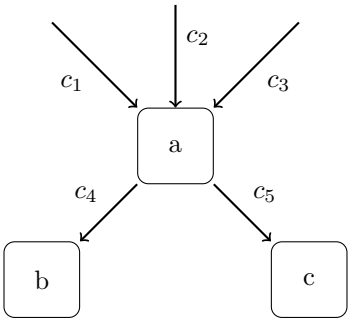

Figure 3.4: Control flow encoding based on edge conditions

As a consequence, a *visit flag* for a basic block can be defined. A basic block is visited, if one of its incoming edge conditions is true.

Definition 3.5.2 (visit flag). *Let $G = (V, E)$ be a control flow graph. Let $b \in V$ be a basic block and $e_1, e_2, \ldots, e_n \in E$ its incoming edges. Additionally, let $c_1, \ldots, c_n$ be the corresponding edge conditions. A basic block b is visited if its visit flag*

$$visit\_b = \bigvee_{k=1}^{n} c_k$$

is true.

To complete control flow encoding, the edge conditions must be set. Let b be a basic block with the visit flag $visit\_b$, the jump condition j and the two direct successors s_1 and s_2. Next, assume that the instruction pointer IP determines whether the control flow will be directed to s_1 or to s_2 with $IP = j \,?\, s_1 : s_2$. Then, the edge conditions will be defined as $c_{s_i} = visit\_b \wedge (IP == s_i)$ for $e = (b, s_i)$, with $i \in \{1, 2\}$. If b has only one direct successor s, then a jump condition does not exist. The instruction pointer is defined as $IP = s$. In this case, the edge condition will be defined as $c_e = visit\_b \wedge (IP == s)$ with $e = (b, s)$.

For instance, in Figure 3.4, let the instruction pointer IP for the jump condition j of a be $IP = j \,?\, b : c$. Then, c_4 is defined as $c_4 = visit\_a \wedge (IP == b)$. In other words, c_4 is true if the next basic block is b and a is visited. Equally, c_5 is true if a is visited and the next basic block is c.

Finally, this logical encoding of the control flow works as follows. The visit flags define a path from a basic block to the root of the directed acyclic graph. For instance, if the visit flag of b is true, c_4 has to be true since it is the only incoming edge of b. Therefore, a has to be visited. This implies that c_1, c_2 or c_3 must be true. On the other hand, the derived block conditions (cf. Section 3.4) encode the control flow from the head of the directed graph down to the leaves. For instance, the derived block conditions for a are $IP == b \Rightarrow j == 1$ and $IP == c \Rightarrow j == 0$. Therefore, the control flow is encoded for both directions.

3.6 Memory model

Memory management describes the strategy as memory access will be modelled. Based on Miasm IR, Cylyx rewrites the intermediate representation (cf. Section 3.4) such that it constitutes a load/store architecture. A *load/store architecture* characterises the property that exactly two operations exist to access memory [18].

Definition 3.6.1 (load/store architecture). *In a load/store architecture, memory access will be performed by two operations, load and store. Load reads a value from memory; store writes a value into memory.*

Memory will be modelled architecture-independently as a global array, which is represented by a variable M; Cylyx utilises a global memory model based on the theory of arrays (cf. Section 2.5.5). Memory access is represented by `v = mem_read(M, address, size)` for the read operation and by `M = mem_write(M, address, v, size)` for the write operation in the rewritten intermediate representation. Similar to the theory of arrays, a value v will be read from or written into an array M at an index *address*; equally, a value v or a memory array M, which encodes the write access, will be returned. The only difference is the additional parameter *size*; a value that will be read from or written into global memory has a defined byte size.

Cylyx implements a low-level memory model in SMT formulas. A memory array is a map of bit vectors with a size of l_1 to bit vectors with a size of l_2. Memory addresses

with a length of l_1 will be mapped to bit vectors with a length of $l_2 = 8$; the memory model is *byte-addressable* (cf. Section 4.6 [79]). The *word size* – the number of bits a processor architecture can process simultaneously (cf. Section 2.1 [79]) – defines the size of memory addresses. Then, memory reads and writes will be performed iteratively, byte per byte, until a value v of size s has been read or written. If the size s of a value v is not a multiple of 8 – if the size is not *aligned* (cf. Section 4.6 [79]) – then v will be aligned. Each operation can be fulfilled endian-independently, on *little-endian* architectures as well as on *big-endian* architectures.

For instance, on the x86 architecture, the word size is $l = 32$. Therefore, memory addresses with a size of 32 bit will be mapped to values with a size of 8 bit. Let v be a value with a size of $l = 15$ bit. Additionally, let *addr* be the memory address that contains v. Then, the expression `v = mem_read(M, addr, 15)` will be translated into $v == [\text{concat}(\text{read}(M, addr + 1), \text{read}(M, addr))]_0^{14}$. Since v has a size of $l = 15$, the size will be aligned to $l' = 16$. Then, two bytes will be read and concatenated in little-endian representation. Lastly, only the first 15 bits will be returned, which is denoted by $[a]_0^{14}$ for a value a.

The characteristics of this memory model facilitate an architecture-independent modelling of memory access. Additionally, since the memory array is a variable and since a memory write returns a memory variable, global memory can be transformed into SSA form. To put it differently, there exists a unique memory state for each program state. Lastly, the combination of SSA form and this memory model implies that an SMT solver handles *memory aliasing*. For instance, an SMT solver detects if `[EAX]` and `[EBX + 0x8]` point to the same memory location, since it knows the relations of `EAX` and `EBX`.

3.7 Formula generation and SMT solving

In the final step, the SMT formula consisting of the unrolled transition system, preconditions and postconditions will be created. Then, this formula will be solved. If it can be satisfied, there exists a solution for the input crafting problem for the defined bound k; otherwise, it is `UNSAT`.

Each program statement in the intermediate representation in SSA form will be translated into an SMT formula based on Miasm's translation for Miasm IR. Memory depends on the theory of arrays; variables are modelled in the theory of bit vectors. Therefore, the SMT formula will be a formula for the combined theory of quantifier-free bit vectors and arrays, QF_ABV (cf. Section 2.5.3).

Then, the SMT formula for the unrolled transition system, *prog*, consists of the conjunction of all variable definitions, Φ-functions, block conditions and control flow conditions. The postcondition defines the property that a basic block has to be visited. The visit of a basic block b will be forced by setting its visit flag to true, $visit\_b == 1$. If b has been unrolled k times in a loop, the postcondition consists of

the disjunction of the visit flags of the corresponding basic blocks, $postcondition = (\bigvee_{i=0}^{k-1}(visit\_b_i) == 1)$. The preconditions are optional and can be used for modelling assumptions; for instance, if variables of a function are known to be limited before they will be used in the modelled program semantics. Finally, the SMT formula φ will be defined as $\varphi = preconditions \wedge prog \wedge postcondition$.

If the SMT solver returns `UNSAT` for φ, then there exists no solution for the input crafting problem for the unrolled transition system. The bound k can be increased and the procedure repeated. If φ is satisfiable, then the SMT solver returns a model m. m consists of a set of assignments for all variables, visit flags and memory values. These depend on the control flow to the basic block b that must be visited. The inputs can be parsed in m by locating the variable or memory address where they will be defined and stored. Additionally, it is possible to reconstruct the execution path by filtering the true visit flags of the basic blocks.

4 Discussion

After the approach of bounded model checking for binary input crafting has been described, the following sections discuss its strengths and weaknesses as well as its applicability on real-world programs. For this, the methodology will be outlined, which forms the basis for the case studies that follow. Then, the experimental setup will be described. Following, different case studies will be performed, which facilitate an evaluation of this approach. Finally, based on the results of this chapter, the research question will be answered: Is bounded model checking for binary input crafting feasible and efficient?

4.1 Methodology

To examine the feasibility and efficiency of bounded model checking for binary input crafting, the characteristics of bounded model checking on the binary level must be evaluated. These characteristics will be assumed to be the impact of loops, function calls, API calls, memory operations, the amount of instructions and the number of paths. This assumption is premised on the fact that loops, function calls, API calls and memory operations are common structures on the assembly level. In addition, since the amount of instructions and the number of paths can be large, they may have a considerable influence on efficiency.

To evaluate these characteristics, case studies will be designed that combine some of those properties. For each case study, a concrete test case will be implemented and Cylyx will be used in its assessment. After this, Cylyx will be applied to real-world binaries to estimate the viability of this approach in practice. It must be admitted that the test cases, in general, cannot be representative for binary input crafting on the basis of bounded model checking. A multitude of factors exist that may influence the results, for instance, compiler optimisations, incomplete disassembly, imprecise modelling of the program semantics or program structures, which are not known to be difficult for SMT solvers. Therefore, the test cases cannot be complete and cannot cover all possibilities. Despite the test cases not being fully representative, a methodology based on the test cases supports classifying the advantages and disadvantages of this approach.

In the following case studies, test cases will be designed for loop unrolling, nested arrays, function calls and path explosion. Loop unrolling allows for analysis of the impact of the amount of instructions for SMT solving. Furthermore, the case studies for nested arrays and function inlining will be used to evaluate the effects of memory operations and API calls. Lastly, the case study for path explosion examines the

efficiency of path selection by SMT solvers and compares the results to the efficiency of symbolic execution.

It will be evaluated to what extend this approach can be applied independent to architecture. Therefore, some case studies will be carried out on different architectures. Other case studies, which are designed for different goals (e.g., path selection and feasibility on real-world binaries), will only be evaluated for the x86_64 architecture, since Cylyx has mostly been tested for this platform.

The amount of instructions, as well as the time needed to generate and solve the SMT formulas, will be measured and compared. If it is practicable, the SMT formulas will be solved by two independent SMT solvers in order to obtain less biased results. If a user input satisfies the SMT formula but not the program semantics, the user input will be crafted by symbolic execution to verify the results of Cylyx.

In contrast to the designed test cases, which analyse the strengths and weaknesses of the described approach, the main focus of examination for the real-world programs remains on feasibility. It will be determined whether or not the approach is feasible in practice and which challenges may occur.

In general, the structure of a test case takes the following form: A set of integers builds the input of a function. The function performs operations in dependence on the inputs and returns 1 if certain conditions are true, which depend on the input itself; otherwise, it returns 0. The function may contain loops or calls to other functions of a similar form, but no API calls. The crafted inputs satisfy the conditions such that the function returns 1. If not stated otherwise, the test cases will be compiled for the x86_86 architecture, as mentioned above.

4.2 Experimental setup

In this section, the experimental setup for the proceeding case studies will be described. The hardware configuration consists of an Intel Core i7-620M (2 cores, 4 MB cache, 2.66 GHz), 8 GB RAM and 10 GB swap space. Based on an *Arch Linux* [80], test cases will be compiled at optimisation level 0, with the *GNU Compiler Collection (GCC)* [81] (version 5.2.0) and debugged with the *GNU Project Debugger (GDB)* [82]. *QEMU* [83] and Miasm's just-in-time compilation will be used to emulate executables of non-native targets. Symbolic execution will be performed on the basis of scripts, which are included in Cylyx, if operations of Cylyx have to be verified; to perform symbolic execution for path exploration, *angr* [24], a framework for binary program analysis, will be used. The reason for choosing angr is twofold. First, it performs symbolic execution on the binary level for different architectures. Second, its 'approach to symbolic execution draws on concepts proposed' (p. 8 [24]) in KLEE [7], FuzzBALL [23] and Mayhem [22], which are concepts of well-known symbolic execution engines, 'adapted to [the authors'] specific problem domain' (p. 8 [24]).

Furthermore, in this setup, Cylyx relies on Miasm's last git commit from 30th October 2015. To solve SMT formulas, the current versions of the SMT solvers, Z3 (version 4.4.2) and *Boolector* [84] (version 2.1.1), will be utilised. The translation of Miasm IR into SMT formulas for Z3 justifies the usage of this SMT solver. In addition, Boolector will be used in comparison to Z3, since it won first place in the track QF_ABV of the *10th International Satisfiability Modulo Theories Competition (SMT-COMP 2015)* [85]; as stated in Section 3.7, all formulas created by Cylyx are included in this track. To solve such an SMT formula with Boolector, this formula will be transformed so that it is compatible with the SMT-LIB standard. This transformation is based on a feature of Z3. Lastly, for API function inlining, the library *dietlibc* [86] (version 0.33), which is optimised for small size, will be used, since the resulting assembly code is less complex.

4.3 Case study 1: loop unrolling

In this case study, the impact of the amount of SSA instructions on SMT solving will be examined. In addition, it will be demonstrated that SMT solvers are able to exploit structures of certain loops.

4.3.1 Description

The function consists of a loop that increments a counter up to a certain value. Within this loop, the truth value of a condition, which is based on the loop counter and the function's input parameters, decides if the function returns 1. For instance, as illustrated in Figure 4.1, based on the input parameters a and b as well as the loop counter c, with $0 \leq c < 1000$, the function `foo` returns 1 if and only if $(a + b == 1337) \wedge (c == 50)$ is true. In the following analysis, this loop has been unrolled with increasing unrolling bounds.

4.3.2 Analysis

Table 4.1 shows, in relation to Figure 4.1, the number of SSA instructions for each unrolling bound, as well as the measured time for SMT formula generation and solving. Since loop unrolling copies the basic blocks of the loop body in each unrolling step k times, as it can be seen, the number of the SSA instructions rises linearly to the unrolling bound. The creation of the SMT formula consists of the process described in Section 3.1.2, excluding SMT solving. Since the procedures of generating SMT formulas are identical up to the step of loop detection, the following steps determine the required time: loop unrolling, instruction rewriting, SSA transformation and translating the SSA instructions into SMT formulas. In other words, the generation of SMT formulas mainly depends on the polynomial complexity of the graph algorithms, at least in this case.

```
int foo(int a, int b)
{
    int c = 0;
    while (counter < 1000)
    {
        if ((a + b  == 1337) && (c == 50))
                return 1;

        counter++;
    }
    return 0;
}
```

Figure 4.1: Simple loop condition relying on an incremented counter

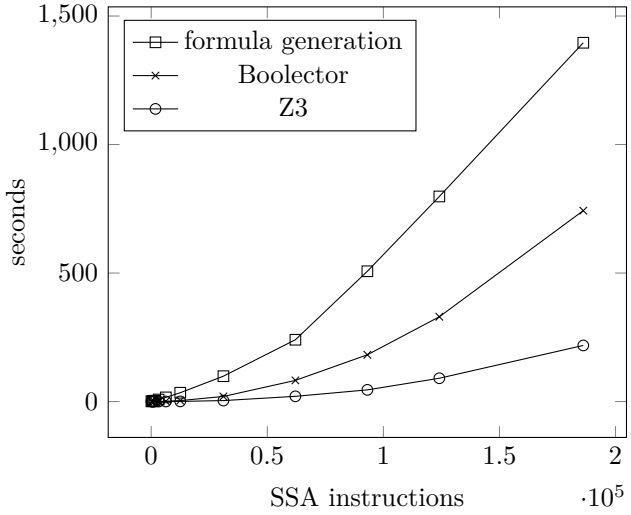

Figure 4.2: Polynomial growth in the number of SSA instructions

Since the number of paths grows exponentially in the unrolling bound, it may be assumed that the time for SMT solving grows also exponentially in the unrolling bound or, equivalently, for this case, in the number of SSA instructions. However, as Table 4.1 and the corresponding visualisation in Figure 4.2 show, the SMT solving time grows polynomially in the number of SSA instructions. It can be proven by curve fitting (the construction of a mathematical function which has the best fit to a set of data tuples [87]) that the curves of formula generation and Z3 grow quadratically and the curve of Boolector grows cubically. To put it differently, although the number of paths grows exponentially in the unrolling bound, SMT solving time remains

unroll factor	# SSA instructions	formula generation	Z3	Boolector
0	90	0.8 s	0.01 s	0.01 s
5	421	1.3 s	0.02 s	0.02 s
10	731	2.0 s	0.03 s	0.03 s
20	1351	3.4 s	0.04 s	0.07 s
50	3211	8.0 s	0.11 s	0.42 s
100	6311	16.2 s	0.25 s	1.28 s
200	12511	34.4 s	0.61 s	3.81 s
500	31111	98.8 s	3.90 s	19.89 s
1000	62111	240.6 s	20.40 s	82.38 s
1500	93111	506.96 s	45.51 s	182.02 s
2000	124111	797.94 s	80.75 s	329.94 s
3000	186111	1395.97 s	218.31 s	742.84 s

Table 4.1: Influences of loop unrolling on SMT solvers

polynomially. From this, it follows that SMT solvers exploit the high-level structure of this function.

An SMT solver has to learn two conditions; $a + b == 1337$ and $c == 50$ must be true. Due to the SSA form, an SMT solver deduces that a and b are free variables. (a and b are free variables because they do not depend on previous calculations. In other words, they will be used on the right-hand side before they will be redefined on the left-hand side.) Therefore, guessing a solution is straightforward. For the second condition, if an SMT solver guesses a value $1000 > c' > 50$, it learns that the condition $c < c'$ must hold and will prune the search space for all paths that do not satisfy this condition. Last, if the SMT solver guesses $c' > 1000$, it learns that the loop cannot be entered, since the loop condition $0 \leq c' < 1000$ does not hold.

To conclude, while the number of SSA instructions grows linearly in the unrolling bound, the time for SMT solving grows polynomially. Therefore, SMT solvers are able to exploit the structure of this loop. Lastly, it must be noted that, on average in this case study, Z3 was four times as fast as Boolector.

4.4 Case study 2: nested arrays

The second case study was designed to examine the efficiency of memory operations, nested conditions and memory aliasing for different architectures. The test case includes function inlining and is loop-free.

4.4.1 Description

In this case study, a function contains operations on arrays, whose values are initialised in dependence on the input parameters. Then, several conditions, based on the input parameters and the arrays, redefine certain array elements such that memory aliasing may occur. Additionally, a function of the same structure will be called, whose return value is part of the condition that decides if 1 will be returned. For instance, the implementation, which is the basis of the following analysis and is illustrated in Listing B.1, has two input parameters, a and b. A satisfying model is $[a = 1431655878, b = 1431656874]$.

4.4.2 Analysis

Similar to the previous case study, Table 4.2 shows the number of SSA instructions and the times of generating and solving the SMT formulas. Instead of an unrolling bound, the first column represents the architecture for which the binary has been created – the main architectures supported by Miasm.

architecture	# SSA instructions	formula generation	Z3	Boolector
MIPS32	234	1.7 s	49.1 s	2.1 s
ARMv7	245	2.4 s	21.5 s	0.4 s
AARCH64	268	2.9 s	162.3 s	0.9 s
x86	412	1.6 s	12.8 s	0.8 s
x86_64	402	1.7 s	31.7 s	1.0 s

Table 4.2: Influence of nested arrays on different architectures

First, Table 4.2 shows that the SMT formula will be generated and solved for each architecture between a period of three to four seconds. However, a significant difference between Z3 and Boolector exists: Boolector is up to 162 times faster than Z3. Furthermore, the time it takes for Boolector to solve SMT formulas is mostly consistent across different architectures, while the times for Z3 are inconsistent.

Without generalising, one possible reason for this might be the assumption that Boolector is much more efficient in memory operations (in the theory of arrays) than Z3, since calling functions and operations on arrays depend heavily on memory operations. This could be substantiated by results of the competition SMT-COMP 2015, in which Boolector is seven times faster than Z3 in the main track of QF_ABV (sequential performance) [88], all while solving more benchmarks.

For the MIPS32 and ARMv7 architectures, the SMT solvers return a model for a and b, in which the inputs do not direct the control flow such that 1 will be returned. However, the execution path in the model is a possible execution path that may or may not be satisfiable. Therefore, the SMT formula encodes the control flow, but

does not represent the program semantics. To verify this result, the inputs will be crafted symbolically, based on Miasm's intermediate representation. The derived path conditions will be solved by Z3, whereby a model contains crafted inputs. In this case, these inputs also do not have the desired characteristics. Thus, it is safe to assume that some transformations from assembly instructions into Miasm IR do not represent the instruction semantics.

Lastly, while Z3 requires 31.7 seconds for 402 SSA instructions on the x86_64 architecture, as seen in Table 4.2, the SMT solver locates a solution in 20.4 seconds for the 62111 SSA instructions in Table 4.1. As a result, it can be said that no necessary link exists between the amount of SSA instructions and the time it takes to solve an SMT formula. Therefore, it can be deduced that the efficiency of input crafting on the basis of bounded model checking depends on SMT solvers' ability to exploit the underlying program structure.

4.5 Case study 3: function inlining

To improve the results of the preceding section, this case study will also be examined for different architectures and includes memory operations and nested conditions. The focus here is on complex code, nested function calls and inlining of API functions.

4.5.1 Description

The structure of this test case differs from the usual design. Firstly, a function `foo` may return different values, in dependence on the inputs. Then, it calls a function `bar`, which performs calculations on the inputs, but returns a constant value. A function `main` calls API functions and `foo`. Lastly, a condition in `main` depends on the return value of `foo`.

In the code listed in Listing B.2, the function `main` calls the API function `atoi` thrice, once for each input a, b and c. If `foo` returns 4, then the API function `printf` will be called and will print a string. If `foo` returns a different value, then `printf` will print a different string. The function `foo` contains a loop, nested conditions and array operations. Additionally, it calls `bar`, which returns 4; `foo` will return 4 if and only if `bar` will be called. A satisfying model is $m = [a = 416029827, b = 2055293730, c = 254662596]$.

Subsequently, two different scenarios will be examined. Firstly, to investigate the efficiency of complex code consisting of function calls, loops, arrays and arithmetic operations, the input will be crafted for `foo` such that `bar` is called and `foo` returns 4. This will be examined for different architectures. Secondly, the efficiency of API call inlining will be inspected. For this, for the x86_64 architecture, the code will be crafted for `main` such that `foo` returns 4. The API function `atoi` will be inlined, while `printf` will be removed from the control flow graph.

4.5.2 Analysis

As in the previous case study, Table 4.3 shows the number of SSA instructions, as well as the measured time of SMT formula generation and solving for different architectures. Additionally, it includes the starting point of the input crafting. The last row in the table represents the test case with API function inlining. Loops have been unrolled nine times.

architecture	# SSA instructions	formula generation	Z3	Boolector	start
MIPS32	681	3.0 s	38.2 s	14.9 s	`foo`
ARMv7	875	4.0 s	$>$ 7200 s	$>$ 7200 s	`foo`
AArch64	-	-	-	-	`foo`
x86	1199	3.6 s	20.0 s	32.0 s	`foo`
x86_64	1181	3.9 s	24.4 s	11.8 s	`foo`
x86_64	4008	9.4 s	$>$ 43200 s	278.8 s	`main`

Table 4.3: Effects of function inlining on SMT solvers

First, as it can be seen, no solutions exist for the AARCH64 and ARMv7 architectures. In the case of the former, some assembly instructions are not supported by Miasm. In the case of the latter, both SMT solvers do not terminate within two hours. Further, as in the previous case study, inputs crafted by symbolic execution do not satisfy the path condition. Therefore, it can be assumed that some transformations from assembly code into the intermediate representation do not represent the corresponding instruction semantics. These examples demonstrate the necessity to model program semantics precisely.

Second, the requisite time to solve SMT formulas differs for all architectures. Boolector requires less time to solve the formula for the x86_64 architecture than for the x86 architecture, although the amount of SSA instructions as well as the underlying program structure, which the SMT solver exploits, are similar. A reason for this may be the usage of different calling conventions on the assembly level. In this test case, for the x86 architecture, function parameters are passed on the stack, while, for the x86_64 architecture, function parameters are passed in registers. To specify, since Cylyx utilises a global memory model (cf. Section 3.6), an SMT formula contains nested array writes. Deeply nested array writes are known to be less efficient [89]. Therefore, it may be the general case that, for bounded model checking, operations in the theory of arrays are less efficient than in the theory of bit vectors.

Third, as evidenced by Table 4.3, SMT solving requires significantly more time when API functions are inlined. While Boolector is 23 times slower with API function inlining than without it, Z3 does not terminate within 12 hours. The reasons for this are twofold. On the one hand, the structure of the API functions and

their instructions build an additional overhead. On the other hand, function calls themselves may be less efficient, since a function operates in its own memory space and an SMT solver has to keep track of the links between the memory spaces, as well as the operations inside the function. Furthermore, this is supported by the assumption that operations in the theory of arrays may be less efficient. Last, it must be denoted that the results of this case study sustain the observation of Section 4.4, which states that Z3 seems to be far less efficient with respect to memory operations than Boolector.

To conclude, the results of this case study are that API call inlining is inefficient, while function inlining and, in particular, its corresponding memory operations may decrease the efficiency of binary bounded model checking. Additionally, the importance of preserving the program semantics and Boolector's advantage on memory operations have been pointed out.

4.6 Case study 4: path selection

In the following case study, the efficiency of SMT solvers on the path selection problem will be examined. Then, the results will be compared to the performance of a symbolic execution engine.

4.6.1 Description

To evaluate the efficiency of SMT solvers in path exploration, this case study was designed such that it is improbable that an SMT solver can exploit the program structure with less effort. The goal is to generate a number of execution paths that are, as far as possible, independent from each other. For this, opaque predicates will be utilised. Informally, an *opaque predicate* is a predicate of first-order logic, whose truth value is known a priori [90]. To put it differently, an opaque predicate can either only be true or only be false.

The general structure of the function in this test study consists of n conditions, $c_1, \ldots, c_n$, which depend on the user input and can be true or false. This function will return 1 if the conjunction of these n conditions is true. Then, c_i will be replaced by n new conditions, $d_1, \ldots, d_n$, whereby each d_i consists of the disjunction of the corresponding c_i and k distinct opaque predicates. Each opaque predicate evaluates as false and depends on the user input. In total, there exist $n \cdot k$ distinct opaque predicates. Figure 4.3 illustrates this process for $n = 4$. To exemplify, the function returns 1, if the conditions $c_1, \ldots, c_4$ are true. Let, for instance, c_1 be $c_1 = (a + b == 1337)$, whereby a and b are the functions' inputs. Then, c_1 will be replaced with $d_1 = c_1 \vee op_1 \vee \cdots \vee op_k$, whereby all op_i, with $i \in \{1, \ldots, k\}$, are pairwise distinct opaque predicates. After that, the function will return 1, if the conjunction of all d_i, $d_1 \wedge \cdots \wedge d_4$, is true. As a result, the function returns true if and only if each c_i is true, since the opaque predicates cannot be true (they evaluate

as false by design). In the control flow graph of this function, many paths exist such that 1 will be returned, but only one path is satisfiable.

```
if ((c_1 || op_1 || ··· || op_k)
    && (c_2 || op_{k+1} || ··· || op_{2k})
    && (c_3 || op_{2k+1} || ··· || op_{3k})
    && (c_4 || op_{3k+1} || ··· || op_{4k}))

    return 1;
```

Figure 4.3: Obfuscating a condition with opaque predicates

In the implementation of this test case, the function has two input parameters and it holds $n = 4$, as illustrated in Figure 4.3. In the analysis, the bound k, representing the number of opaque predicates for each condition c_i, will be increased and the input for this function crafted by an SMT solver. c_1, c_2, c_3 and c_4 are the conditions, which depend on the user input, a and b, and have to be true; Further, $op_1, \ldots, op_{4k}$ represent the $4 \cdot k$ distinct opaque predicates, which also depend on the user input. For instance, an opaque predicate may be of the form $(((a + 895543861) == (a + (a \cdot a \cdot 1294569136) + 1931398089)))$, for bit vector a of the length $l = 32$.

The opaque predicates have been randomly generated and are proven to be unsatisfiable by an SMT solver for bit vectors of the length $l = 32$, both in general and on the assembly level. To generate an opaque predicate, arbitrary arithmetic expressions (without bit shifts, division and modulo operations to exclude division by zero) will be randomly generated. Expressions depend on a bit vector variable a and random constants of the same size as a. If an SMT solver can prove, within two seconds, that the equality of two of those expressions is unsatisfiable, then it is an opaque predicate that evaluates to false. Limiting the decision time of the SMT solver to two seconds implies that a single opaque predicate cannot cause a long running time of an SMT solver, which would falsify the results.

4.6.2 Analysis

Table 4.4 shows the number of SSA instructions, as well as the time to generate and solve an SMT formula, for different numbers of opaque predicates k. In the following, k represents the total number of opaque predicates. Since $n = 4$, there exist $\frac{k}{4}$ opaque predicates for each c_i. For instance, if $k = 800$, then each disjunction contains 200 opaque predicates. Additionally, the table lists the time in which angr, a symbolic execution framework, locates the one possible path. Path exploration with angr has been performed on the basis of Listing B.3. To recapitulate, angr will be used since it is able to perform symbolic execution on the x86_64 architecture and since it is based on techniques, amongst others, proposed in KLEE [7].

# predicates	# SSA instructions	formula generation	Z3	Boolector	angr
0	147	0.9 s	0.0 s	0.0 s	2.6 s
40	2172	7.0 s	0.4 s	2.7 s	49.6 s
80	4074	13.6 s	1.1 s	6.2 s	110.2 s
120	5438	18.2 s	2.5 s	7.1 s	147.2 s
200	8270	27.8 s	5.1 s	6.4 s	329.5 s
400	16617	57.2 s	18.1 s	23.1 s	745.1 s
800	34028	119.9 s	111.5 s	18.1 s	1785.1 s
1200	51267	180.0 s	335.7 s	49.1 s	4265.0 s
1600	67107	247.0 s	715.9 s	208.3 s	5871.1 s
2000	82291	325.1 s	956.9 s	299.6 s	10534.5 s

Table 4.4: SMT solvers and path exploration

In theory, for a given k, there exist $(\frac{k}{4}+1)^4$ different paths between the function start and the basic block that returns 1. However, in practice, the total number of paths may differ for two reasons. First, the compiler may eliminate some opaque predicates during compilation. Second, each opaque predicate may be represented by several basic blocks in the intermediate representation. Thus, to find the best fit for the growth of the number of paths, this number will be calculated on the basis of the directed acyclic graph for some k. 14641 paths exist for $k = 40$ (47 basic blocks) and 176400 paths for $k = 80$ (85 basic blocks). In addition, there exist 4753392 paths for $k = 200$ (190 basic blocks) and 77995008 paths for $k = 400$ (379 basic blocks). For $k > 400$, the calculation of the number of paths does not terminate after several hours. However, for $k = 1200$, the number of basic blocks is 1130 and 1837 for $k = 2000$. Therefore, the number of paths grows polynomially, since the best fit for tuples consisting of the number of opaque predicates and the number of paths is a curve 3 or 4 degrees.

As it can be seen in Table 4.4, SMT solvers are very efficient in path selection. For instance, Cylyx locates the only satisfiable path out of 77995008 possible paths ($k = 400$) within 18 seconds, using Z3 as the SMT solver. In this case, it is 10 times faster than angr, which finds the same path in 745 seconds. Further, for $k = 2000$, Cylyx is 17 times faster, using Boolector as the SMT solver. This is also illustrated in Figure 4.4, which visualises the results of both Cylyx and angr using the time for SMT formula generation and the minimal time for SMT solving. As a result, the factor of speed in which Cylyx is faster than angr grows in the number of opaque predicates.

Since deciding the quantifier-free fragments of the theories of arrays and bit vectors is NP-complete (cf. Section 2.5.3), it will be expected that SMT solving becomes inefficient for some k. However, this k has not been found. For instance, Cylyx locates the satisfying path for $k = 4000$ (3679 basic blocks) in 1498.1 seconds, whereby

Boolector decides the SMT formula within 712.6 seconds; Cylyx is seven times faster for $k = 4000$ than angr is for $k = 2000$.

Lastly, Z3 and Boolector differ in their timings. In the beginning, Z3 is faster than Boolector. However, for $k \geq 800$, Boolector is significantly faster. Since the previous results do not indicate a possible reason for that, it will be grasped as an observation.

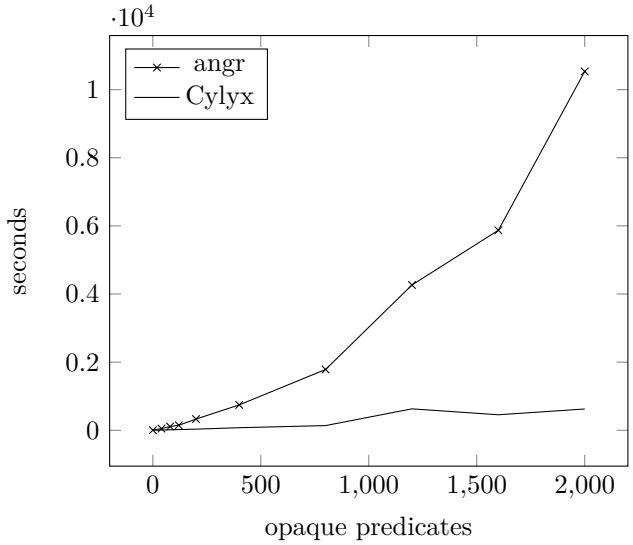

Figure 4.4: Growth of path exploration in the number of opaque predicates

4.7 Case study 5: real-world programs

After the strengths and weaknesses of bounded model checking for binary input crafting have been examined, this case study investigates the applicability of this approach to real-world programs. In addition, possible challenges will be demonstrated.

4.7.1 Description

Subsequently, the process of input crafting on real-world binaries will be described. For this process, Cylyx will be utilised to analyse functions in two different binaries. Functions similar to the previous test cases will be chosen; functions may contain function calls, loops and receive inputs as function parameters. Those functions facilitate a comparison with the previous case studies, since they share similar characteristics.

First, the function `base64_decode_ctx` of the binary *base64* from the *Coreutils* [91] will be examined. Second, input will be crafted for the function `hwaddr_aton` of the *WPA supplicant* [92]. Last, the results will be analysed.

4.7.2 Base64

The inputs of the function `base64_decode_ctx` are a context object, a pointer to an input buffer, a pointer to an output buffer and the length of both buffers. The input buffer contains a string, which is base64 encoded; the output buffer shall contain the decoded string, after the function returns. The function prototype is `[...] (struct base64_decode_context *ctx, const char *restrict in, size_t inlen, char *restrict out, size_t *outlen)`. This function consists of several loops, nested conditions, an API call to `memchr` and a function call to `decode_4`, which contains a loop and conditions.

Cylyx is able to inline the functions and unroll the loops. However, it is not possible to inline the API function, since Miasm does not support transforming the assembly instruction `pshufd` into Miasm IR. As a result, the program semantics cannot be encoded precisely and input crafting cannot be performed.

4.7.3 WPA supplicant

The function `hwaddr_aton` converts an ASCII string into a hardware address, represented as bytes. For instance, the hardware address 'aa:bb:cc:dd:ee:ff' will be converted into $0xaabbccddeeff$. Two parameters will be passed to this function – a pointer to a char buffer containing the ASCII string, in `RDI`, and a pointer into a buffer consisting of bytes, in `RSI`. The function calls another function, `hwaddr_parse`, and in dependence upon the return value of the called function, it returns 0 if the string is a valid hardware address; otherwise, it returns -1. Cylyx will be utilised to craft a valid input such that 0 will be returned; a valid hardware address will be crafted. However, the structure of the underlying functions will be discussed beforehand. The assembly code of those functions is listed in Appendix B.4.

The called function, `hwaddr_parse`, consists of nested conditions, a loop and a function call to `hex2byte` within this loop. The loop will be passed a maximum of six times, whereby the ASCII string will be separated at the colon. For instance, the first loop iteration processes 'aa', the second loop iteration 'bb' and the last loop iteration 'ff'. Each time, `hex2byte` will be called, which transforms a string into a byte; for instance, 'aa' will be transformed into $0xaa$. This function contains nested conditions and two function calls to `hex2num`. To `hex2num`, a single character will be passed and the corresponding byte value returned. For instance, for the character 'a' with the ASCII value $0x61$, the value $0xa$ will be returned. In total, to craft a valid hardware address, the loop has to be unrolled six times and 19 functions must be inlined.

Cylyx generates an SMT formula with an unrolling bound of $k = 7$ in 10.2 seconds. In total, this SMT formula contains 4230 translated SSA instructions. Boolector locates a satisfying model within 258 seconds. However, the crafted input does not represent a valid hardware address, but the execution path is equal to that of a

dynamic trace, whose input is a valid hardware address. Further analysis reveals the reason for this behaviour. The SMT formula contains four free variables – `RDI`, `RSI`, `RBP` and `RSP`. (They are free because they are not bounded by other variables.) Therefore, the SMT solver has to estimate a satisfying memory layout, as well as values for those free variables. In this case, the SMT solver sets `RDI` and `RSP` to the same value. Since `RSP` operates as the stack pointer, which will be used for referencing local variables, and since `RDI` contains a pointer to the char array, this will cause a conflict, because two distinct memory regions overlap. To put it differently, although the SMT formula represents the program semantics of the function `hwaddr_aton`, the SMT solver assigns values for free variables such that the memory layout does not represent the program semantics.

To solve this problem, preconditions will be applied. The preconditions set `RDI`, `RSI`, `RSP` and `RBP` to different values such that they cannot overlap. Then, Boolector solves the SMT formula in about 260 seconds. In this case, the crafted input represents a valid hardware address, 'eA:79:82:f4:d9:ff'; the input can be crafted successfully.

4.7.4 Analysis

The results of this case study are threefold. Firstly, bounded model checking for binary input crafting is limited in practice, since the program semantics have to be precisely modelled. Both an API function, which cannot be inlined, or an assembly instruction, which cannot be transformed into the intermediate representation, may influence the approach. Second, SMT solvers may operate in an unexpected and unpredictable manner. If this behaviour is known or can be figured out, it may be handled by applying preconditions. Last, it has been shown that functions in real-world binaries exist, for which bounded model checking for binary input crafting can be applied efficiently. The underlying algorithms have not to be reverse engineered in detail; defining a starting point, a basic block that must be reached and a bound for loop unrolling are sufficient for crafting satisfying inputs.

4.8 Conclusion

As stated in Section 1.3, the research question asks whether bounded model checking to craft inputs for binary programs is both feasible and efficient. To answer this question, a process to apply bounded model checking to binary programs has been described in Section 3. The main idea of this process is to create an SMT formula of a directed acyclic graph, which will be transformed into SSA. Then, in this chapter, methods have been derived and case studies designed such that the main characteristics of this approach, which are described in Section 4.1, can be examined. The Cylyx framework, which is based on methods described in Chapter 3, has been used for investigation.

As exhibited in the previous sections, the number of SSA instructions is not related to the required time for SMT solving. In some cases, the satisfiability of 268 instructions in SSA form can be decided in several minutes by an SMT solver (cf. Section 4.4), while, in other cases, the satisfiability of 31111 SSA instructions can be decided within seconds (cf. Section 4.3).

In the case of path selection, it has been shown that bounded model checking is very efficient, in many cases. For instance, in the case of loop unrolling in Section 4.3, where the number of paths grows exponentially in the unrolling bound, bounded model checking remains polynomially since SMT solvers are able to exploit the structure of the program. Additionally, Section 4.6 demonstrated that, in other cases, bounded model checking is significantly more efficient in path selection than the symbolic execution engine angr.

Furthermore, it has been shown that function inlining is less efficient. For this, a relation to the supposed inefficiency of memory operations (in particular, nested memory writes) has been assumed in Section 4.5. Primarily, it has been stated that inlining of API functions is inefficient. However, API functions may be handled more efficiently by modelling their behaviour, as described in Section 3.2.2.

Apart from these results, some observations have been made. First, Z3 seems to be significantly slower than Boolector in memory operations. Secondly, in general, predicting the necessary time to solve an SMT formula is difficult. In some cases, within the scope of this dissertation, it cannot be clearly stated whether SMT solvers are efficient or inefficient (cf. Section 4.6). Lastly, as denoted in Section 4.7.3, SMT solvers may operate differently than expected, which may be caused by underlying assumptions that have not been modelled explicitly.

To summarise, bounded model checking is feasible for binary input crafting, as long as the program semantics can be precisely modelled. In practice, this precise modelling is the main limitation of this approach. In many cases, the approach works efficiently, since SMT solvers exploit the structure of the program semantics. As a result, they are very efficient in path selection. On the contrary, nested memory writes decelerate SMT solving. In general, this approach is limited by complexity, because it operates on problems that are, in the best case scenario, NP-complete. Nevertheless, it has been demonstrated that bounded model checking on binary programs is architecture-independent and can be applied efficiently and successfully to input crafting on real-world binaries.

5 Future work

This dissertation introduced bounded model checking for binary programs and its application to input crafting. In this chapter, future work will be discussed. Improvements and optimisations on the implementation level will be outlined and future research will be described.

5.1 Implementation level

In the following, concepts on the implementation level will be presented, which may improve the described process of bounded model checking on the binary level (cf. Chapter 3). While some concepts may improve the precision of modelling the program semantics, other concepts may optimise its efficiency.

Section 2.1.6 illustrated the power of SSA to optimise code; SSA simplifies essentially algorithms for optimisations, such as dead-code elimination or constant propagation, data flow analysis and decompilation [93]. Therefore, in the process of bounded model checking, applying optimisations on the SSA instructions may significantly reduce the complexity of the SMT formula and improve the performance of the SMT solver.

To enhance the modelling of program semantics, API functions have to be handled. Although it has been shown that API call inlining is inefficient, the area of application can be extended by preparing common API functions for inlining; this is especially true, if inlining is the only way to model the semantics of an API function. As mentioned in Section 3.2.2, SMT-based modelling may be a more efficient and architecture-independent method of handling API functions. The modelling of `malloc` and `free` is of special interest because it allows for the modelling of dynamic memory allocations. Sinz, Falke and Merz [38] demonstrated this in their LLBMC.

In relation to loop unrolling, loop detection will be performed on the basis of strongly connected components, as described in Section 3.3.2. Therefore, the loop structures of high-level programming languages will not be considered because only outermost loops will be detected. To consider the high-level structure and loop nesting, loop nesting trees [42] can be utilised for loop unrolling; nested loops will be unrolled iteratively, starting from the innermost loop and ending with the outermost loop. As a result, the SMT solver may exploit the loop structure more efficiently.

5.2 Future research

In this dissertation, input crafting has been applied to bounded model checking on the binary level as a postcondition. However, arbitrary postconditions can be applied to an SMT formula that represents the logical encoding of a binary function. Building upon the strengths of SMT solvers, bounded model checking may operate powerfully in other areas related to binary application security.

Section 4.6 demonstrated that SMT solvers are very efficient in path selection on the basis of opaque predicates, a common technique in the context of program obfuscation [90]. In addition, SMT solvers have been used to synthesise programs [94] as well as for equivalence checking [12]. Therefore, bounded model checking on the binary level may be the groundwork for SMT-based obfuscation, deobfuscation, equivalence checking and program synthesis.

Sinz, Falke and Merz [38] introduced bounded model checking on the basis of LLVM's intermediate representation in order to locate integer overflows, illegal memory access and other vulnerabilities in C programs. This approach may be applied to the binary level. Furthermore, since arbitrary conditions can be encoded as SMT formulas, it is possible to define the values of arbitrary memory locations for a specific program state. For instance, this may be used to craft inputs that write malicious code into defined memory areas. Therefore, the approach of vulnerability discovery may be extended to automatic exploit generation on the basis of bounded model checking.

A Contributions to Miasm

Date	Git commit	Description
24.04.2015	4ff91550b953a661abaa49b936ae76a6b955df9f	Submitted bugs in the Z3 translation.
25.03.2015	f72d0de75c6815db54f9d54824e8948d962eee5a	Z3 translation has been added for `idiv`.
25.03.2015	31b70a4a6dc770bb21db97cdac27f7f1e54055d7	The x86 semantics have been added for `pxor`.
01.05.2015	f9bf6fbbce0d984549fb11e16cb9acfc7be00c86	Redundant code has been removed and typography has been improved.
03.06.2015	eb85ae0db115b40190446c0aa841e8a1b562eece	Graph algorithms have been added to compute immediate dominators and the dominance frontier.
06.07.2015	35852d91259bcac6f15c5db4edf2ccf50f00f444	Submitted a sandbox class for x86_64 Linux.
29.07.2015	817fc666eac74c802d4d592f50a3872a3197f4a5	Z3 translations have been added for `udiv`, `imod` and `umod`.
12.08.2015	4d93231e77a8af094c198a8b3b3733b0250cd45a	Fixed a link in the documentation.
06.09.2015	cc20f3d79c641d929865f12471a70a2575b8cf54	Submitted graph walks, natural loop detection and the computation of strongly connected components.
16.10.2015	861e0dc047b3a6675aa8a9b131a53cb6d4dd033f	Z3 translations have been added for left/right rotations.

Table A.1: Contributions to Miasm, sorted by date of git merges

B Listings

B.1 Case study 2

```
int bar(int a, int b)
{
    int ar[10];

    ar[0] = 3;
    ar[1] = a;
    ar[2] = 2 * a + b;
    ar[3] = b << 3;
    ar[4] = b + a;
    ar[5] = 55 * a;
    ar[6] = ar[5];
    ar[7] = a + 1337;
    ar[8] = ar[0] + ar[4];
    ar[9] = 999;

    if (ar[0] + 2 * ar[4] + ar[3] >> ar[0] == 1337)
    {
        ar[ar[0] + 2] = 21;
        ar[3] = 11;
        ar[9] = a * ar[2];
    }

    else
    {
        ar[3] = ar[1] + ar[0];
        ar[7] = 11;
    }

    if (ar[9] == ar[7] + ar[3] + 123)
        return ar[9] + ar[3];
    else
        return ar[9] - ar[3];
}

int foo(int a, int b)
{
    int ar[3];

    ar[0] = 3;
    ar[1] = a;
    ar[2] = 2 * a + b;


    if (ar[0] + ar[2] == 1337)
    {
        ar[0] = 1;
        ar[1] = 2;
        ar[2] = a + b;
    }

    else
```

```
52      {
53          ar[1] = ar[0] + ar[1];
54          ar[0] = ar[1] * b;
55      }
56      if (ar[0] + b + ar[1] == 1337 + bar(a + b , ar[0]))
57          return 1;
58
59      return 0;
60  }
```

Listing B.1: Nested array operations and a function call

B.2 Case study 3

```
1   #include<stdlib.h>
2   #include<stdio.h>
3
4   int bar(int a, int b);
5
6   int foo(int a, int b, int c)
7   {
8       int ar[3];
9       ar[0] = a + b + c;
10      ar[1] = a * b;
11      ar[2] = ar[0] + c * ar[1] - (b * b) + 1;
12      int d = (a * b) << c;
13
14      if (ar[2] + ar[1] * c + a == 1337 + d)
15      {
16          a = b;
17          b = a * c;
18          c = (a + b) * d;
19          if (a + b == 200)
20              return 2;
21
22          else if (a + b)
23          {
24              c = c + 1;
25              d = b + 2;
26              if (b + d + a == c * b + 300)
27              {
28                  int i = 0;
29                  while (i < 4)
30                  {
31                      i++;
32                      if ((i + c > 100) && (i > 2))
33                          return 3;
34                  }
35                  if (ar[2] + ar[1])
36                  {
37                      if ((a < b) || (c > d))
38                          return bar(a, b);
39                      return 5;
40                  }
41                  else
42                  {
43                      if (a < b)
44                          return 6;
```

```
45                  return 7;
46                  }
47              }
48          }
49          return 1;
50      }
51      return 0;
52  }
53
54  int bar(int a, int b)
55  {
56      int d = a * b;
57      a = b * b;
58      d = 2;
59      b = d * 2;
60
61      return b;
62  }
63
64  int main(int argc, char * argv[])
65  {
66      int a,b,c;
67
68      a = atoi(argv[1]);
69      b = atoi(argv[2]);
70      c = atoi(argv[3]);
71
72      if (foo(a, b, c) == 4)
73      {
74          printf("Correct!\n");
75          return 0;
76      }
77      else
78      {
79          printf("Wrong!\n");
80          return -1;
81      }
82  }
```

Listing B.2: Nested conditions with a loop and a function call

B.3 Case study 4

```
1   import sys
2   import angr
3   import simuvex
4
5   if len(sys.argv) != 4:
6       print "[*] Syntax: <filename> <start address> <end address>"
7       sys.exit(0)
8
9   # sample
10  f = sys.argv[1]
11  start_addr = int(sys.argv[2], 16)
12  end_addr = int(sys.argv[3], 16)
13
14  b = angr.Project(f, load_options={"auto_load_libs": False})
15  initial_state = b.factory.blank_state(addr=start_addr,
```

```
                                    remove_options={simuvex.o.LAZY_SOLVES})

pg = b.factory.path_group(initial_state, immutable=False)
pg.explore(find=end_addr)

if pg.found:
    found_state = pg.found[0].state
    print "A satisfying path has been found."
```

Listing B.3: Path selection with angr

B.4 Case study 5

```
0x00415f3c  push rbp
0x00415f3d  mov rbp, rsp
0x00415f40  sub rsp, 0x10
0x00415f44  mov qword [rbp - 8], rdi
0x00415f48  mov qword [rbp - 0x10], rsi
0x00415f4c  mov rdx, qword [rbp - 0x10]
0x00415f50  mov rax, qword [rbp - 8]
0x00415f54  mov rsi, rdx
0x00415f57  mov rdi, rax
0x00415f5a  call sym.hwaddr_parse
0x00415f5f  test rax, rax
0x00415f62  je 0x415f6b
0x00415f64  mov eax, 0
0x00415f69  jmp 0x415f70
0x00415f6b  mov eax, 0xffffffff
0x00415f70  leave
0x00415f71  ret
```

Listing B.4: WPA supplicant: `hwaddr_aton`

```
0x00415ebe  push rbp
0x00415ebf  mov rbp, rsp
0x00415ec2  sub rsp, 0x20
0x00415ec6  mov qword [rbp - 0x18], rdi
0x00415eca  mov qword [rbp - 0x20], rsi
0x00415ece  mov qword [rbp - 8], 0
0x00415ed6  jmp 0x415f2f
0x00415ed8  mov rax, qword [rbp - 0x18]
0x00415edc  mov rdi, rax
0x00415edf  call sym.hex2byte
0x00415ee4  mov dword [rbp - 0xc], eax
0x00415ee7  cmp dword [rbp - 0xc], 0
0x00415eeb  jns 0x415ef4
0x00415eed  mov eax, 0
0x00415ef2  jmp 0x415f3a
0x00415ef4  add qword [rbp - 0x18], 2
0x00415ef9  mov rdx, qword [rbp - 0x20]
0x00415efd  mov rax, qword [rbp - 8]
0x00415f01  add rax, rdx
0x00415f04  mov edx, dword [rbp - 0xc]
0x00415f07  mov byte [rax], dl
0x00415f09  cmp qword [rbp - 8], 4
0x00415f0e  ja 0x415f2a
0x00415f10  mov rax, qword [rbp - 0x18]
```

```
0x00415f14  lea rdx, qword [rax + 1]
0x00415f18  mov qword [rbp - 0x18], rdx
0x00415f1c  movzx eax, byte [rax]
0x00415f1f  cmp al, 0x3a
0x00415f21  je 0x415f2a
0x00415f23  mov eax, 0
0x00415f28  jmp 0x415f3a
0x00415f2a  add qword [rbp - 8], 1
0x00415f2f  cmp qword [rbp - 8], 5
0x00415f34  jbe 0x415ed8
0x00415f36  mov rax, qword [rbp - 0x18]
0x00415f3a  leave
0x00415f3b  ret
```

Listing B.5: WPA supplicant: `hwaddr_parse`

```
0x00415e55  push rbp
0x00415e56  mov rbp, rsp
0x00415e59  sub rsp, 0x18
0x00415e5d  mov qword [rbp - 0x18], rdi
0x00415e61  mov rax, qword [rbp - 0x18]
0x00415e65  lea rdx, qword [rax + 1]
0x00415e69  mov qword [rbp - 0x18], rdx
0x00415e6d  movzx eax, byte [rax]
0x00415e70  movsx eax, al
0x00415e73  mov edi, eax
0x00415e75  call sym.hex2num
0x00415e7a  mov dword [rbp - 4], eax
0x00415e7d  cmp dword [rbp - 4], 0
0x00415e81  jns 0x415e8a
0x00415e83  mov eax, 0xffffffff
0x00415e88  jmp 0x415ebc
0x00415e8a  mov rax, qword [rbp - 0x18]
0x00415e8e  lea rdx, qword [rax + 1]
0x00415e92  mov qword [rbp - 0x18], rdx
0x00415e96  movzx eax, byte [rax]
0x00415e99  movsx eax, al
0x00415e9c  mov edi, eax
0x00415e9e  call sym.hex2num
0x00415ea3  mov dword [rbp - 8], eax
0x00415ea6  cmp dword [rbp - 8], 0
0x00415eaa  jns 0x415eb3
0x00415eac  mov eax, 0xffffffff
0x00415eb1  jmp 0x415ebc
0x00415eb3  mov eax, dword [rbp - 4]
0x00415eb6  shl eax, 4
0x00415eb9  or eax, dword [rbp - 8]
0x00415ebc  leave
0x00415ebd  ret
```

Listing B.6: WPA supplicant: `hex2byte`

```
0x00415e06  push rbp
0x00415e07  mov rbp, rsp
0x00415e0a  mov eax, edi
0x00415e0c  mov byte [rbp - 4], al
0x00415e0f  cmp byte [rbp - 4], 0x2f
0x00415e13  jle 0x415e24
0x00415e15  cmp byte [rbp - 4], 0x39
0x00415e19  jg 0x415e24
0x00415e1b  movsx eax, byte [rbp - 4]
```

```
0x00415e1f  sub eax, 0x30
0x00415e22  jmp 0x415e53
0x00415e24  cmp byte [rbp - 4], 0x60
0x00415e28  jle 0x415e39
0x00415e2a  cmp byte [rbp - 4], 0x66
0x00415e2e  jg 0x415e39
0x00415e30  movsx eax, byte [rbp - 4]
0x00415e34  sub eax, 0x57
0x00415e37  jmp 0x415e53
0x00415e39  cmp byte [rbp - 4], 0x40
0x00415e3d  jle 0x415e4e
0x00415e3f  cmp byte [rbp - 4], 0x46
0x00415e43  jg 0x415e4e
0x00415e45  movsx eax, byte [rbp - 4]
0x00415e49  sub eax, 0x37
0x00415e4c  jmp 0x415e53
0x00415e4e  mov eax, 0xffffffff
0x00415e53  pop rbp
0x00415e54  ret
```

Listing B.7: WPA supplicant: `hex2num`

Bibliography

[1] E. Eilam. *Reversing: Secrets of Reverse Engineering*. John Wiley & Sons, 2011.

[2] A. J. Menezes, S. A. Vanstone and P. C. V. Oorschot. *Handbook of Applied Cryptography*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 1996.

[3] T. Avgerinos et al. 'AEG: Automatic Exploit Generation'. In: *NDSS*. Vol. 11. 2011, pp. 59–66.

[4] A. V. Aho et al. *Compilers: Principles, Techniques, & Tools*. 2nd. USA: Addison-Wesley Publishing Company, 2007.

[5] E. J. Schwartz, T. Avgerinos and D. Brumley. 'All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)'. In: *IEEE Symposium on Security and Privacy 2010*. IEEE. 2010, pp. 317–331.

[6] S. Gulwani, S. Srivastava and R. Venkatesan. 'Program analysis as constraint solving'. In: *ACM SIGPLAN Notices*. Vol. 43. 6. ACM. 2008, pp. 281–292.

[7] C. Cadar, D. Dunbar and D. Engler. 'KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs'. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 209–224.

[8] P. Godefroid, M. Y. Levin and D. Molnar. 'SAGE: whitebox fuzzing for security testing'. In: *Queue* 10.1 (2012), p. 20.

[9] C. W. Barrett et al. 'Satisfiability Modulo Theories.' In: *Handbook of satisfiability* 185 (2009), pp. 825–885.

[10] L. De Moura and N. Bjørner. 'Satisfiability modulo theories: An appetizer'. In: *Formal Methods: Foundations and Applications*. Springer, 2009, pp. 23–36.

[11] J. Newsome and D. Song. 'Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software'. In: *12th Annual Network and Distributed System Security Symposium*. 2005.

[12] J. Vanegue, S. Heelan and R. Rolles. 'SMT Solvers in Software Security.' In: *WOOT*. 2012, pp. 85–96.

[13] A. Armando, J. Mantovani and L. Platania. 'Bounded model checking of software using SMT solvers instead of SAT solvers'. In: *Model Checking Software*. Springer, 2006, pp. 146–162.

[14] R. S. Boyer, B. Elspas and K. N. Levitt. 'SELECT—a formal system for testing and debugging programs by symbolic execution'. In: *ACM SigPlan Notices* 10.6 (1975), pp. 234–245.

[15] J. C. King. 'Symbolic execution and program testing'. In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[16] P. Coward. 'Symbolic execution and testing'. In: vol. 33. 1. Elsevier, 1991, pp. 53–64.

[17] W.-T. Tsai, D. Volovik and T. F. Keefe. 'Automated test case generation for programs specified by relational algebra queries'. In: *IEEE Transactions on Software Engineering* 3 (1990), pp. 316–324.

[18] C. Lattner and V. Adve. 'LLVM: A compilation framework for lifelong program analysis & transformation'. In: *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE. 2004, pp. 75–86.

[19] D. Kroening, R. Bryant and O. Strichman. *Decision Procedures: An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg, 2008.

[20] L. De Moura and N. Bjørner. 'Z3: An efficient SMT solver'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[21] R. Brummayer and A. Biere. 'Boolector: An efficient SMT solver for bit-vectors and arrays'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2009, pp. 174–177.

[22] S. K. Cha et al. 'Unleashing Mayhem on Binary Code'. In: *IEEE Symposium on Security and Privacy 2012*. IEEE. 2012, pp. 380–394.

[23] D. Babić et al. 'Statically-directed dynamic automated test generation'. In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM. 2011, pp. 12–22.

[24] Y. Shoshitaishvili et al. 'Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware'. In: *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*. 2015.

[25] angr. *angr, a binary analysis framework*. URL: `http://angr.io` (visited on 26/11/2015).

[26] A. Biere et al. 'Symbolic model checking using SAT procedures instead of BDDs'. In: *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*. ACM. 1999, pp. 317–320.

[27] E. M. Clarke and E. A. Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic*. Springer, 1982.

[28] K. L. M. Millan. 'Symbolic Model Checking: An approach to the state explosion problem'. PhD thesis. Carnegie Mellon University (CMU), 1992.

[29] E. Clarke, D. Kroening and F. Lerda. 'A tool for checking ANSI-C programs'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2004, pp. 168–176.

[30] B. Schlich and S. Kowalewski. 'Model checking C source code for embedded systems'. In: *International journal on software tools for technology transfer* 11.3 (2009), pp. 187–202.

[31] I. Rabinovitz and O. Grumberg. 'Bounded model checking of concurrent programs'. In: *In Computer-Aided Verification (CAV), LNCS 3576*. Springer, 2005, pp. 82–97.

[32] B. Schlich. 'Model checking of software for microcontrollers'. In: *ACM Transactions on Embedded Computing Systems (TECS)* 9.4 (2010), p. 36.

[33] J. Kobashi, S. Yamane and A. Takeshita. 'Development of SMT-Based Bounded Model Checker for embedded assembly program'. In: *IEEE 3rd Global Conference on Consumer Electronics (GCCE 2014)*. IEEE. 2014, pp. 696–698.

[34] A. Armando and L. Compagna. 'SATMC: A SAT-Based Model Checker for Security Protocols'. In: *Logics in Artificial Intelligence*. Ed. by J. Alferes and J. Leite. Vol. 3229. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 730–733.

[35] A. Armando and L. Compagna. 'SAT-based model-checking for security protocols analysis'. In: *International Journal of Information Security* 7.1 (2008), pp. 3–32.

[36] Y.-W. Huang et al. 'Verifying web applications using bounded model checking'. In: *International Conference on Dependable Systems and Networks (DSN-2014)*. IEEE. 2004, pp. 199–208.

[37] F. Besson et al. 'Model checking security properties of control flow graphs'. In: *Journal of computer security* 9.3 (2001), pp. 217–250.

[38] C. Sinz, S. Falke and F. Merz. 'A precise memory model for low-level bounded model checking'. In: *Proceedings of the 5th international conference on Systems software verification*. USENIX Association. 2010.

[39] R. G. V. meets Algorithm Engineering. *LLBMC – The Low-Level Bounded Model Checker*. URL: `http://llbmc.org` (visited on 26/11/2015).

[40] J. Bang-Jensen and G. Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer London, 2013.

[41] R. Cytron et al. 'An efficient method of computing static single assignment form'. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1989, pp. 25–35.

[42] P. Havlak. 'Nesting of reducible and irreducible loops'. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19.4 (1997), pp. 557–567.

[43] H. N. Gabow. 'Path-based depth-first search for strong and biconnected components'. In: *Information Processing Letters* 74.3 (2000), pp. 107–114.

[44] K. D. Cooper, T. J. Harvey and K. Kennedy. 'A simple, fast dominance algorithm'. In: *Software Practice & Experience* 4 (2001), pp. 1–10.

[45] S. Unger and F. Mueller. *Handling irreducible loops: Optimized node splitting vs. dj-graphs*. Springer, 2001.

[46] F. Nielson, H. Nielson and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[47] B. Korel. 'Automated software test data generation'. In: *IEEE Transactions on Software Engineering* 16.8 (1990), pp. 870–879.

[48] C. V. Ramamoorthy, S.-B. F. Ho and W. Chen. 'On the automated generation of program test data'. In: *IEEE Transactions on Software Engineering* 4 (1976), pp. 293–300.

[49] Q. Yi et al. 'Postconditioned Symbolic Execution'. In: *IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 2015, pp. 1–10.

[50] W. Chang, B. Streiff and C. Lin. 'Efficient and extensible security enforcement using dynamic data flow analysis'. In: *Proceedings of the 15th ACM conference on Computer and communications security*. ACM. 2008, pp. 39–50.

[51] U. Khedker, A. Sanyal and B. Sathe. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009.

[52] A. Bradley and Z. Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer Berlin Heidelberg, 2007.

[53] R. Nieuwenhuis, A. Oliveras and C. Tinelli. 'Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL (T)'. In: *Journal of the ACM (JACM)* 53.6 (2006), pp. 937–977.

[54] C. Barrett, P. Fontaine and C. Tinelli. *The SMT-LIB Standard: Version 2.5*. Tech. rep. Department of Computer Science, The University of Iowa, 2015. URL: `http://smtlib.cs.uiowa.edu` (visited on 26/11/2015).

[55] SMT-LIB. *Logics*. URL: `http://smtlib.cs.uiowa.edu/logics.shtml` (visited on 26/11/2015).

[56] SMT-LIB. *QF_BV*. URL: `http://smtlib.cs.uiowa.edu/logics-all.shtml#QF_BV` (visited on 26/11/2015).

[57] L. Hadarean. 'An Efficient and Trustworthy Theory Solver for Bit-vectors in Satisfiability Modulo Theories'. PhD thesis. New York University, 2015.

[58] A. Franzén. 'Efficient solving of the satisfiability modulo bit-vectors problem and some extensions to SMT'. PhD thesis. University of Trento, 2010.

[59] S. Falke, C. Sinz and F. Merz. 'A Theory of Arrays with set and copy Operations.' In: *SMT@ IJCAR*. 2012, pp. 98–108.

[60] C. I. Security. *Miasm2*. URL: `https://github.com/cea-sec/miasm` (visited on 26/11/2015).

[61] *GNU General Public License*. Free Software Foundation. URL: https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html (visited on 26/11/2015).

[62] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*. Aug. 2012.

[63] ARM Limited. *ARM Architecture Reference Manual – ARMv7-A and ARMv7-R edition*. 2012.

[64] ARM Limited. *ARM Architecture Reference Manual – ARMv8, for ARMv8-A architecture profile*. 2013.

[65] MIPS Technologies, Inc. *MIPS32^{TM} Architecture For Programmers – Volume I: Introduction to the MIPS32^{TM} Architecture*. 2001.

[66] F. Bellard. *Tiny C Compiler*. URL: http://bellard.org/tcc (visited on 26/11/2015).

[67] F. Desclaux. *Miasm : Framework de reverse engineering*. Symposium sur la sécurité des technologies de l'information et des communications (SSTIC 2012). 2012. URL: https://www.sstic.org/media/SSTIC2012/SSTIC-actes/miasm_framework_de_reverse_engineering/SSTIC2012-Article-miasm_framework_de_reverse_engineering-desclaux_1.pdf (visited on 26/11/2015).

[68] serpilliere and commial. *miasm2/expression/expressions.py*. Miasm2 source code. URL: https://github.com/cea-sec/miasm/blob/master/miasm2/expression/expression.py (visited on 26/11/2015).

[69] commial. *Order of IR instructions and memory access*. 2015. URL: https://github.com/cea-sec/miasm/issues/194 (visited on 26/11/2015).

[70] commial. *IRDst is erroneous in some cases*. 2015. URL: https://github.com/cea-sec/miasm/issues/86#issuecomment-75729194 (visited on 26/11/2015).

[71] Hex-Rays. *IDA*. URL: https://www.hex-rays.com/products/ida/index.shtml (visited on 26/11/2015).

[72] C. Cifuentes. 'Reverse compilation techniques'. PhD thesis. Queensland University of Technology, 1994.

[73] T. Bao et al. 'BYTEWEIGHT: Learning to Recognize Functions in Binary Code'. In: *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. 2014, pp. 845–860.

[74] pancake et al. *radare2*. URL: http://www.radare.org/r (visited on 26/11/2015).

[75] C. Eagle. *The IDA Pro Book, 2nd Edition: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press Series. No Starch Press, 2011.

[76] T. I. S. T. Committee. *Executable and Linking Format (ELF) – Specification – Version 1.2*. 1995. URL: http://refspecs.linuxbase.org/elf/elf.pdf (visited on 26/11/2015).

[77] M. Corporation. *Microsoft Portable Executable and Common Object File Format Specification – Revision 8.3*. 2013. URL: https://msdn.microsoft.com/en-us/windows/hardware/gg463119 (visited on 26/11/2015).

[78] V. Ganapathy et al. 'Automatic discovery of API-level exploits'. In: *Proceedings of the 27th international conference on Software engineering*. ACM. 2005, pp. 312–321.

[79] L. Null, P. Null and J. Lobur. *The Essentials of Computer Organization and Architecture*. Jones & Bartlett Learning, LLC, 2014.

[80] *Arch Linux*. URL: https://www.archlinux.org (visited on 26/11/2015).

[81] *GCC, the GNU Compiler Collection*. Free Software Foundation. URL: https://gcc.gnu.org (visited on 26/11/2015).

[82] *GDB: The GNU Project Debugger*. Free Software Foundation. URL: https://www.gnu.org/software/gdb/gdb.html (visited on 26/11/2015).

[83] *QEMU*. URL: http://www.qemu.org (visited on 26/11/2015).

[84] A. Niemetz, M. Preiner and A. Biere. 'Boolector 2.0'. In: *Journal of Satisfiability, Boolean Modeling and Computation (JSAT)* 9 (2015), pp. 53–58. URL: https://satassociation.org/jsat/index.php/jsat/article/view/120 (visited on 26/11/2015).

[85] S. Conchon, D. Déharbe and T. Weber. *10th International Satisfiability Modulo Theories Competition (SMT-COMP 2015)*. 2015. URL: http://smtcomp.sourceforge.net/2015/index.shtml (visited on 26/11/2015).

[86] F. von Leitner. *diet libc – a libc optimized for small size*. URL: http://www.fefe.de/dietlibc/ (visited on 26/11/2015).

[87] P. Lancaster and K. Šalkauskas. *Curve and surface fitting: an introduction*. Computational mathematics and applications. Academic Press, 1986.

[88] *QF_ABV (Main Track)*. 10th International Satisfiability Modulo Theories Competition (SMT-COMP 2015). 2015. URL: http://smtcomp.sourceforge.net/2015/results-QF_ABV.shtml (visited on 26/11/2015).

[89] V. Ganesh and D. L. Dill. 'A decision procedure for bit-vectors and arrays'. In: *Computer Aided Verification*. Springer. 2007, pp. 519–531.

[90] M. Dalla Preda et al. 'Opaque predicates detection by abstract interpretation'. In: *Algebraic methodology and software technology*. Springer, 2006, pp. 81–95.

[91] *Coreutils – GNU core utilities*. Free Software Foundation. URL: http://www.gnu.org/software/coreutils/coreutils.html (visited on 26/11/2015).

[92] J. Malinen. *Linux WPA/WPA2/IEEE 802.1X Supplicant*. URL: http://w1.fi/wpa_supplicant (visited on 26/11/2015).

[93] M. J. Van Emmerik. 'Static single assignment for decompilation'. PhD thesis. The University of Queensland, 2007.

[94] S. Srivastava, S. Gulwani and J. S. Foster. 'Template-based program verification and program synthesis'. In: *International Journal on Software Tools for Technology Transfer* 15.5-6 (2013), pp. 497–518.